Communications System Toolbox™

Reference

MATLAB®&SIMULINK®

R2018a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
☎ Phone:	508-647-7000

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

Communications System Toolbox ™ Reference

Trademarks

Patents

Revision History

April 2011	First printing	New for Version 5.0
September 2011	Online only	Revised for Version 5.1 (R2011b)
March 2012	Online only	Revised for Version 5.2 (R2012a)
September 2012	Online only	Revised for Version 5.3 (R2012b)
March 2013	Online only	Revised for Version 5.4 (R2013a)
September 2013	Online only	Revised for Version 5.5 (R2013b)
March 2014	Online only	Revised for Version 5.6 (R2014a)
October 2014	Online only	Revised for Version 5.7 (R2014b)
March 2015	Online only	Revised for Version 6.0 (R2015a)
September 2015	Online only	Revised for Version 6.1 (R2015b)
March 2016	Online only	Revised for Version 6.2 (R2016a)
September 2016	Online only	Revised for Version 6.3 (R2016b)
March 2017	Online only	Revised for Version 6.4 (R2017a)
September 2017	Online only	Revised for Version 6.5 (R2017b)
March 2018	Online only	Revised for Version 6.6 (Release 2018a)

Contents

1

Functions — Alphabetical List

algdeintrlv

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

Interleave and Deinterleave Symbols

This example uses the Takeshita-Costello method of algintrlv and algdeintrlv.

Generate random data symbols to interleave. The number of rows of input data, num, must be a power of two.

```
num = 16;
ncols = 3;
data = rand(num,ncols)
```

```
data = 16×3

    0.8147    0.4218    0.2769
    0.9058    0.9157    0.0462
    0.1270    0.7922    0.0971
    0.9134    0.9595    0.8235
    0.6324    0.6557    0.6948
    0.0975    0.0357    0.3171
    0.2785    0.8491    0.9502
    0.5469    0.9340    0.0344
    0.9575    0.6787    0.4387
    0.9649    0.7577    0.3816
         ⋮
```

Interleave the symbols using the Takeshita-Costello method. Set the multiplicative factor, k, to an odd integer less than num, and the cyclic shift, h, to a nonnegative integer less than num.

```
k = 3;
h = 4;
intdata = algintrlv(data,num,'takeshita-costello',k,h)
```

```
intdata = 16×3

    0.9572    0.6555    0.1869
    0.2785    0.8491    0.9502
    0.1576    0.7431    0.7655
    0.0975    0.0357    0.3171
    0.8147    0.4218    0.2769
    0.1270    0.7922    0.0971
    0.9058    0.9157    0.0462
    0.9575    0.6787    0.4387
    0.5469    0.9340    0.0344
    0.1419    0.0318    0.6463
         ⋮
```

Deinterleave the symbols to obtain the original order.

```
deintdata = algdeintrlv(intdata,num,'takeshita-costello',k,h)
```

```
deintdata = 16×3

    0.8147    0.4218    0.2769
    0.9058    0.9157    0.0462
    0.1270    0.7922    0.0971
    0.9134    0.9595    0.8235
    0.6324    0.6557    0.6948
    0.0975    0.0357    0.3171
    0.2785    0.8491    0.9502
    0.5469    0.9340    0.0344
    0.9575    0.6787    0.4387
    0.9649    0.7577    0.3816
       ⋮
```

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`

Topics

“Interleaving”

Introduced before R2006a

algintrlv

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

Reorder Symbols Using Algebraically Derived Permutation Table

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

Define `num` such that `num+1` is prime. Create data to interleave.

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

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)
```

```
pr = 4×2

     3     1
     4     1
     5     1
     9     1
```

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 = 4×1

     8
     7
     6
     2
```

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

```
alph = primel(1);
intrlvd = algintrlv(data,num,'Welch-Costas',alph);
```

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`

Topics

“Interleaving”

Introduced before R2006a

amdemod

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

Demodulate AM Signal

Set the sample rate and carrier frequency.

```
fc = 10e3;
fs = 80e3;
```

Generate a sinusoidal signal having a 0.01 s duration.

```
t = [0:1/fs:0.01]';
s = sin(2*pi*300*t)+2*sin(2*pi*600*t);
```

Create a lowpass filter.

```
[num,den] = butter(10,fc*2/fs);
```

Amplitude modulate the signal, `s`.

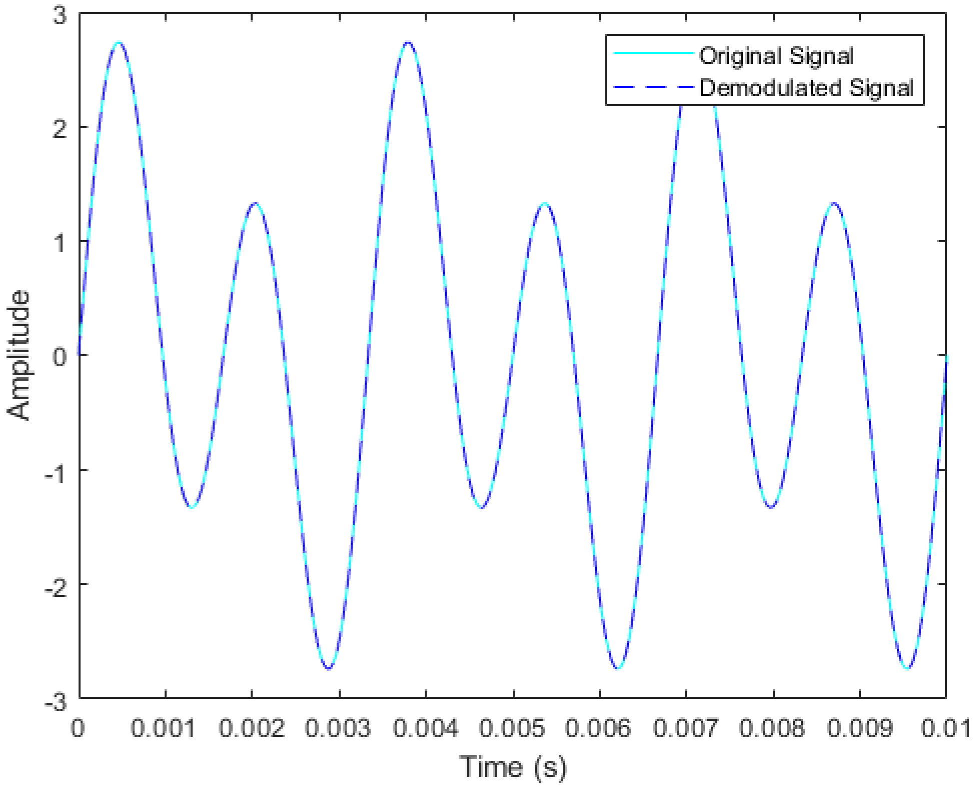
```
y = ammod(s,fc,fs);
```

Demodulate the received signal.

```
s1 = amdemod(y,fc,fs,0,0,num,den);
```

Plot the original and demodulated signals.

```
plot(t,s,'c',t,s1,'b--')
legend('Original Signal','Demodulated Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



The demodulated signal is nearly identical to the original signal.

See Also

`ammod` | `fmdemod` | `pmdemod` | `ssbdemod`

Topics

“Digital Modulation”

Introduced before R2006a

ammod

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

Compare Double-Sideband and Single-Sideband Amplitude Modulation

Set the sample rate to 100 Hz. Create a time vector 100 seconds long.

```
fs = 100;
t = (0:1/fs:100)';
```

Set the carrier frequency to 10 Hz. Generate a sinusoidal signal.

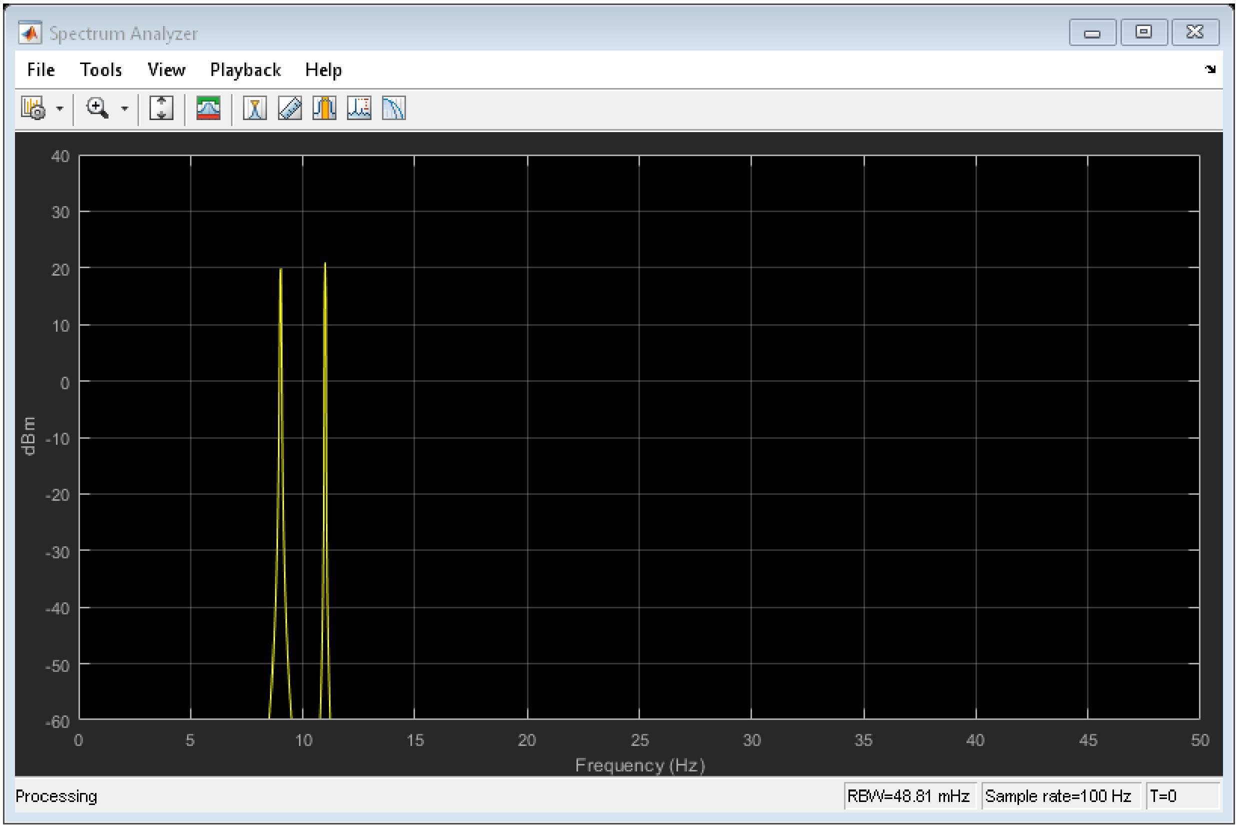
```
fc = 10;
x = sin(2*pi*t);
```

Modulate `x` using single- and double-sideband AM.

```
ydouble = ammod(x,fc,fs);
ysingle = ssbmod(x,fc,fs);
```

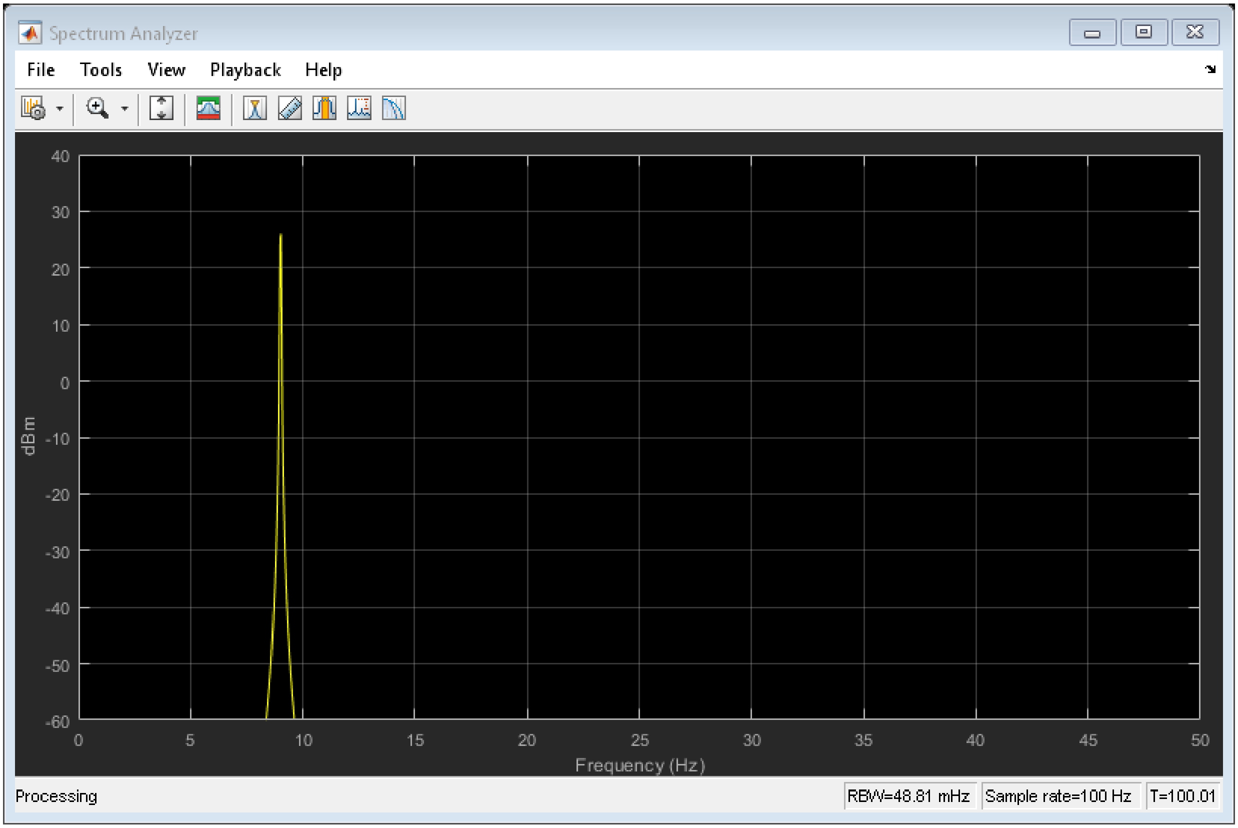
Create a spectrum analyzer object to plot the spectra of the two signals. Plot the spectrum of the double-sideband signal.

```
sa = dsp.SpectrumAnalyzer('SampleRate',fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'YLimits',[-60 40]);
step(sa,ydouble)
```



Plot the single-sideband spectrum.

```
step(sa,ysingle)
```



See Also

`amdemod` | `fmmod` | `pmmod` | `ssbmod`

Topics

"Digital Modulation"

Introduced before R2006a

apskdemod

Amplitude phase shift keying (APSK) demodulation

Syntax

```
z = apskdemod(y,M,radii)
z = apskdemod(y,M,radii,phaseoffset)
z = apskdemod( ___ ,Name,Value)
```

Description

`z = apskdemod(y,M,radii)` performs APSK demodulation of the input signal `y`, based on the specified number of constellation points per PSK ring, `M`, and the radius of each PSK ring, `radii`. For a description of APSK demodulation, see "Algorithms" on page 1-26.

`z = apskdemod(y,M,radii,phaseoffset)` specifies an initial phase offset for each PSK ring of the APSK modulated signal.

`z = apskdemod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate 16-APSK Signal

Demodulate a 16-APSK signal that has an unequal number of constellation points on each circle. Plot the received constellation.

Define vectors for modulation order and PSK ring radii. Generate random 16-ary data symbols.

```
M = [4 12];
radii = [1 2];
modOrder = sum(M);

x = randi([0 modOrder-1],1000,1);
```

Apply APSK modulation to the data.

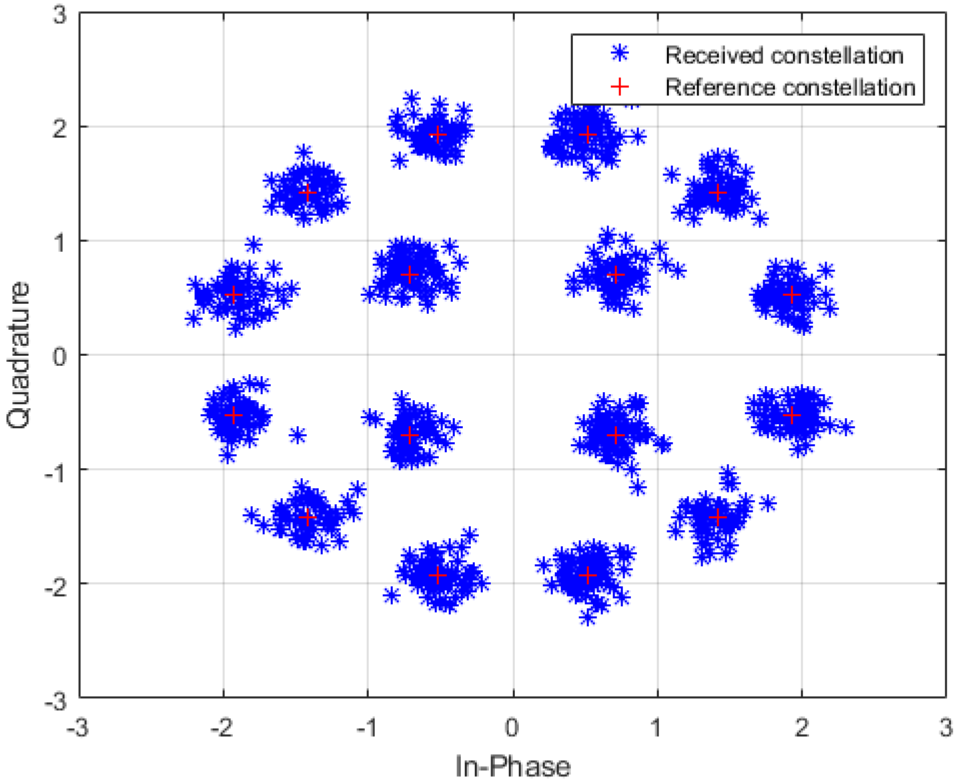
```
txSig = apskmod(x,M,radii);
```

Pass the modulated signal through a noisy channel.

```
snr = 20; % dB
rxSig = awgn(txSig,snr,'measured');
```

Plot the transmitted (reference) signal points and the noisy received signal points.

```
plot(rxSig,'b*')
hold on
grid
plot(txSig,'r+')
xlim([-3 3])
ylim([-3 3])
xlabel('In-Phase')
ylabel('Quadrature')
legend('Received constellation','Reference constellation')
```



Demodulate the received signal and compare to the input data.

```
z = apskdemod(rxSig,M,radii);
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate 64-APSK Custom Symbol Mapped Signal

Demodulate a 64-APSK signal with custom symbol mapping. Compute hard decision bit output and verify that the input matches the output.

Define vectors for modulation order and PSK ring radii. Generate 100 symbols of random bit input.

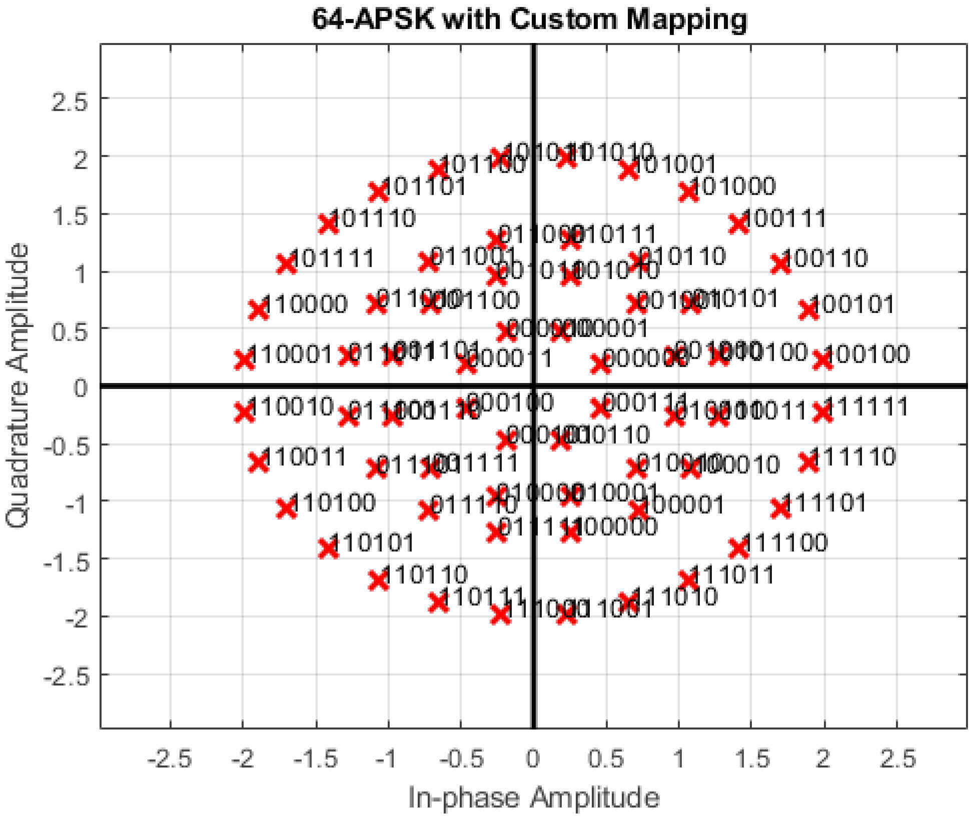
```
M = [8 12 16 28]; % 4-PSK circles
modOrder = sum(M);
radii = [0.5 1 1.3 2];
x = randi([0 1],100*log2(modOrder),1);
```

Create a custom symbol mapping vector of binary mapping.

```
cmap = 0:63;
```

Modulate the data and plot the constellation.

```
y = apskmod(x,M,radii,'SymbolMapping',cmap,'inputType','bit', ...
    'PlotConstellation',true);
```



Demodulate the received signal.

```
z = apskdemod(y,M,radii,'SymbolMapping',cmap,'OutputType','bit');
```

Verify that the demodulated signal is equal to the original data.

```
isequal(x,z)
```

```
ans = logical
   1
```

Soft Bit Demodulate 32-APSK Signal

Demodulate a 32-APSK signal and calculate soft bits.

Define vectors for modulation order and PSK ring radii. Generate 10000 symbols of random bit data.

```
M = [16 16];
modOrder = sum(M);
radii = [0.6 1.2];
numSym = 10000;
x = randi([0 1], numSym*log2(modOrder),1);
```

Generate a reference constellation. Create a constellation diagram object.

```
refAPSK = apskmod(0:modOrder-1,M,radii);
constDiagAPSK = comm.ConstellationDiagram('ReferenceConstellation',refAPSK, ...
  'Title','Received Symbols','XLimits',[-2 2],'YLimits',[-2 2]);
```

Modulate the data.

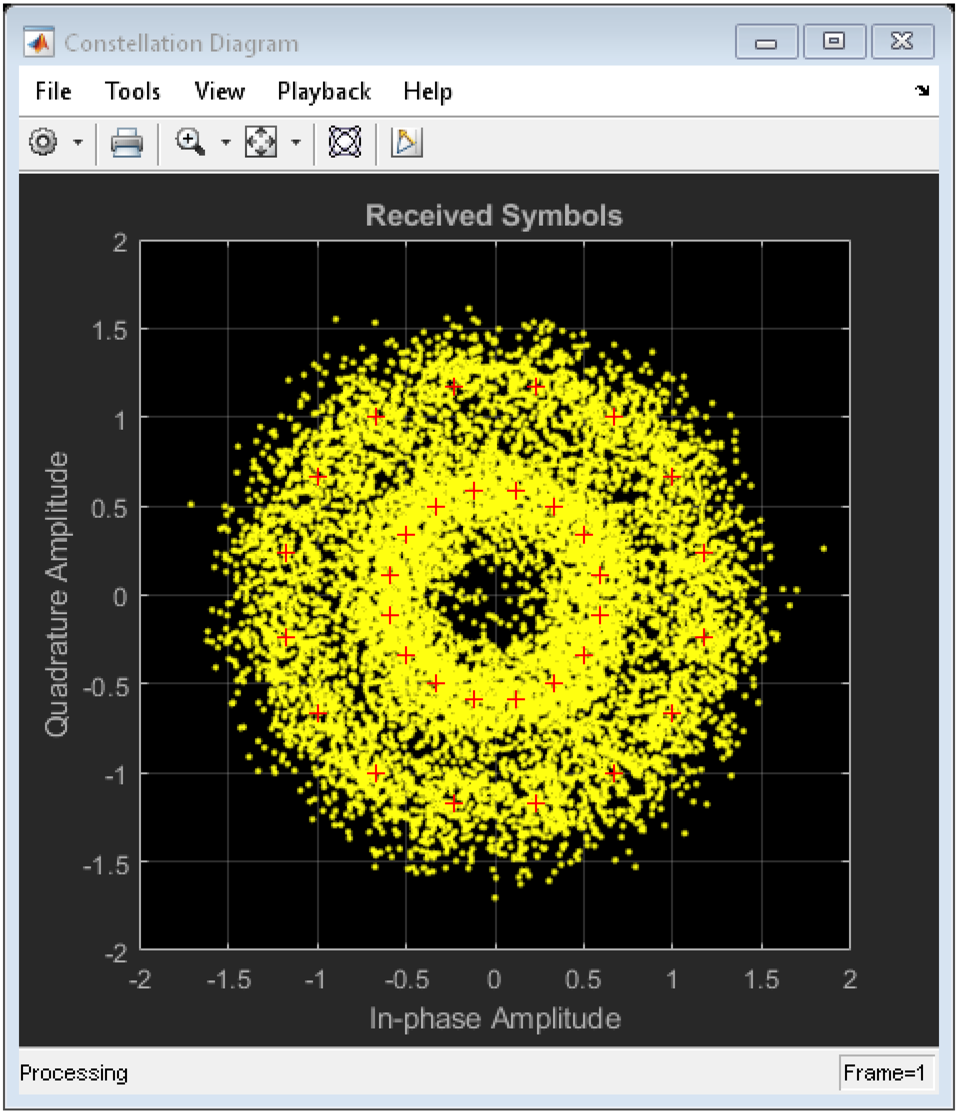
```
txSig = apskmod(x,M,radii,'InputType','bit');
sigPow = var(txSig);
```

Pass the signal through a noisy channel.

```
snr = 15;
rxSig = awgn(txSig,snr,sigPow,'linear');
```

Plot the reference and received constellation symbols.

```
constDiagAPSK(rxSig)
```



Demodulate the signal and compute soft bits.

```
z = apskdemod(rxSig,M,radii,'OutputType','approxllr', ...
    'NoiseVariance',sigPow/snr);
```

Input Arguments

y — APSK modulated signal

scalar | vector | matrix

APSK modulated signal, specified as a complex scalar, vector, or matrix. Each column is treated as an independent channel.

Data Types: `double` | `single`
Complex Number Support: Yes

`M` — Constellation points per PSK ring

vector

Constellation points per PSK ring, specified as a vector. Vector elements indicate the number of constellation points in each PSK ring. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. Element values must be multiples of four and `sum(M)` must be a power of two. The modulation order is the total number of points in the signal constellation and equals the sum of the vector elements, `sum(M)`.

Example: `[4 12 16]` specifies a three PSK ring constellation with a modulation order of `sum(M)` = 32.

Data Types: `double`

`radii` — PSK ring radii

vector

PSK ring radii, specified as a vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. The elements must be positive and arranged in increasing order.

Example: `[0.5 1 2]` defines constellation PSK ring radii. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

Data Types: `double`

`phaseoffset` — PSK ring phase offsets

`[pi/M(1) pi/M(2) … pi/M(end)]` (default) | scalar | vector

Phase offset of each PSK ring in radians, specified as a scalar or vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. The `phaseoffset` can be a scalar only if all the elements of `M` are the same value.

Example: `[pi/4 pi/12 pi/16]` defines three constellation PSK ring phase offsets. The inner ring has a phase offset of pi/4, the second ring has a phase offset of pi/12, and the outer ring has a phase offset of pi/16.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = apskdemod(x,M,radii,'OutputType','bit','OutputDataType','single');`

SymbolMapping — Symbol mapping
`'gray'` | `'contourwise-gray'` | integer vector

Symbol mapping, specified as the comma-separated pair consisting of `'SymbolMapping'` and one of the following:

- `'contourwise-gray'` — Uses Gray mapping along the contour in phase dimension.
- `'gray'` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all the values for `M` must be equal and all the values for `phaseoffset` must be equal. For a description of the Gray mapping used, see [2].
- integer vector — Use custom symbol mapping. Vector must consist of `sum(M)` unique elements with values from 0 to (`sum(M)-1`). The first element corresponds to the constellation point in the first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

The default symbol mapping depends on `M` and `phaseOffset`. When all the elements of `M` and `phaseOffset` are equal, the default is `'gray'`. For all other cases, the default is `'contourwise-gray'`.

Data Types: `double` | `char` | `string`

OutputType — Output type
`'integer'` (default) | `'bit'` | `'llr'` | `'approxllr'`

Output type, specified as the comma-separated pair consisting of `'OutputType'` and `'integer'`, `'bit'`, `'llr'`, or `'approxllr'`. For a description of the returned output, see `z`.

Data Types: `char` | `string`

`OutputDataType` — Output data type

`'double'` (default) | `...`

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and one of the indicated data types. Acceptable values for `'OutputDataType'` depend on the `'OutputType'` value.

`'OutputType'` Value	Acceptable `'OutputDataType'` Values
`'integer'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, or `'uint32'`
`'bit'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'logical'`

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'integer'` or `'bit'`.

Data Types: `char` | `string`

`NoiseVariance` — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of `'NoiseVariance'` and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

Tip When the output type is set to `'llr'`, an exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are a very small values

When the output returns any of these values, try setting output type to `'approxllr'` instead. The approximate LLR algorithm does not compute exponentials.

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'llr'` or `'approxllr'`.

Data Types: `double`

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

z — Demodulated signal

scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of `z` depend on the specified `'OutputType'` value.

`'OutputType' Value`	Return Value of apskdemod	Dimensions of z
`'integer'`	Demodulated integer values from 0 to (sum(`M`) – 1)	`z` has the same dimensions as input `y`.
`'bit'`	Demodulated bits	The number of rows in `z` is $\log_2$(sum(`M`)) times the number of rows in `y`. Each demodulated symbol is mapped to a group of $\log_2$(sum(`M`)) elements in a column,
`'llr'`	Log-likelihood ratio value for each bit	

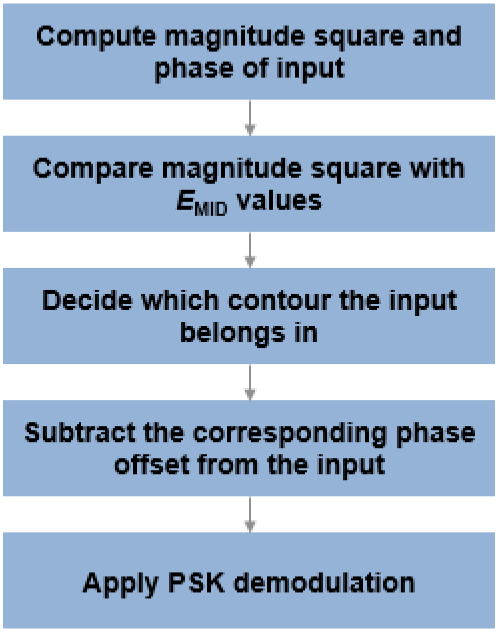
`'OutputType' Value`	Return Value of `apskdemod`	Dimensions of `z`
`'approxllr'`	Approximate log-likelihood ratio value for each bit	where the first element represents the MSB and the last element represents the LSB.

Algorithms

APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding as described in [1].

Calculate the radial partitions of the constellation by taking the mean of consecutive elements of radii. Square the radial partition values and store them in a variable E_{MID}.


Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide which contour the input belongs in
Subtract the corresponding phase offset from the input
Apply PSK demodulation

APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio algorithms are available: exact LLR and approximate LLR. Exact LLR provides the greatest accuracy but is slower, while approximate LLR is less accurate but faster.

For a description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

References

[1] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskmod` | `dvbsapskdemod` | `genqamdemod` | `mil188qamdemod` | `pskdemod` | `qamdemod`

System Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator` | `comm.RectangularQAMDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

apskmod

Amplitude phase shift keying (APSK) modulation

Syntax

```
y = apskmod(x,M,radii)
y = apskmod(x,M,radii,phaseoffset)
y = apskmod( ___ ,Name,Value)
```

Description

`y = apskmod(x,M,radii)` performs APSK modulation on the input signal, `x`, based on the specified number of constellation points per PSK ring, `M`, and the radius of each PSK ring, `radii`. For a description of APSK modulation, see “Algorithms” on page 1-40.

`y = apskmod(x,M,radii,phaseoffset)` specifies an initial phase offset for each PSK ring of the APSK modulated signal.

`y = apskmod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply APSK Modulation

Modulate data using APSK with an unequal number of constellation points on each circle.

Define vectors for modulation order and PSK ring radii. Generate data for constellation points.

```
M = [4 8 20];
radii = [0.3 0.7 1.2];
```

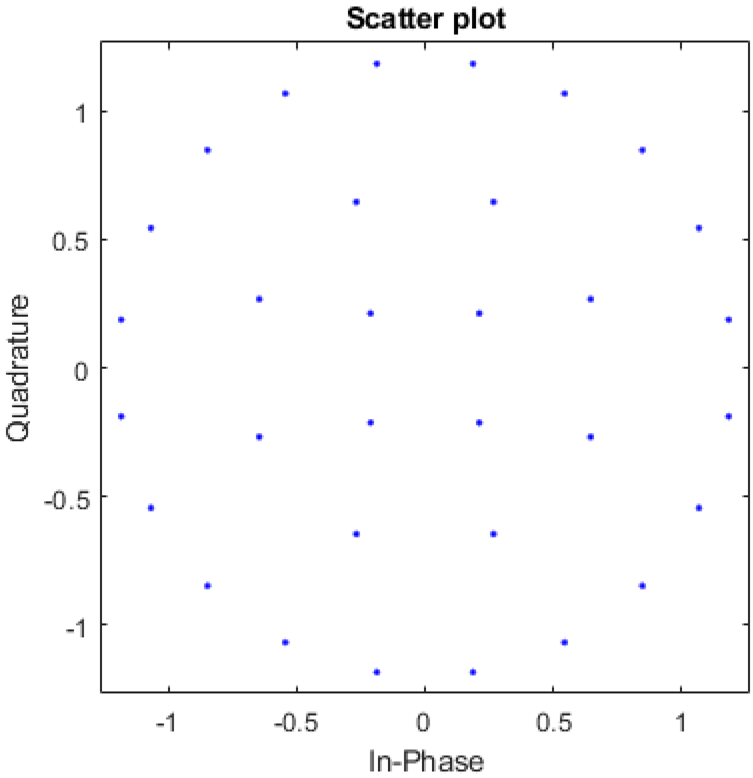
```
modOrder = sum(M);
x = 0:modOrder-1;
```

Apply APSK modulation to the data.

```
y = apskmod(x,M,radii);
```

Plot the resulting constellation using a scatter plot.

```
scatterplot(y)
```



Apply APSK Modulation with Phase Offset

Modulate a random data sequence using APSK with zero phase offset for the inner circle and pi/6 phase offset for the outer circle.

Define vectors for modulation order, PSK ring radii, and PSK ring phase offset. Generate random data.

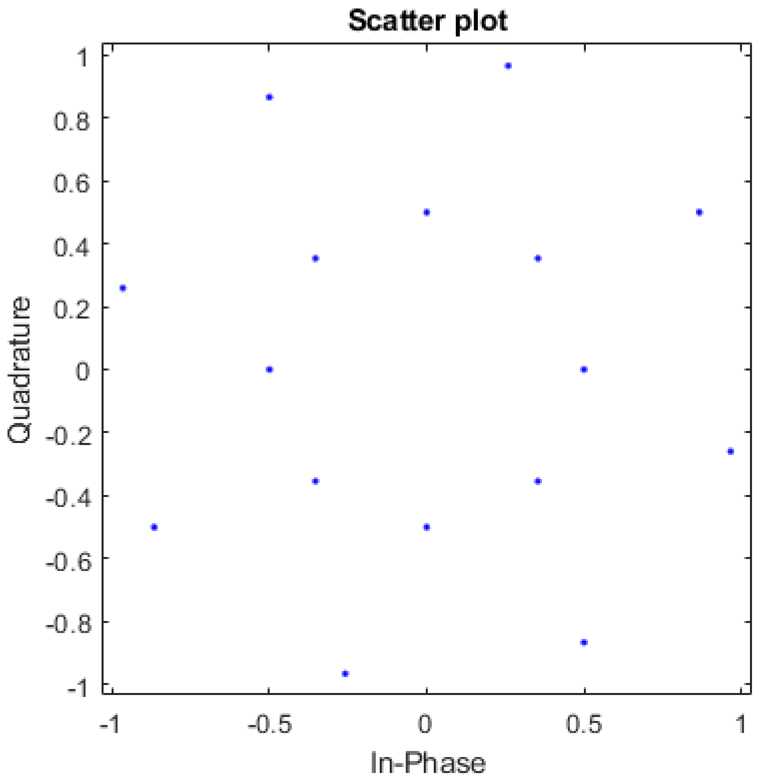
```
M = [8 8];
modOrder = sum(M);
radii = [0.5 1];
phOff = [0 pi/6];

x = randi([0 modOrder-1],100,1);
```

Apply APSK modulation to the data.

```
y = apskmod(x,M,radii,phOff);
```

Plot the resulting constellation using a scatter plot and observe the phase offset between the constellation circles.

```
scatterplot(y)
```



Apply APSK Modulation Modifying Symbol Ordering

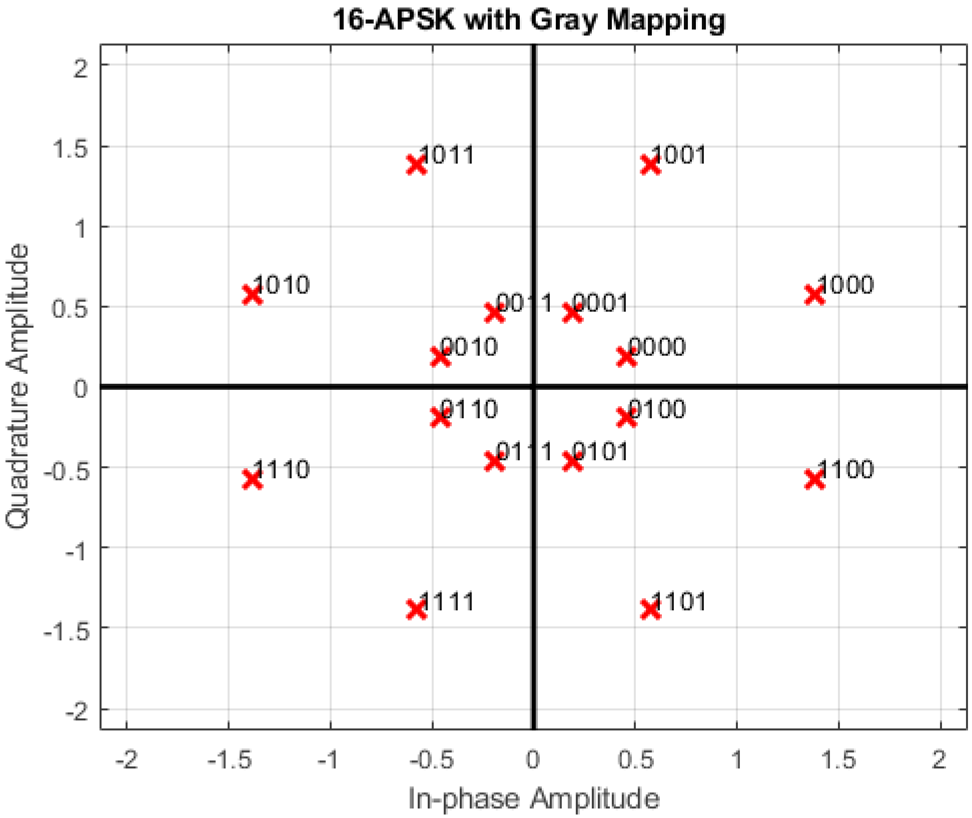
Plot APSK constellations for gray and custom symbol mappings.

Define vectors for modulation order and PSK ring radii. Generate bit data for constellation points.

```
M = [8 8];
modOrder = sum(M);
radii = [0.5 1.5];
x = 0:modOrder-1;
xBit = de2bi(x);
```

Apply APSK modulation to the data using the default phase offset. Since element values for `M` are equal and element values for phase offset are equal, the symbol mapping defaults to `'gray'`. Binary input is used to highlight the Gray nature of the constellation mapping. Plot the constellation.

```
y = apskmod(xBit,M,radii,'PlotConstellation',true,'InputType','bit');
```

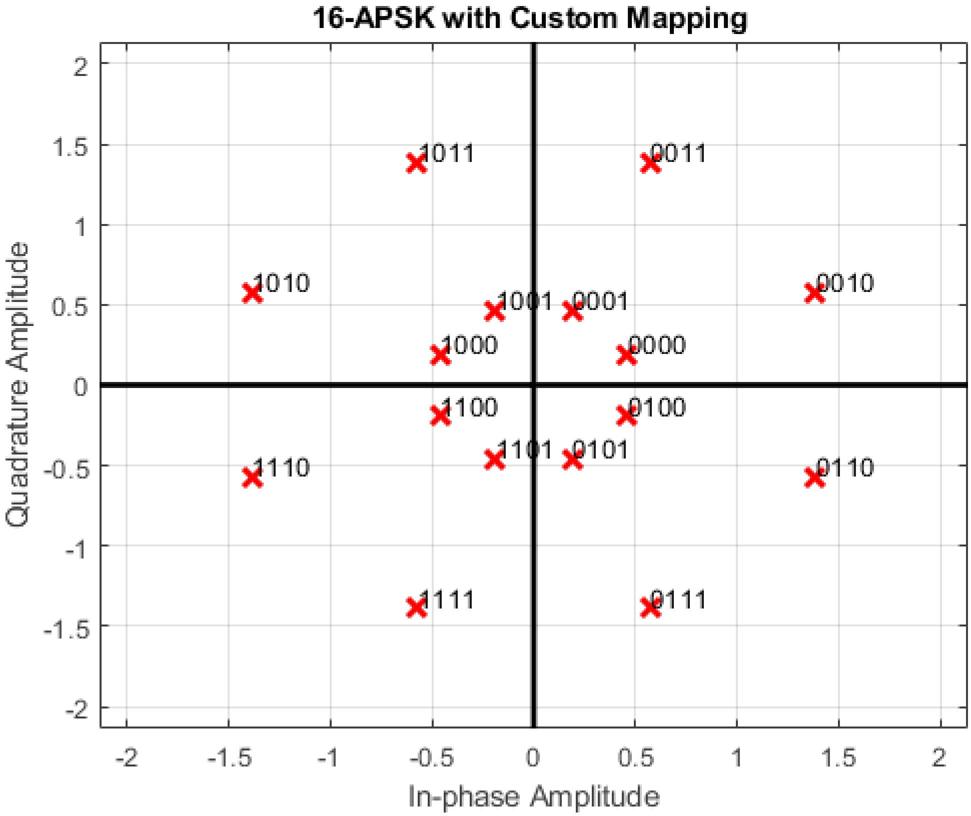


Create a custom symbol mapping vector. This custom mapping happens to be another Gray mapping.

```
cmap = [0;1;9;8;12;13;5;4;2;3;11;10;14;15;7;6];
```

Apply APSK modulation with a custom symbol mapping. Plot the constellation. Binary input is used to highlight that the custom mapping defines different Gray symbol mapping.

```
z = apskmod(xBit,M,radii,'SymbolMapping',cmap,'PlotConstellation',true,'InputType','bi
```



Apply APSK Modulation to Input Bits

Modulate a random bit sequence using APSK and output data type `single`. Pass the signal through a noisy channel and display the constellation diagram.

Define vectors for modulation order and PSK ring radii. Generate random binary data.

```
M = [8 12 20 24];
radii = [0.8 1.2 2 2.5];
bitsPerSym = log2(sum(M));

x = randi([0 1],2000*bitsPerSym,1);
```

Apply APSK modulation to the data and use a name-value pair to output as data type `single`.

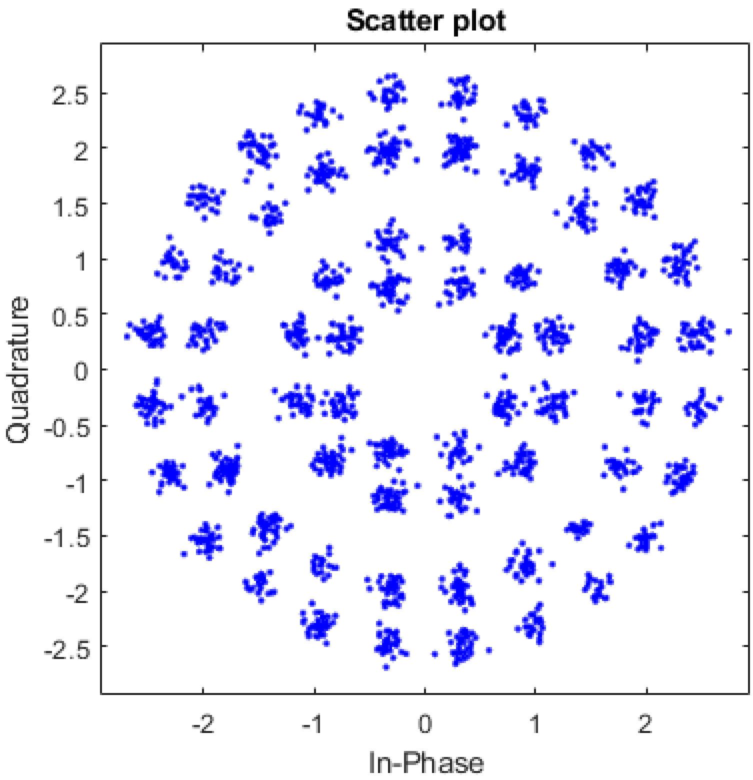
```
y = apskmod(x,M,radii,'InputType','bit','OutputDataType','single');
```

Pass through an AWGN channel with a 25 dB SNR.

```
yrec = awgn(y,25,'measured');
```

Plot the received constellation as a scatter plot.

```
scatterplot(yrec)
```



Input Arguments

x — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers that range from 0 to (`sum(M)-1`).

Note To process input signal as binary elements, set the `'InputType'` name-value pair to `'bit'`. For binary inputs, the number of rows must be an integer multiple of

$\log_2(\mathrm{sum}(M))$. Groups of $\log_2(\mathrm{sum}(M))$ bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`M` — Constellation points per PSK ring

scalar | vector

Constellation points per PSK ring, specified as a scalar or vector. A scalar specifies one PSK ring. Vector elements indicate the number of constellation points in each PSK ring. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. Element values must be multiples of four and `sum(M)` must be a power of two. The modulation order is the total number of points in the signal constellation and equals the sum of the vector elements, `sum(M)`.

Example: `[4 12 16]` specifies a three PSK ring constellation with a modulation order of `sum(M)` = 32.

Data Types: `double`

`radii` — PSK ring radii

vector

PSK ring radii, specified as a vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. The elements must be positive and arranged in increasing order.

Example: `[0.5 1 2]` defines constellation PSK ring radii. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

Data Types: `double`

`phaseoffset` — PSK ring phase offsets

`[pi/M(1) pi/M(2) … pi/M(end)]` (default) | scalar | vector

Phase offset of each PSK ring in radians, specified as a scalar or vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. The `phaseoffset` can be a scalar only if all the elements of `M` are the same value.

Example: `[pi/4 pi/12 pi/16]` defines three constellation PSK ring phase offsets. The inner ring has a phase offset of pi/4, the second ring has a phase offset of pi/12, and the outer ring has a phase offset of pi/16.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = apskmod(x,M,radii,'InputType','bit','OutputDataType','single');`

SymbolMapping — Symbol mapping

`'gray'` | `'contourwise-gray'` | integer vector

Symbol mapping, specified as the comma-separated pair consisting of `'SymbolMapping'` and one of the following:

- `'contourwise-gray'` — Uses Gray mapping along the contour in the phase dimension for each PSK ring.
- `'gray'` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all the values for `M` must be equal and all the values for `phaseoffset` must be equal. For a description of the Gray mapping used, see [2].
- integer vector — Use custom symbol mapping. Vector must consist of `sum(M)` unique elements with values from 0 to (`sum(M)-1`). The first element corresponds to the constellation point in the first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

The default symbol mapping depends on `M` and `phaseOffset`. When all the elements of `M` and `phaseOffset` are equal, the default is `'gray'`. For all other cases, the default is `'contourwise-gray'`.

Data Types: `double` | `char` | `string`

InputType — Input type

`'integer'` (default) | `'bit'`

Input type, specified as the comma-separated pair consisting of 'InputType' and either 'integer' or 'bit'. To use 'integer', the input signal must consist of integers from 0 to (sum(M)-1). To use 'bit', the input signal must contain binary values and the number of rows must be an integer multiple of $\log_2$(sum(M)).

Data Types: char | string

OutputDataType — Output data type
'double' (default) | 'single'

Output data type, specified as the comma-separated pair consisting of 'OutputDataType' and either 'double' or 'single'.

Data Types: char | string

PlotConstellation — Plot reference constellation
false (default) | true

Plot reference constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the reference constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

y — APSK modulated signal
scalar | vector | matrix

APSK modulated signal, returned as a complex scalar, vector, or matrix. The dimensions of y depend on the specified 'InputType' value.

InputType	Dimensions of y
'integer'	y has the same dimensions as input x.
'bit'	The number of rows in y equals the number of rows in x divided by $\log_2$(sum(M)).

Algorithms

The function implements a pure APSK constellation. A pure M-APSK constellation is composed of N_C concentric rings or contours, each with uniformly spaced PSK points. The M-APSK constellation set is:

$$\chi = \begin{cases} R_1 \exp\left(j\left(\frac{2\pi}{M_1} i + \theta_1\right)\right), & i = 0,\ldots,M_1 - 1, \\ R_2 \exp\left(j\left(\frac{2\pi}{M_2} i + \theta_2\right)\right), & i = 0,\ldots,M_2 - 1, \\ \vdots & \vdots \\ R_{\mathrm{Nc}} \exp\left(j\left(\frac{2\pi}{M_{\mathrm{Nc}}} i + \theta_{\mathrm{Nc}}\right)\right), & i = 0,\ldots,M_{\mathrm{Nc}} - 1, \end{cases}$$

where

- M_l is the number of constellation points in the lth ring.
- R_l is the radius of the lth ring.
- θ_l is the phase offset of the lth ring.
- j = √(-1).
- N_C is the number of concentric rings.
- The modulation order is equal to sum(M_l) for l = 1, 2, ... , N_C.

References

[1] Corazza, Giovanni E. *Digital Satellite Communications*. New York: Springer Science Business Media, LLC, 2007.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskmod` | `mil188qammod` | `pskmod` | `qammod`

System Objects

`comm.GeneralQAMModulator` | `comm.PSKModulator` | `comm.RectangularQAMModulator`

Introduced in R2018a

arithdeco

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

Decode Sequence Using Arithmetic Code

Set the `counts` vector so that a one occurs 99% of the time.

```
counts = [99 1];
```

Set the sequence length to 1000. Generate a random sequence.

```
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]);
```

Arithmetically encode the random sequence then, decode the encoded sequence.

```
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq));
```

Verify that the decoded sequence matches the original sequence.

```
isequal(seq,dseq)

ans = logical
   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`

Topics

"Arithmetic Coding"

Introduced before R2006a

arithenco

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

Encode Data Sequence with Arithmetic Code

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 fewer than 1000 elements. The actual number of elements in encoded sequence varies depending on the particular random sequence.

Set `counts` so that a one occurs 99% of the time.

```
counts = [99 1];
```

Generate a random data sequence of length 1000.

```
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]);
```

Encode the sequence and display the encoded length.

```
code = arithenco(seq,counts);
s = size(code)
```

```
s = 1×2

     1    57
```

The length of the encoded sequence is only 5.7% of the length of the original sequence.

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`

Topics

“Arithmetic Coding”

Introduced before R2006a

BER Analyzer

Analyze bit error rate (BER) performance of communications systems

Description

The **BER Analyzer** app calculates BER as a function of the energy per bit to noise power spectral density ratio (E_b/N_0). Using this app, you can:

- Plot theoretical BER vs. E_b/N_0 estimates and upper bounds.
- Plot BER vs. E_b/N_0 using the semianalytic technique. The semianalytic technique estimates BER performance by using a combination of simulation and analysis. Use this technique when the system error rate is small, for example, < 10^{-6}.
- Estimate BER performance by using MATLAB® functions or Simulink® models.

Open the BER Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `bertool`.

Examples

Theoretical Plot

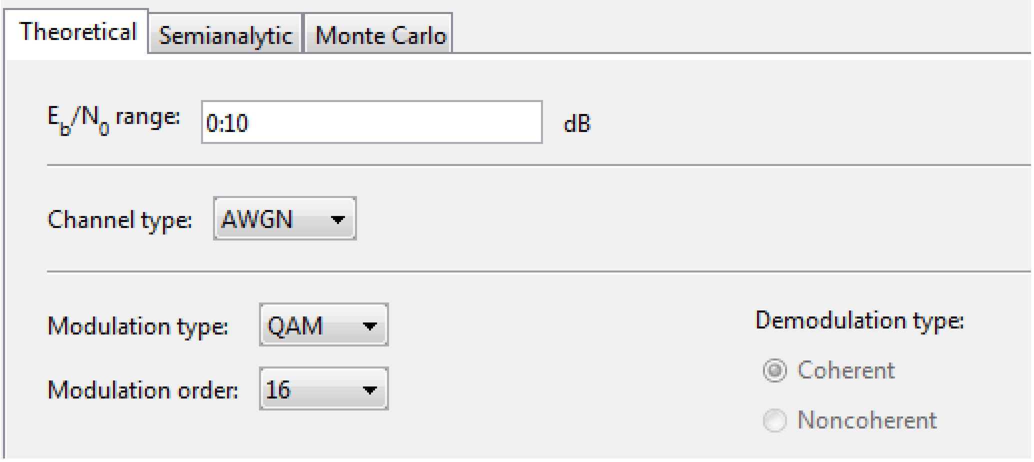
Generate a theoretical estimate of BER performance for a 16-QAM link in AWGN.

Open the **BER Analysis** app.

```
bertool
```

Specify the **E_b/N_0 range** as `0:10`.

Set **Modulation type** to `QAM` and **Modulation order** to `16`.



E_b/N_0 range: 0:10 dB

Channel type: AWGN

Modulation type: QAM

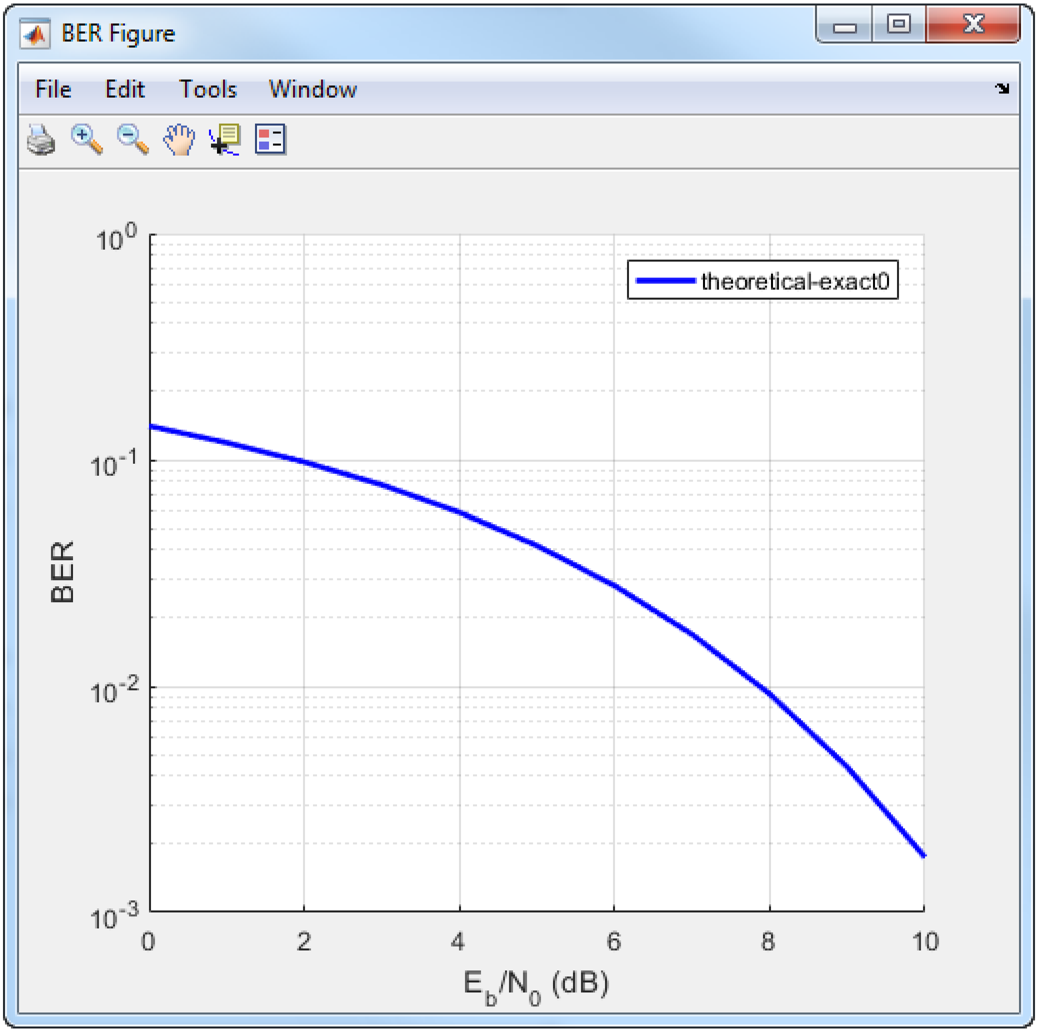
Modulation order: 16

Demodulation type:

● Coherent

○ Noncoherent

Plot the BER curve by clicking **Plot**.



Semianalytic Plot

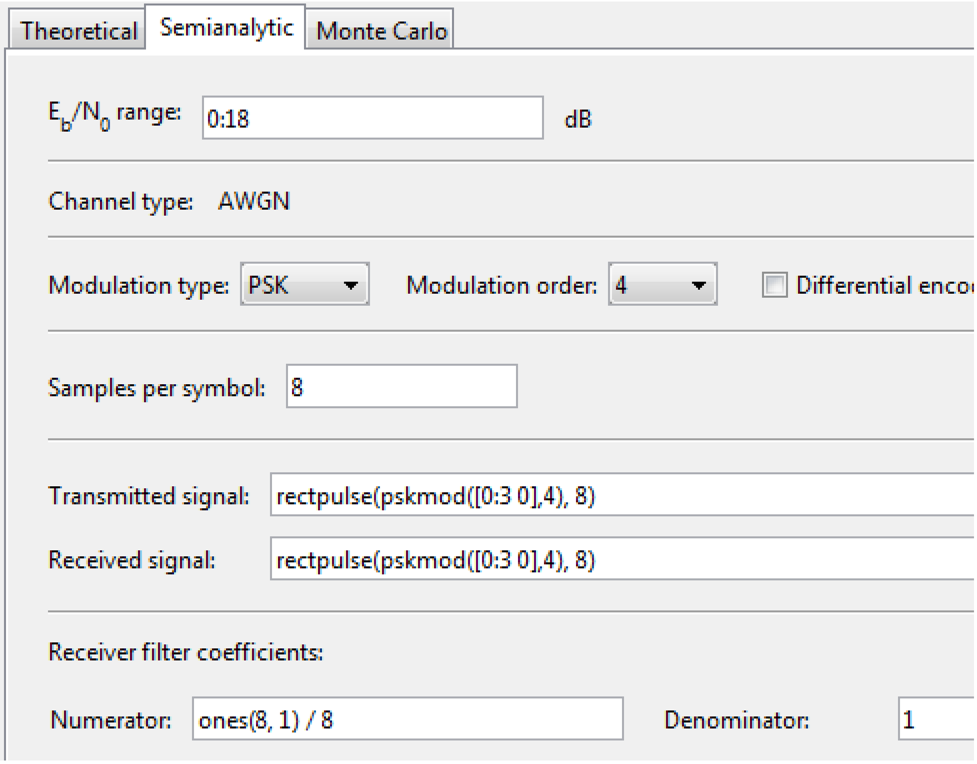
Use the semianalytic technique to plot the BER for a QPSK link having rectangular pulses.

Open the **BER Analysis** app.

```
bertool
```

On the **Semianalytic** tab, set these parameters:

- Set the **Modulation order** to `4`.
- Set the **Samples per symbol** parameter to `8`.
- Set the **Transmitted signal** and **Received signal** parameters to `rectpulse(pskmod([0:3 0],4),8)`. To use the semianalytic technique, the number of symbols must exceed M^L, where M is the modulation order and L is the impulse response length. The impulse response is 1, so a minimum of five symbols is required.
- Specify the **Numerator** as `ones(8,1)/8`. These coefficients specify an ideal integrator having eight samples per symbol.



E_b/N_0 range: 0:18 dB

Channel type: AWGN

Modulation type: PSK

Modulation order: 4

□ Differential enco

Samples per symbol: 8

Transmitted signal: rectpulse(pskmod([0:3 0],4), 8)

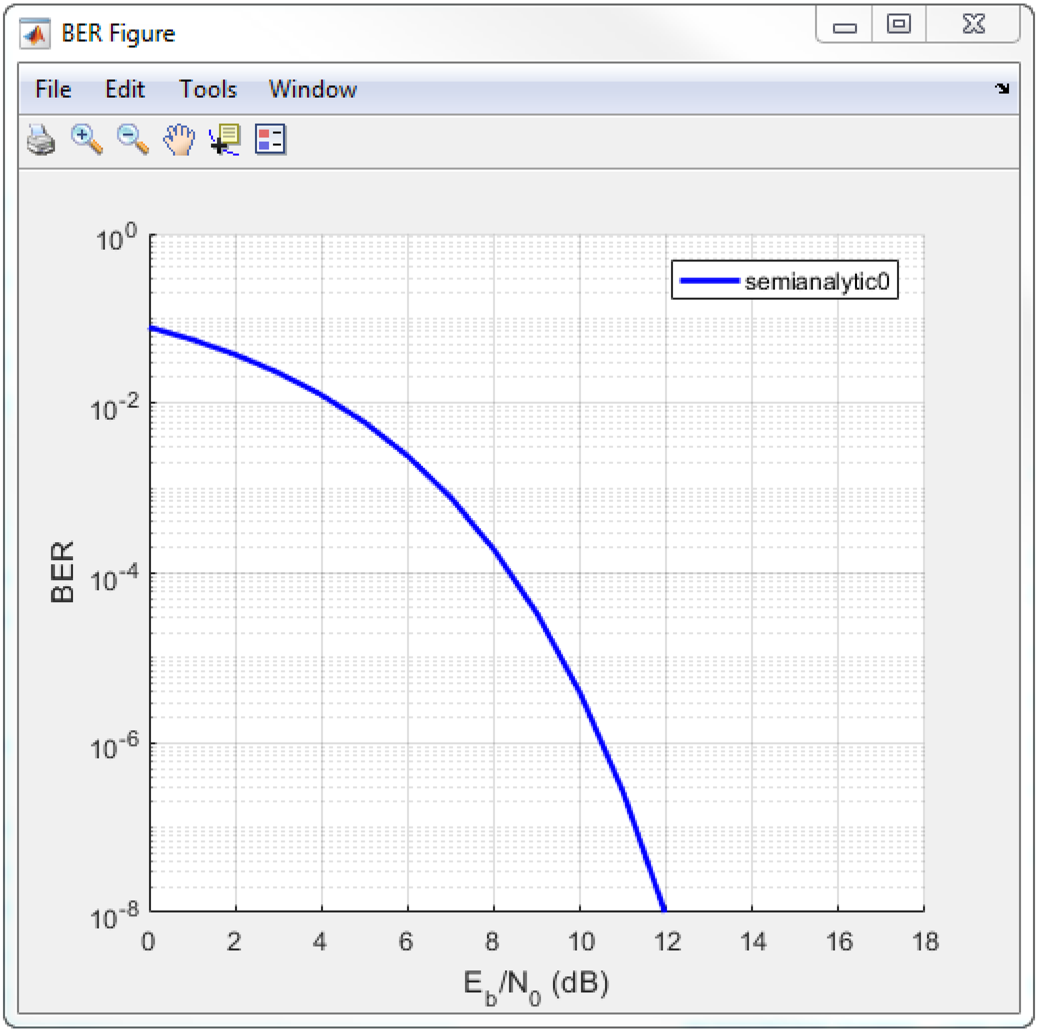
Received signal: rectpulse(pskmod([0:3 0],4), 8)

Receiver filter coefficients:

Numerator: ones(8, 1) / 8

Denominator: 1

Plot the BER vs. E_b/N_0 curve by clicking **Plot**.



Monte Carlo Simulation

Simulate BER using a custom MATLAB function.

Open the **BER Analysis** app.

```
bertool
```

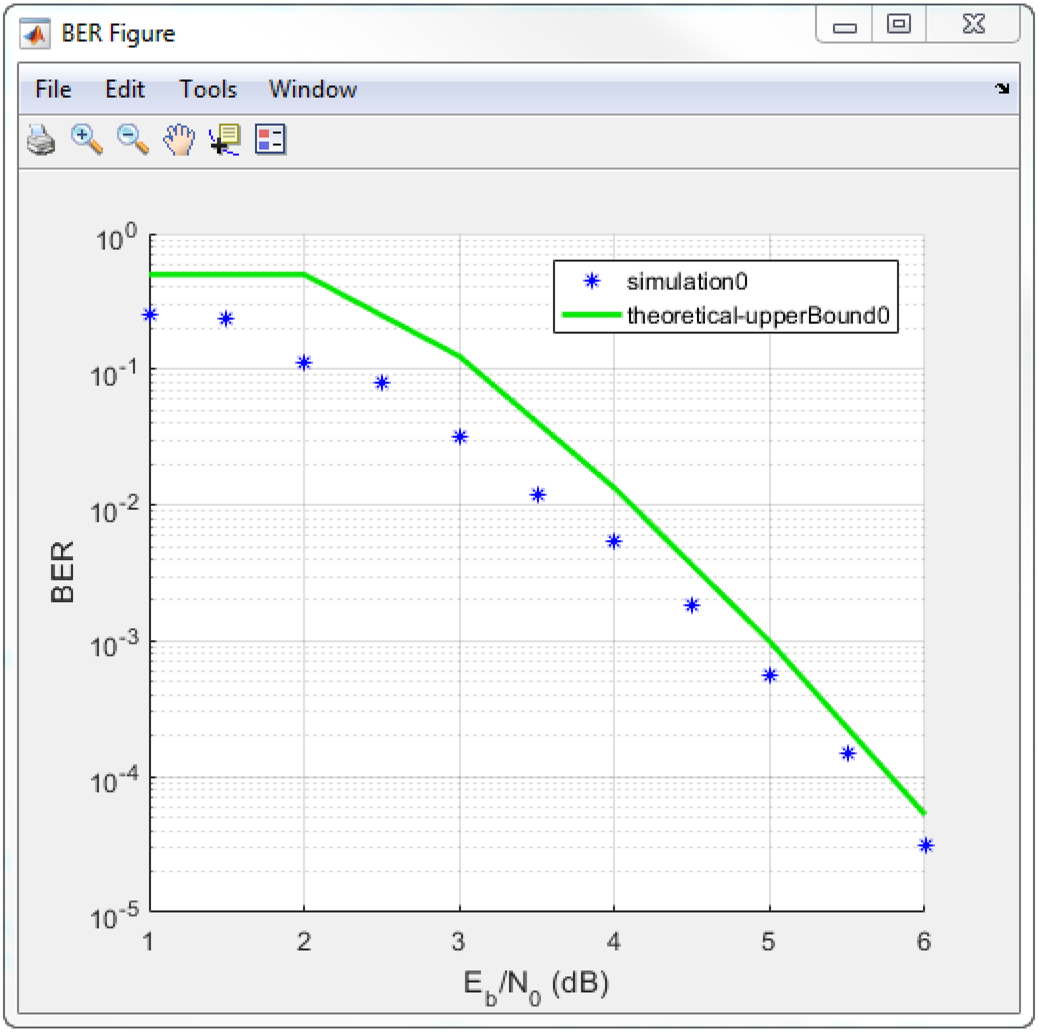
On the **Monte Carlo** tab, specify E_b/N_0 range as `1:.5:6`.

To plot estimated BER values, run the simulation by clicking **Run**.

On the **Theoretical** tab, specify E_b/N_0 range as `1:6` and set **Modulation order** to `4`.

Enable convolutional coding by selecting the **Convolutional** check box.

Plot the upper bound of the BER curve by clicking **Plot**.



Parameters

Theoretical

E_b/N_0 range — Range of E_b/N_0 values over which the BER is evaluated
`0:18` (default) | vector

Specifies the range of E_b/N_0 values, in dB, over which the BER is evaluated. The values in the range vector must be real.

Example: 5:10

Channel type — Type of channel over which the BER is evaluated
AWGN (default) | Rayleigh | Rician

Specifies the type of channel over which the BER is evaluated. The Rayleigh and Rician options correspond to flat fading channels.

Modulation type — Modulation type of the communication link
PSK (default) | DPSK | OQPSK | PAM | QAM | FSK | MSK | CPFSK

Specifies the modulation type of the communication link.

Modulation order — Modulation order of the communication link
2 (default) | 4 | 8 | 16 | 32 | 64

Specifies the modulation order of the communication link.

Differential encoding — Differential encoding of the input data
Off (default) | On

Specifies if the input data sequence is differentially encoded.

Demodulation type — Demodulation type
Coherent | Noncoherent

Specifies if Coherent or Noncoherent demodulation is used. This parameter is available only when the **Modulation type** is FSK or MSK.

Channel coding — Channel coding used in estimating the BER
None (default) | Convolutional | Block

Specifies the type of channel coding used to estimate the theoretical BER.

Synchronization — Synchronization error
Perfect synchronization (default) | Normalized timing error | RMS phase noise level

Specifies the synchronization error in the demodulation process. This parameter is available only when the **Modulation type** is PSK and the **Modulation order** is 2.

- When **Synchronization** is `Normalized timing error`, specify the normalized error as a real number from `0` to `0.5`.
- When **Synchronization** is `RMS phase noise level`, specify the RMS phase noise as a nonnegative real number.

`Decision method` — Decoding decision method

`Hard` (default) | `Soft`

Specify the method used to decode the received data. This parameter is available when either of these conditions exist:

- **Channel coding** is set to **Convolutional**
- **Channel coding** is set to **Block** and **Coding Type** is `General`

`Trellis` — Convolutional code trellis

`poly2trellis(7,[171 133])` (default) | structure

Specify the convolutional code trellis as a structure variable. You can generate this structure by using the `poly2trellis` function. The parameter is available only when the **Channel coding** parameter is `Convolutional`.

`Coding type` — Specify block coding type

`General` (default) | `Hamming` | `Golay` | `Reed-Solomon`

Specify the block code used in the BER evaluation.

`N` — Codeword length

positive integer

Specify the codeword length as a positive integer.

`K` — Message length

positive integer

Specify the message length as a positive integer such that **K** is less than **N**.

d_{min} — Minimum code distance

positive integer

Specify the minimum distance of the (N,K) block code as a positive integer. This parameter is available when the **Coding type** is `General`.

Semianalytic

`Samples per symbol` — Samples per symbol

`16` (default) | positive integer

Specify the number of samples per symbol as a positive integer.

`Transmitted signal` — Transmitted sample sequence

`rectpulse(step(comm.BPSKModulator, [0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0].'), 16)` (default) | vector

Specify the transmit sequence as a real or complex column vector.

Data Types: `double`

`Received signal` — Received sample sequence

`rectpulse(step(comm.BPSKModulator, [0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0].'), 16)` (default) | vector

Specify the received sequence as a real or complex column vector.

Data Types: `double`

`Numerator` — Numerator of the receive filter coefficients

`ones(16,1)/16` (default) | scalar | vector

Specify the numerator of the receive filter coefficients as a vector.

`Denominator` — Denominator of the receive filter coefficients

`1` (default) | scalar | vector

Specify the denominator of the receive filter coefficients as a vector.

Monte Carlo

`Simulation MATLAB file or Simulink model` — File name of the BER simulation

character vector

Specify the name of the MATLAB file or Simulink model containing the simulation code.

`BER variable name` — Name of the variable containing the BER simulation data

character vector

Specify the name of the MATLAB workspace variable that contains the BER simulation data.

`Number of errors` — Number of errors measured before simulation stop
`100` (default) | positive integer

Specify the number of errors that must be measured before the simulation stops. Typically, 100 measured errors are enough to produce an accurate BER estimate.

`Number of bits` — Number of bits processed before simulation stop
`1e8` (default) | positive integer

Specify the number of bits that must be processed before the simulation stops. This parameter is used to prevent the simulation from running too long.

Note The Monte Carlo simulation stops when either the number of errors or number of bits threshold is reached.

See Also

Functions
`berawgn` | `bercoding` | `berfading` | `berfit`

Topics

"Bit Error Rate (BER)"

Introduced before R2006a

Eye Diagram Analyzer

(To be removed) Visualize and measure effects of impairments

Note `commscope.eyediagram` will be removed in a future release. Use `comm.EyeDiagram` instead.

Description

The **Eye Diagram Analyzer** app displays and measures the effects of various impairments. Using this app, you can:

- Visualize eye diagrams.
- Measure these quantities:
  - Horizontal and vertical eye openings
  - Random, deterministic, total, RMS, and peak-to-peak jitter
  - Rise and fall times
  - Signal-to-noise ratio
- Import and compare measurement results for eye diagrams of multiple signals.

Open the Eye Diagram Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `eyescope`.

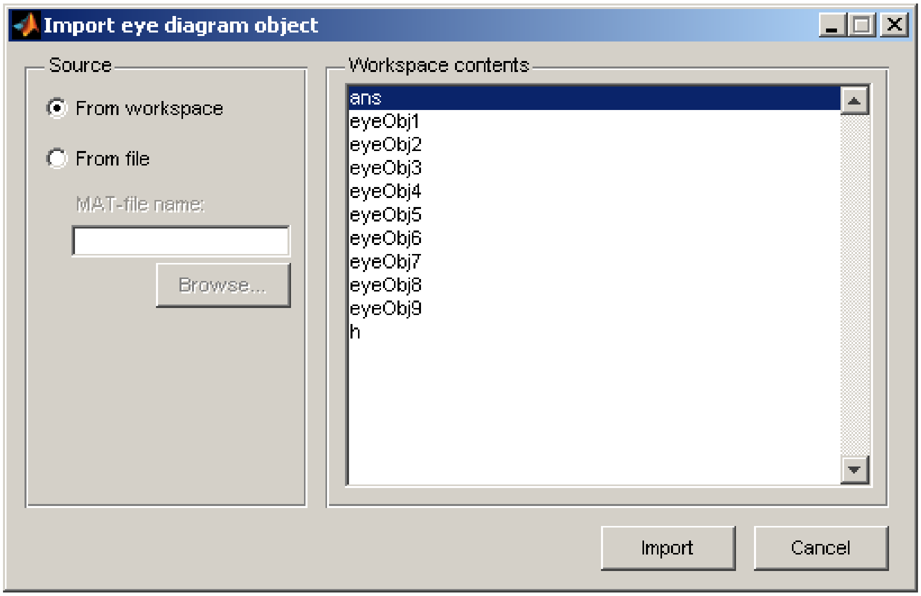
Examples

Import Eye Diagrams and Compare Measurement Results

MATLAB software includes a set of data containing nine eye diagram objects, which you can import into EyeScope. While EyeScope can import eye diagram objects from either the workspace or a MAT-file, this introduction covers importing from the workspace. EyeScope reconstructs the variable names it imports to reflect the origin of the eye diagram object.

1 Type `load commeye_EyeMeasureDemoData` at the MATLAB command line to load nine eye diagram objects into the MATLAB workspace.
2 Type `eyescope` at the MATLAB command line to start the EyeScope tool.
3 In the EyeScope window, select **File > Import Eye Diagram Object**.

   The **Import eye diagram object** dialog box opens.



Import eye diagram object

Source

● From workspace

○ From file

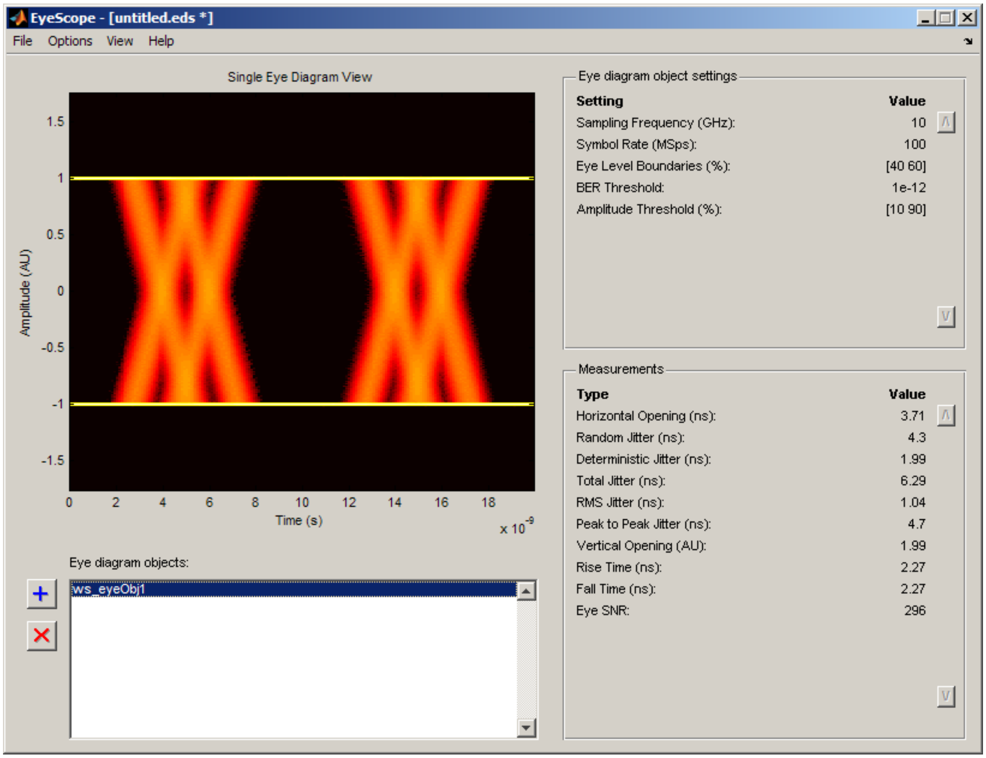
MAT-file name:

Workspace contents

ans
eyeObj1
eyeObj2
eyeObj3
eyeObj4
eyeObj5
eyeObj6
eyeObj7
eyeObj8
eyeObj9
h

   In this window, the **Workspace contents** panel displays all eye diagram objects available in the source location.
4 Select **eyeObj1** and click **Import**. EyeScope imports the object, displaying an image in the object plot and listing the file name in the **Eye diagram objects** list.

Note Object names associated with eye diagram objects that you import from the work space begin with the prefix *ws*.



Eye diagram object settings

Setting	Value
Sampling Frequency (GHz):	10
Symbol Rate (MSps):	100
Eye Level Boundaries (%):	[40 60]
BER Threshold:	1e-12
Amplitude Threshold (%):	[10 90]

Measurements

Type	Value
Horizontal Opening (ns):	3.71
Random Jitter (ns):	4.3
Deterministic Jitter (ns):	1.99
Total Jitter (ns):	6.29
RMS Jitter (ns):	1.04
Peak to Peak Jitter (ns):	4.7
Vertical Opening (AU):	1.99
Rise Time (ns):	2.27
Fall Time (ns):	2.27
Eye SNR:	296

Review the image and note the default **Eye diagram object settings** and **Measurements** selections. For more information, refer to the EyeScope on page 1-481 reference page.

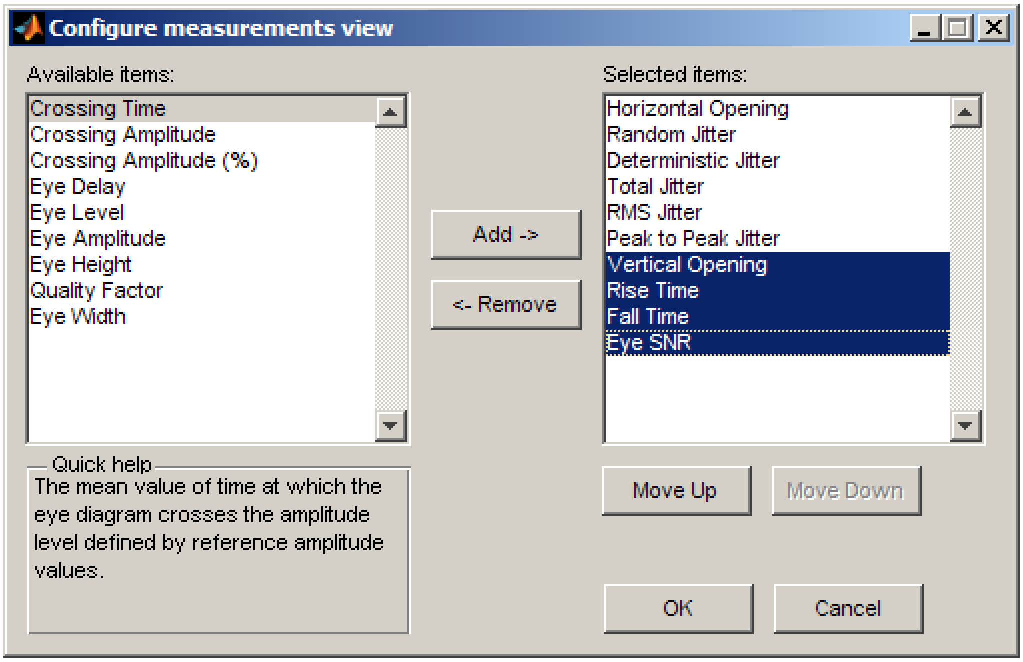
5 In the EyeScope window, click the Import + button.

6 From the Import eye diagram object window, click to select eyeObj5 then click the **Import** button.

- The EyeScope window changes, displaying a new plot and adding ws_eyeObj5 to the **Eye diagram objects** list. EyeScope displays the same settings and measurements for both eye diagram objects.
- You can switch between the eyediagram plots EyeScope displays by clicking on an object name in the Eye diagram object list.
- Next, click **ws_eyeObj1** and note the EyeScope plot and measurement values changes.

7 To change or remove measurements from the EyeScope display:

- Select **Options > Measurements View**. The **Configure measurement view** shuttle control opens.



Available items:

Crossing Time
Crossing Amplitude
Crossing Amplitude (%)
Eye Delay
Eye Level
Eye Amplitude
Eye Height
Quality Factor
Eye Width

Selected items:

Horizontal Opening
Random Jitter
Deterministic Jitter
Total Jitter
RMS Jitter
Peak to Peak Jitter
Vertical Opening
Rise Time
Fall Time
Eye SNR

Quick help

The mean value of time at which the eye diagram crosses the amplitude level defined by reference amplitude values.

- Hold down the <Ctrl> key and click to select Vertical Opening, Rise Time, Fall Time, Eye SNR. Then click **Remove**.

8 From the left side of the shuttle control, select **Crossing Time** and **Crossing Amplitude** and then click **Add**. To display EyeScope with these new settings, click **OK**. EyeScope's **Measurement** region displays Crossing Time and Crossing Amplitude at the bottom of the Measurements section.

9 Change the list order so that **Crossing Time** and **Crossing Amplitude** appear at the top of the list.

- Select **Options > Measurements View**.
- When the **Configure measurement view** shuttle control opens, hold down the <Ctrl> key and click to select **Crossing Time** and **Crossing Amplitude**.
- Click the **Move Up** button until these selections appear at the top of the list. Then, click **OK**

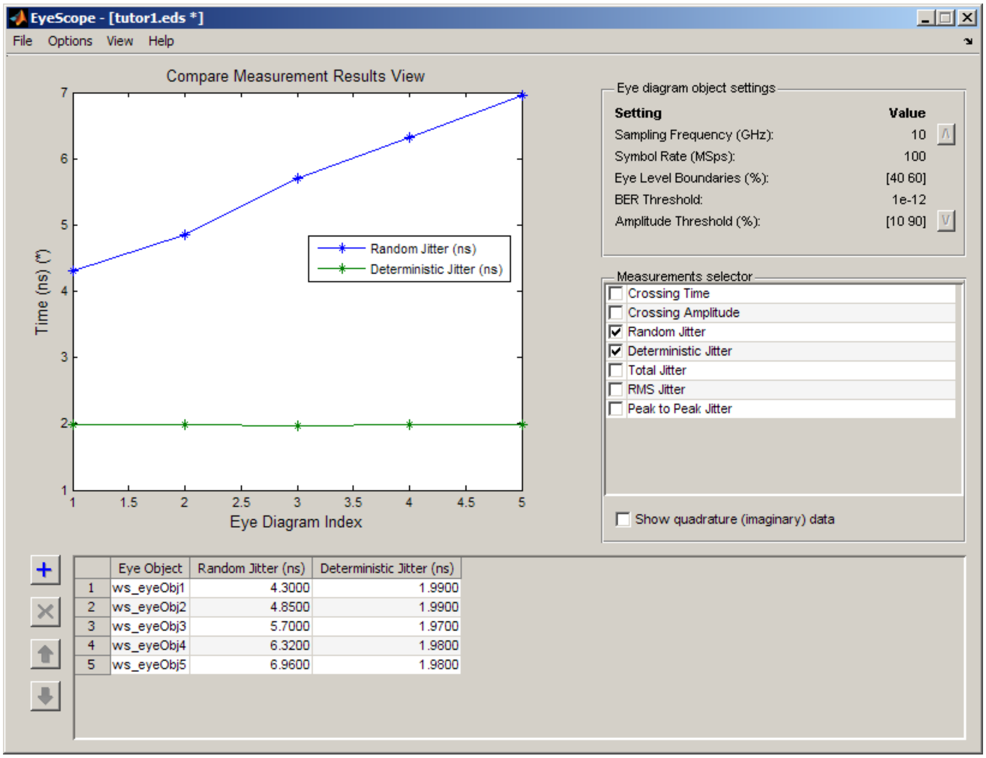
10 Select **File > Save session as** and then type a file name in the pop-up window.

11 Import **ws_eyeObj2**, **ws_eyeObj3**, and **ws_eyeObj4**. EyeScope now contains eye diagram objects 1, 5, 2, 3, and 4 in the list.

12 Select **ws_eyeObj5**, and click the delete ✕ button.

13 Click **File > Import Eye Diagram Object**, and select **ws_eyeObj5**.

14 To compare measurement results for multiple eye diagram objects, click **View > Compare Measurement Results View**.



Eye diagram object settings

Setting	Value
Sampling Frequency (GHz):	10
Symbol Rate (MSps):	100
Eye Level Boundaries (%):	[40 60]
BER Threshold:	1e-12
Amplitude Threshold (%):	[10 90]

Measurements selector

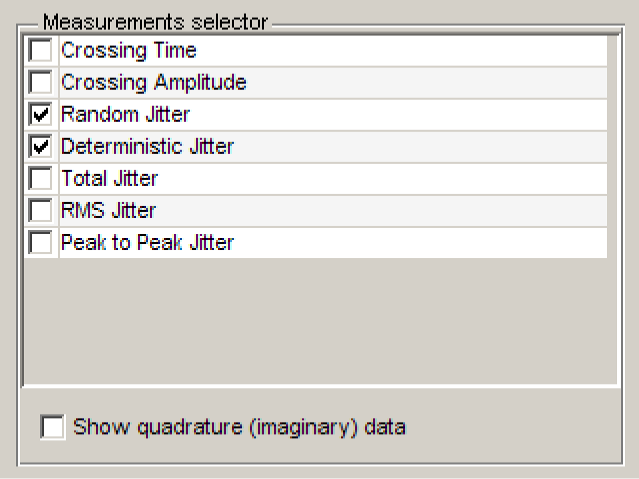
- □ Crossing Time
- □ Crossing Amplitude
- ☑ Random Jitter
- ☑ Deterministic Jitter
- □ Total Jitter
- □ RMS Jitter
- □ Peak to Peak Jitter

□ Show quadrature (imaginary) data

	Eye Object	Random Jitter (ns)	Deterministic Jitter (ns)
1	ws_eyeObj1	4.3000	1.9900
2	ws_eyeObj2	4.8500	1.9900
3	ws_eyeObj3	5.7000	1.9700
4	ws_eyeObj4	6.3200	1.9800
5	ws_eyeObj5	6.9600	1.9800

In the data set, random jitter increases from experiment 1 to experiment 5, as you can see in both the table and plot figure.

15 To include any data from the Measurements selection you chose earlier in this procedure, use the **Measurement selector**. Go to the **Measurement selector** and select **Total Jitter**. The object plot updates to display the additional measurements.



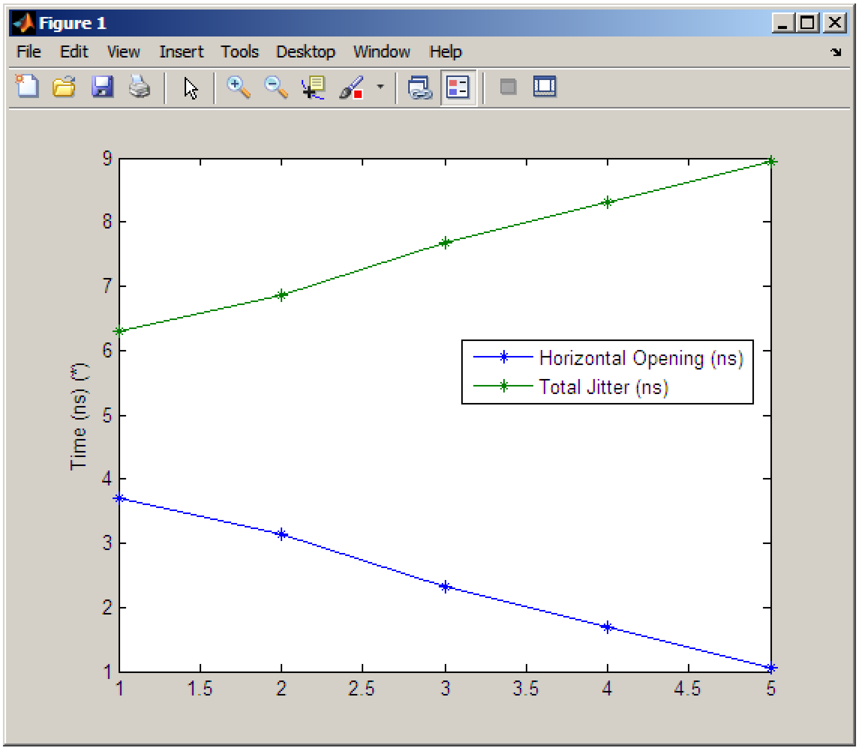
Measurements selector

☐ Crossing Time
☐ Crossing Amplitude
☑ Random Jitter
☑ Deterministic Jitter
☐ Total Jitter
☐ RMS Jitter
☐ Peak to Peak Jitter

☐ Show quadrature (imaginary) data

You can also remove measurements from the plot display. In the **Measurements selector**, select **Random Jitter** and **Deterministic Jitter**. The object plot updates, removing these two measurements.

16 To print the plot display, select **File > Print to Figure**. From **Figure** window, click the print button.



Programmatic Use

`eyescope` calls an empty scope.

`eyescope(obj)` calls the eye scope and displays object `obj`.

See Also

`comm.EyeDiagram`

Introduced in R2008b

awgn

Add white Gaussian noise to signal

Syntax

```
out = awgn(in,snr)
out = awgn(in,snr,signalpower)

out = awgn(in,snr,signalpower,randobject)
out = awgn(in,snr,signalpower,seed)
out = awgn( ___ ,powertype)
```

Description

`out = awgn(in,snr)` adds white Gaussian noise to the vector signal `in`. This syntax assumes that the power of `in` is 0 dBW.

`out = awgn(in,snr,signalpower)` accepts an input signal power value in dBW. To have the function measure the power of `in` before adding noise, specify `signalpower` as `'measured'`.

`out = awgn(in,snr,signalpower,randobject)` accepts input combinations from prior syntaxes and a random number stream object to generate normal random noise samples.

`out = awgn(in,snr,signalpower,seed)` accepts a seed value for initializing the normal random number generator, `randn`, before adding white Gaussian noise to the input signal. To generate repeatable noise samples, either reset the random stream input before calling `awgn` or use the same seed input every time you call awgn.

`out = awgn( ___ ,powertype)` specifies the signal and noise power type as `'dB'` or `'linear'`.

For the relationships between SNR and other measures of the relative power of the noise, such as E_s/N_0, and E_b/N_0, see “AWGN Channel Noise Level”.

Examples

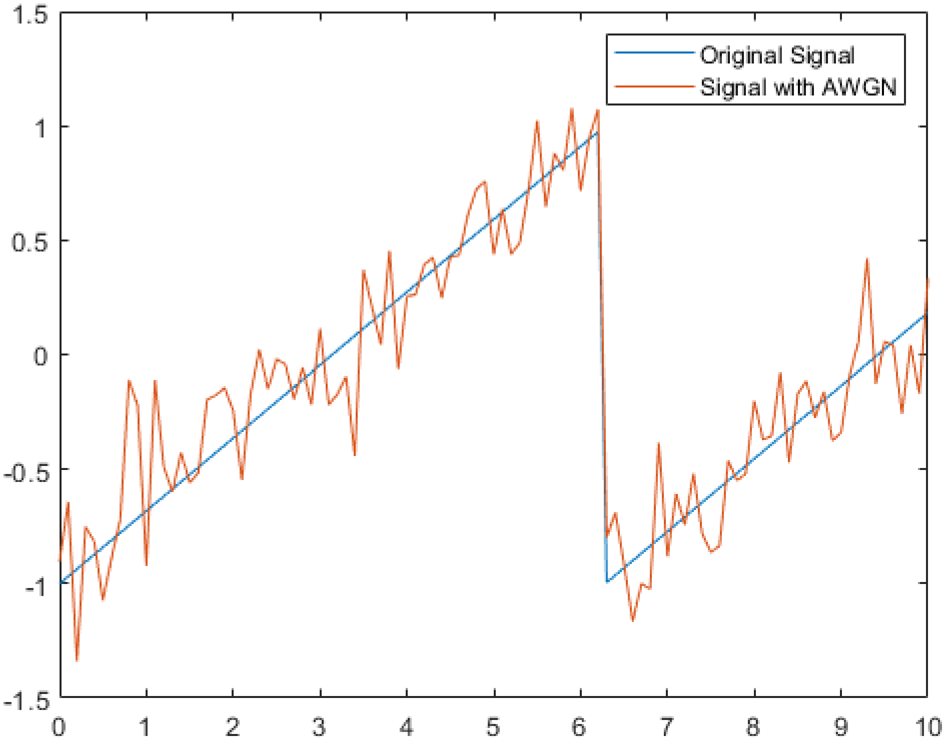
Add AWGN to Sawtooth Signal

Create a sawtooth wave.

```
t = (0:0.1:10)';
x = sawtooth(t);
```

Apply white Gaussian noise and plot the results.

```
y = awgn(x,10,'measured');
plot(t,[x y])
legend('Original Signal','Signal with AWGN')
```



General QAM Modulation in AWGN Channel

Transmit and receive data using a nonrectangular 16-ary constellation in the presence of Gaussian noise. Show the scatter plot of the noisy constellation and estimate the symbol error rate (SER) for two different signal-to-noise ratios.

Create a 16-QAM constellation based on the V.29 standard for telephone-line modems.

```
c = [-5 -5i 5 5i -3 -3-3i -3i 3-3i 3 3+3i 3i -3+3i -1 -1i 1 1i];
M = length(c);
```

Generate random symbols.

```
data = randi([0 M-1],2000,1);
```

Modulate the data by using the `genqammod` function. General QAM modulation is necessary because the custom constellation is not rectangular.

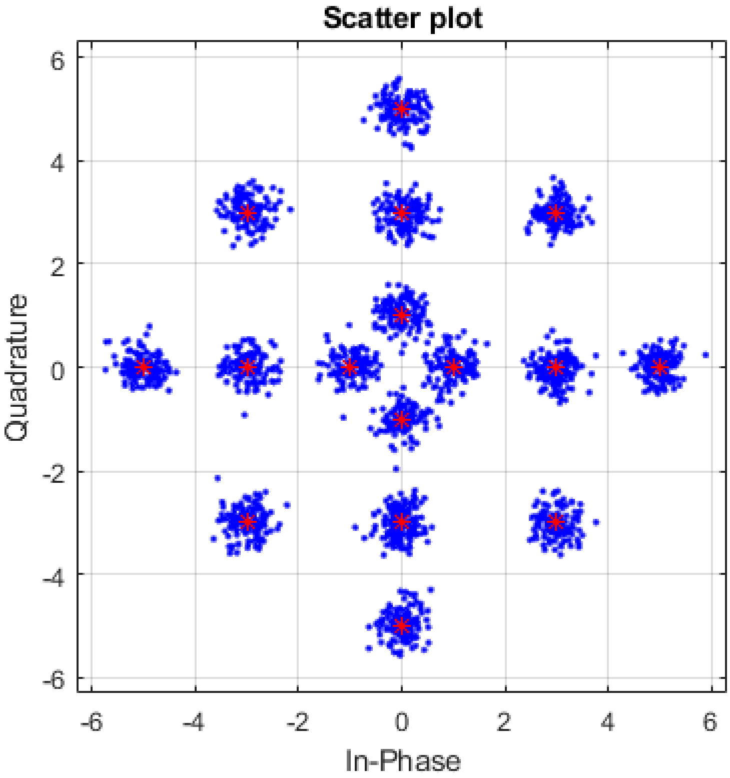
```
modData = genqammod(data,c);
```

Pass the signal through an AWGN channel having a 20 dB signal-to-noise ratio (SNR).

```
rxSig = awgn(modData,20,'measured');
```

Display a scatter plot of the received signal and the reference constellation, `c`.

```
h = scatterplot(rxSig);
hold on
scatterplot(c,[],[],'r*',h)
grid
hold off
```



Demodulate the received signal by using the genqamdemod function. Determine the number of symbol errors and the symbol error ratio.

```
demodData = genqamdemod(rxSig,c);
[numErrors,ser] = symerr(data,demodData)
```

```
numErrors = 1
```

```
ser = 5.0000e-04
```

Repeat the transmission and demodulation process with an AWGN channel having a 10 dB SNR. Determine the symbol error rate for the reduced SNR. As expected, the performance degrades when the SNR is decreased.

```
rxSig = awgn(modData,10,'measured');
demodData = genqamdemod(rxSig,c);
[numErrors,ser] = symerr(data,demodData)
```

```
numErrors = 462

ser = 0.2310
```

Repeatable AWGN with RandStream

Generate white Gaussian noise addition results using a `RandStream` object and “Class” (MATLAB).

Specify the power of X to be 0 dBW, add noise to produce an SNR of 10dB, and utilize a local random stream.

```
S = RandStream('mt19937ar','Seed',5489);
sigin = sqrt(2)*sin(0:pi/8:6*pi);
sigout1 = awgn(sigin,10,0,S);
```

Add AWGN to `sigin. Use` `isequal` to compare `sigout1` to `sigout2`. The outputs are not equal when the random stream was not reset.

```
sigout2 = awgn(sigin,10,0,S);
isequal(sigout1,sigout2)
```

```
ans = logical
   0
```

Reset the random stream object, returning the object to its state prior to adding AWGN to `sigout1.` Add AWGN to `sigin` and compare `sigout1` to `sigout3`. The outputs are equal after the random stream was reset.

```
reset(S);
sigout3 = awgn(sigin,10,0,S);
isequal(sigout1,sigout3)
```

```
ans = logical
   1
```

Input Arguments

`in` — Input signal

scalar | vector | array

Input signal, specified as a scalar, vector, or array. The power of the input signal is assumed to be 0 dBW.

Data Types: `double`
Complex Number Support: Yes

`snr` — Signal-to-noise ratio

scalar

Signal-to-noise ratio in dB, specified as a scalar.

Note When the noise is added, this function applies the same `snr` to all elements of the full input signal. Array input signals do not have a notion of independent channels. To consider multiple channels independently, see `comm.AWGNChannel`.

Data Types: `double`

`signalpower` — Signal power

scalar | `'measured'`

Signal power, specified as a scalar or `'measured'`.

- When `signalpower` is a scalar, the value is used as the signal level of `in` to determine the appropriate noise level based on the value of `snr`.
- When `signalpower` is `'measured'`, the signal level of `in` is computed to determine the appropriate noise level based on the value of `snr`.

  Note When you specify `'measured'`, this function computes the signal power using all elements of the full input signal. When the power is computed, array input signals do not have a notion of independent channels.

Data Types: `double`

`randobject` — Random number stream object

`RandStream` object

Random number stream object, specified as a `RandStream` object. Generate normal random noise samples by using `randn`. The sequence of numbers produced by `randn` is determined by the internal state of the random stream object. `randn` uses one or more uniform values from the `RandStream` object to generate each normal value. Control the random stream object using the `reset` function and its properties.

Providing a random stream object or using the `reset` function on the default random stream object, along with a static `seed` value enables you to generate repeatable noise samples.

seed — Random number generator seed
scalar

Random number generator seed value, specified as a scalar.

Data Types: `double`

powertype — Signal power unit
`'dB'` (default) | `'linear'`

Signal power unit, specified as `'dB'` or `'linear'`

- When `powertype` is `'dB'`, the `snr` is measured in dB and `signalpower` is measured in dBW.
- When `powertype` is `'linear'`, the `snr` is measured as a ratio and `signalpower` is measured in watts.

For the relationships between SNR and other measures of the relative power of the noise, such as E_s/N_0, and E_b/N_0, see “AWGN Channel Noise Level”.

Output Arguments

out — Output signal
scalar | vector | array

Output signal, returned as a scalar, vector, or array. The returned output signal is the input signal with white Gaussian noise added to it.

See Also

Functions
`RandStream` | `bsc` | `randn` | `wgn`

System Objects
`comm.AWGNChannel`

Topics

"AWGN Channel Noise Level"

Introduced before R2006a

bchdec

BCH decoder

Syntax

```
decoded = bchdec(code,N,K)
decoded = bchdec(code,N,K,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. Parity symbols are at the end and the leftmost symbol is the most significant symbol.

In the `decoded` Galois array, each row represents the attempt at decoding the corresponding row in `code`.

`decoded = bchdec(code,N,K,paritypos)` specifies in `paritypos` whether the parity symbols in `code` were appended or prepended to the message in the coding operation.

`[decoded,cnumerr] = bchdec( ___ )` returns a column vector, `cnumerr`, where each element is the number of corrected errors in the corresponding row of `code`. You can return `cnumerr` with either of the preceding syntaxes.

`[decoded,cnumerr,ccode] = bchdec( ___ )` returns `ccode`, the corrected version of `code`.

Examples

Results of Error Correction

BCH-decode an input that has more errors per codeword than the error correcting capability of the BCH decoder. Decode a BCH coded message with two errors per codeword using a single-error correcting BCH decoder. View the effects of the error mismatch on the output codeword.

Check the number of errors per codeword a [63,57] BCH decoder is capable of correcting.

```
n = 63;
k = 57;
t = bchnumerr(n,k)
```

```
t = 1
```

The [63,57] BCH decoder is capable of correcting one error per codeword.

Create a random stream and use it to generate a GF array. Encode the message.

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

Add two errors per codeword and decode the errored code.

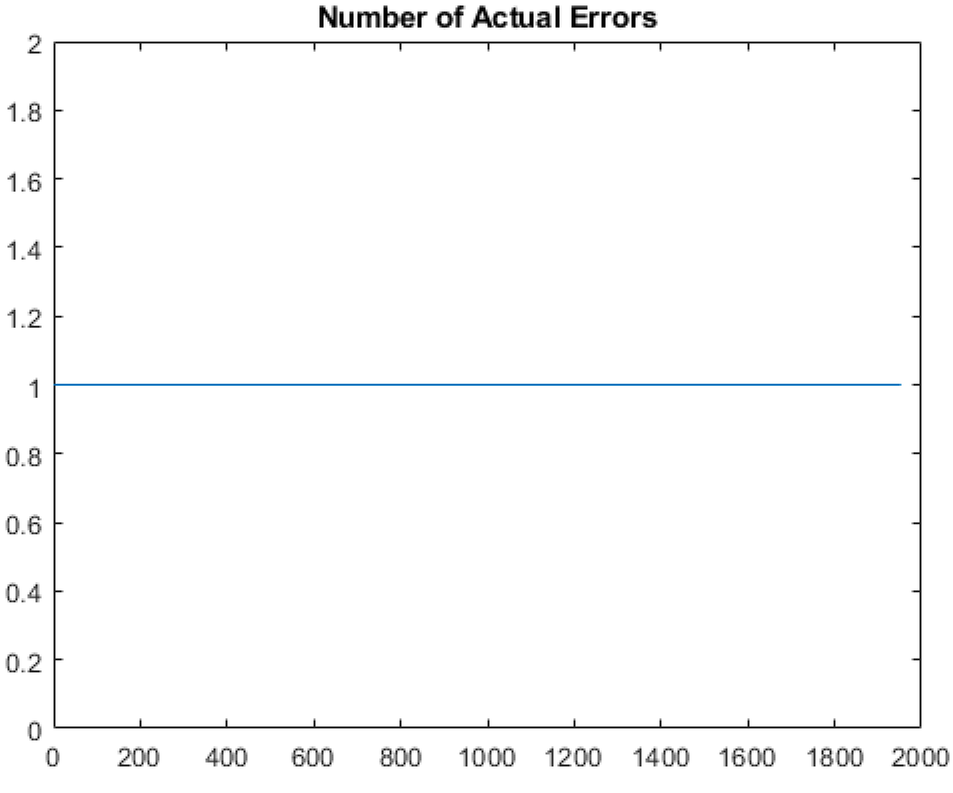
```
cnumerr2 = zeros(nchoosek(n,2),1);
nErrs = zeros(nchoosek(n,2),1);
cnumerrIdx = 1;
for idx1 = 1 : n-1
    %sprintf('idx1 for 2 errors = %d', idx1)
    for idx2 = idx1+1 : n
        errors = zeros(1,n);
        errors(idx1) = 1;
        errors(idx2) = 1;
        erroredCode = code + gf(errors);
 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

Encode the decoded message. Check that the re-encoded message differs from the errored message in only one bit.

```
        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 doubly-errored code and the re-encoded version of the initial decoding.

```
plot(nErrs)
title('Number of Actual Errors')
```



All inputs with two errors were decoded to a codeword that differs in exactly one bit from the re-encoded version.

Decode Received BCH Codeword in Noisy Channel

Set the BCH parameters for a Galois array of GF(2).

```
M = 4;
n = 2^M-1;   % Codeword length
k = 5;       % Message length
nwords = 10; % Number of words to encode
```

Create a message.

```
msgTx = gf(randi([0 1],nwords,k));
```

Find the error-correction capability.

```
t = bchnumerr(n,k)
```

```
t = 3
```

Encode the message.

```
enc = bchenc(msgTx,n,k);
```

Corrupt up to `t` bits in each codeword.

```
noisycode = enc + randerr(nwords,n,1:t);
```

Decode the noisy code.

```
msgRx = bchdec(noisycode,n,k);
```

Validate that the message was properly decoded.

```
isequal(msgTx,msgRx)
```

```
ans = logical
   1
```

Increase the number of possible errors, and generate another noisy codeword.

```
t2 = t + 1;
noisycode2 = enc + randerr(nwords,n,1:t2);
```

Decode the new received codeword.

```
[msgRx2,numerr] = bchdec(noisycode2,n,k);
```

Determine if the message was properly decoded by examining the number of corrected errors, `numerr`. Entries of `-1` correspond to decoding failures, which occur when the codeword has more errors than can be corrected for the specified `[n,k]` pair.

```
numerr
```

```
numerr = 10×1

     1
     2
    -1
     2
     3
     1
    -1
     4
     2
     3
```

Two of the ten transmitted codewords were not correctly received.

Input Arguments

code — Encoded message

Galois array

Encoded message, specified as a Galois array of symbols over GF(2). Each `N`-element row of `code` represents a corrupted systematic codeword.

For more information, see “Creating a Galois Array”.

N — Codeword length

integer

Codeword length, specified as an integer of the form `N` = 2^M–1, where M is an integer from 3 to 16. See “Tips” on page 1-79 for information about valid `N` values, valid (`N`,`K`) pairs, and error correcting capabilities for a given BCH code.

Example: 15 for *M*=4

K — Message length

integer

Message length, specified as an integer. N and K must produce a narrow-sense BCH code.

Example: 5 specifies a Galois array with five elements.

paritypos — Parity position

'end' (default) | 'beginning'

Parity position, specified as 'end' or 'beginning'. Parity symbols are at the end or beginning of each word in the output Galois array. 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.

Output Arguments

decoded — Decoded message

Galois array of symbols over GF(2)

Decoded message, returned as a Galois array of symbols over GF(2). 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 errors per codeword that the decoder is capable of correcting. When a decoding failure occurs, bchdec forms the corresponding row of decoded by removing N-K symbols from the end of the row of code. For more information, see “Error Correcting Capability” on page 1-79.

cnumerr — Number of corrected errors

column vector

Number of corrected errors in the corresponding row of code, returned as a column vector. A value of –1 in cnumerr indicates a decoding failure in that row in code.

ccode — Corrected version of code

Galois array

Corrected version of code, returned as a Galois array. ccode has the same format as the input code. If a decoding failure occurs in a certain row of code, the corresponding row in ccode contains that row unchanged.

Definitions

Error Correcting Capability

BCH decoders correct up to a specified number of errors per codeword based on the (N,K) pair used to BCH encode that message. The error correcting capability, *T*, of a given (N,K) pair is returned by bchnumerr. See “Tips” on page 1-79 for information about valid N values, valid (N,K) pairs, and error correcting capabilities for a given BCH code.

If the coded message contains more errors per codeword than the decoder is capable of correcting, the decoder is unlikely to decode to the correct codeword. For example, when a single-error-correcting BCH decoder (*T*=1) is given an input with two errors per codeword, it decodes it to a valid codeword but not the correct codeword. When a double-error-correcting BCH decoder (*T*=2) is given an input with three errors per codeword, the decoder sometimes decodes to an invalid codeword. The cnumerr and ccode output provide feedback to analyze the correctness of the decoded message.

Tips

- To generate the list of valid (N,K) pairs along with the corresponding values of the error-correction capability, run bchnumerr(N).
- Valid values for N = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of N is 65,535.

Algorithms

bchdec uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” on page 1-80.

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

Functions
`bchenc` | `bchgenpoly` | `bchnumerr`

System Objects
`comm.BCHDecoder`

Topics

"Block Codes"

Introduced before R2006a

bchenc

BCH encoder

Syntax

```
code = bchenc(msg,N,K)
code = bchenc(msg,N,K,paritypos)
```

Description

`code = bchenc(msg,N,K)` encodes the input message using an (N,K) BCH encoder that uses a narrow-sense generator polynomial. For a description of Bose–Chaudhuri–Hocquenghem (BCH) coding, see [1].

`code = bchenc(msg,N,K,paritypos)` appends or prepends the parity symbols to the encoded input message to form the output.

Examples

Decode Received BCH Codeword in Noisy Channel

Set the BCH parameters for a Galois array of GF(2).

```
M = 4;
n = 2^M-1;   % Codeword length
k = 5;       % Message length
nwords = 10; % Number of words to encode
```

Create a message.

```
msgTx = gf(randi([0 1],nwords,k));
```

Find the error-correction capability.

```
t = bchnumerr(n,k)
```

```
t = 3
```

Encode the message.

```
enc = bchenc(msgTx,n,k);
```

Corrupt up to `t` bits in each codeword.

```
noisycode = enc + randerr(nwords,n,1:t);
```

Decode the noisy code.

```
msgRx = bchdec(noisycode,n,k);
```

Validate that the message was properly decoded.

```
isequal(msgTx,msgRx)
```

```
ans = logical
   1
```

Increase the number of possible errors, and generate another noisy codeword.

```
t2 = t + 1;
noisycode2 = enc + randerr(nwords,n,1:t2);
```

Decode the new received codeword.

```
[msgRx2,numerr] = bchdec(noisycode2,n,k);
```

Determine if the message was properly decoded by examining the number of corrected errors, `numerr`. Entries of `-1` correspond to decoding failures, which occur when the codeword has more errors than can be corrected for the specified `[n,k]` pair.

```
numerr
```

```
numerr = 10×1

     1
     2
    -1
     2
     3
     1
    -1
```

```
     4
     2
     3
```

Two of the ten transmitted codewords were not correctly received.

Input Arguments

`msg` — Message to encode

Galois array of symbols over GF(2)

Message to encode, specified as 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.

For more information, see "Creating a Galois Array".

Example: `msgTx = gf(randi([0 1],10,5))`, where `msgTx` is a 10-by-5 `gf array`.

`N` — Codeword length

integer

Codeword length, specified as an integer of the form `N` = 2^M–1, where *M* is an integer from 3 through 16. For more information, see "Tips" on page 1-84.

Example: `15` for *M*=4

`K` — Message length

integer

Message length, specified as an integer. `N` and `K` must produce a narrow-sense BCH code.

Example: `5` specifies a Galois array with five elements

`paritypos` — Parity position

`'end'` (default) | `'beginning'`

Parity position, specified as `'end'` or `'beginning'`. Parity symbols are at the end or beginning of each word in the output Galois array.

Output Arguments

code — Encoded message
Galois array

Encoded message, returned as a Galois array. Parity symbols are at the end or beginning of each word in the output Galois array. To specify the position of the parity symbols, use the `paritypos` argument.

Tips

- To generate the list of valid (`N`,`K`) pairs along with the corresponding values of the error-correction capability, run `bchnumerr`(`N`).
- Valid values for `N` = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of `N` is 65,535.

References

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

See Also

Functions
`bchdec` | `bchgenpoly` | `bchnumerr` | `gf`

System Objects
`comm.BCHEncoder`

Topics

“Block Codes”
“Galois Field Computations”
“How Integers Correspond to Galois Field Elements”

Introduced before R2006a

bchgenpoly

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 either a polynomial character vector or 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

Create a BCH Generator Polynomial

Create two BCH generator polynomials based on different primitive polynomials.

Set the codeword and message lengths, n and k.

```
n = 15;
k = 11;
```

Create the generator polynomial and return the error correction capability, t.

```
[genpoly,t] = bchgenpoly(15,11)
```

```
genpoly = GF(2) array.

Array elements =

   1   0   0   1   1

t = 1
```

Create a generator polynomial for a (15,11) BCH code using a different primitive polynomial expressed as a character vector. Note that genpoly2 differs from genpoly, which uses the default primitive.

```
genpoly2 = bchgenpoly(15,11,'D^4 + D^3 + 1')
```

```
genpoly2 = GF(2) array.

Array elements =
    1    1    0    0    1
```

Limitations

The maximum allowable value of n is 65535.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

bchdec | bchenc | bchnumerr

Topics

"Block Codes"

Introduced before R2006a

bchnumerr

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`.

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

Examples

Determine Message Length Combinations for BCH Code

Calculate the possible message length combinations for a BCH code word length of `15`.

```
T = bchnumerr(15)
```

```
T = 3×3

    15    11     1
    15     7     2
    15     5     3
```

Compute the Correctable Errors for BCH Code

Calculate the number of correctable errors for BCH code 15,11

```
T = bchnumerr(15,11)
```

```
T = 1
```

Input Arguments

N — Codeword length

integer scalar

Codeword length, specified as an integer scalar. N must have the form 2^m-1 for some integer, m, between 3 and 16.

Example: 15

Data Types: single | double

K — Message length

integer scalar

Message length, specified as an integer scalar. N and K must produce a narrow-sense BCH code.

Example: 11

Data Types: single | double

Output Arguments

T — Number of correctable errors

scalar or matrix

Number of correctable errors, returned as a scalar or matrix value.

bchnumerr(N) returns a matrix with three columns. The first column lists N, the second column lists K, and the third column lists T.

`bchnumerr(N, K)` returns a scalar, which represents the number of correctable errors for the BCH code.

See Also

`bchdec` | `bchenc`

Topics

"Block Codes"
"BCH Codes"

Introduced before R2006a

berawgn

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, ...)
```

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 quadrature phase shift keying (OQPSK)	`'oqpsk'`
Differential phase shift keying (DPSK)	`'dpsk'`

Modulation Scheme	Second Input Argument
Pulse amplitude modulation (PAM)	`'pam'`
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 $J = 2^{\frac{k+1}{2}}$. When k is even, a square constellation of size $2^{\frac{k}{2}} \times 2^{\frac{k}{2}}$ is used.

`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.

Examples

Generate Theoretical BER Data for AWGN Channels

This example shows how to generate theoretical bit error rate data for several modulation schemes assuming an AWGN channel.

Create a vector of Eb/No values and set the modulation order, `M`.

```
EbNo = (0:10)';
M = 4;
```

Generate theoretical BER data for QPSK modulation by using the `berawgn` function.

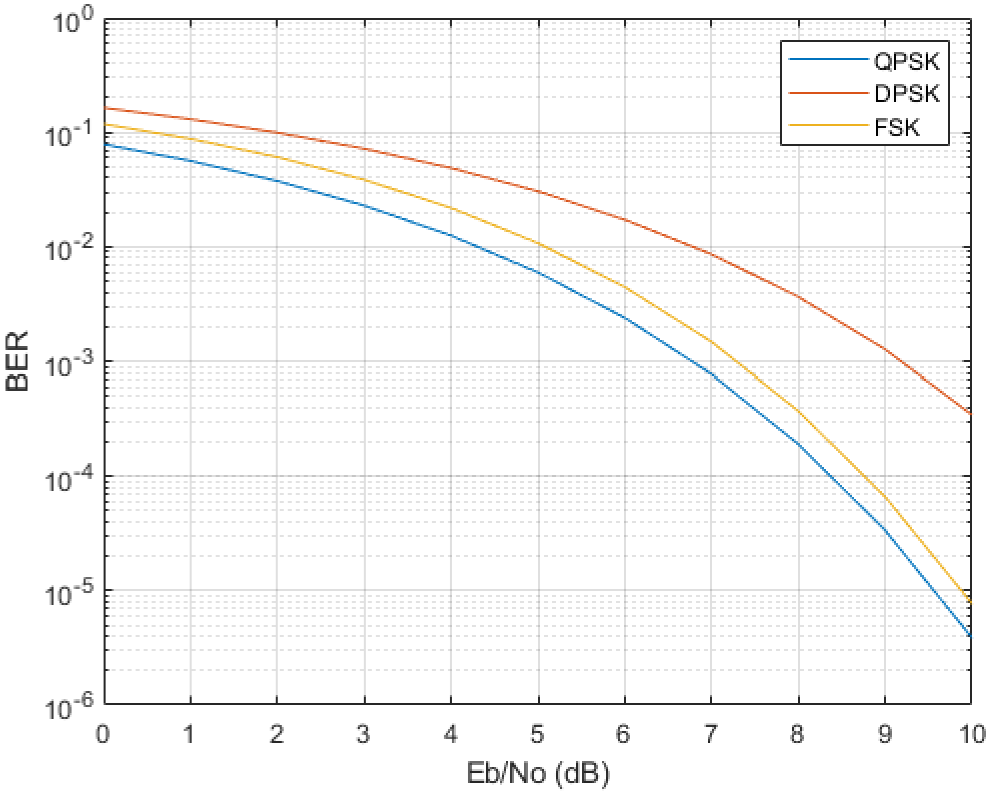
```
berQ = berawgn(EbNo,'psk',M,'nondiff');
```

Generate equivalent data for DPSK and FSK.

```
berD = berawgn(EbNo,'dpsk',M);
berF = berawgn(EbNo,'fsk',M,'coherent');
```

Plot the results.

```
semilogy(EbNo,[berQ berD berF])
xlabel('Eb/No (dB)')
ylabel('BER')
legend('QPSK','DPSK','FSK')
grid
```



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.

Alternatives

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

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` | `bertool`

Topics

"Theoretical Results"
Analytical Expressions Used in berawgn

Introduced before R2006a

bercoding

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)
berapprox = bercoding(...,modulation)
```

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 quadrature 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.

`berapprox = bercoding(...,`*`modulation`*`)` returns an approximation of the BER for coded AWGN channels when you specify a *`modulation`* type. See the `berawgn` function for a listing of the supported modulation types.

Examples

Upper Bound on Theoretical BER for a Block Code

Find an upper bound on the theoretical BER of a (23,12) block code.

Set the example parameters.

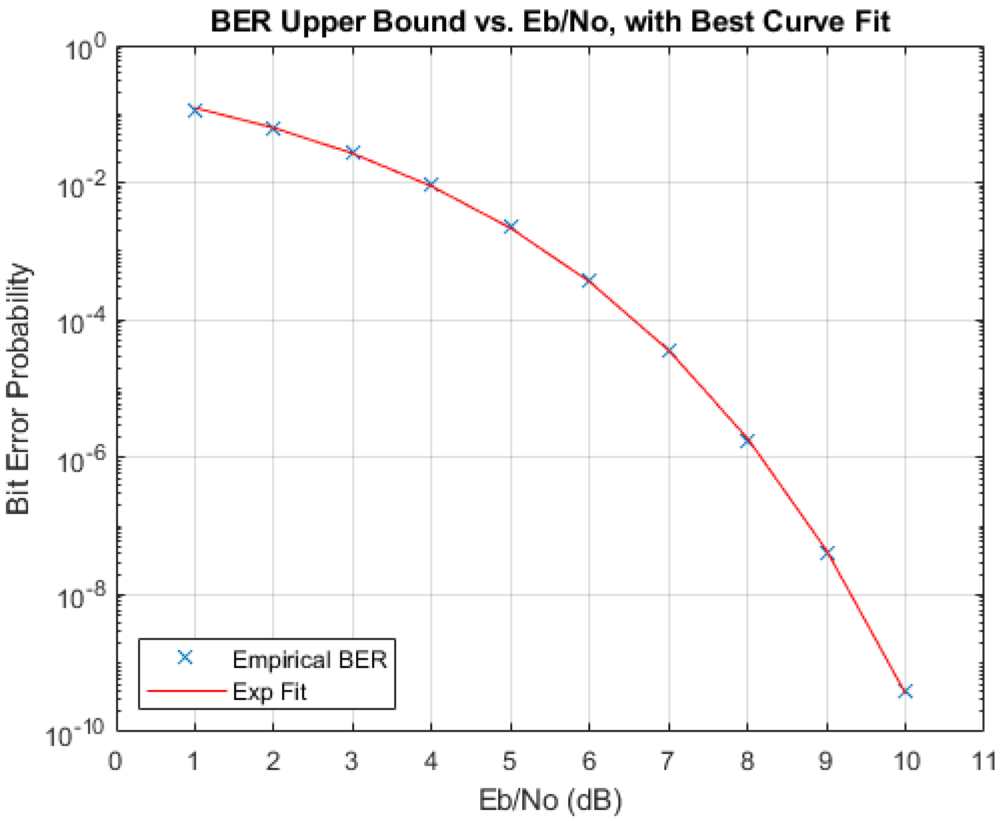
```
n = 23;             % Codeword length
k = 12;             % Message length
dmin = 7;           % Minimum distance
EbNo = 1:10;        % Eb/No range (dB)
```

Estimate the BER.

```
berBlk = bercoding(EbNo,'block','hard',n,k,dmin);
```

Plot the estimated BER.

```
berfit(EbNo,berBlk)
ylabel('Bit Error Probability')
title('BER Upper Bound vs. Eb/No, with Best Curve Fit')
```



Estimate Coded BER Performance of 16-QAM in AWGN

Estimate the performance of a 16-QAM channel in AWGN when encoded with a (15,11) Reed-Solomon code using hard-decision decoding.

Set the input Eb/No range and determine the uncoded BER for 16-QAM.

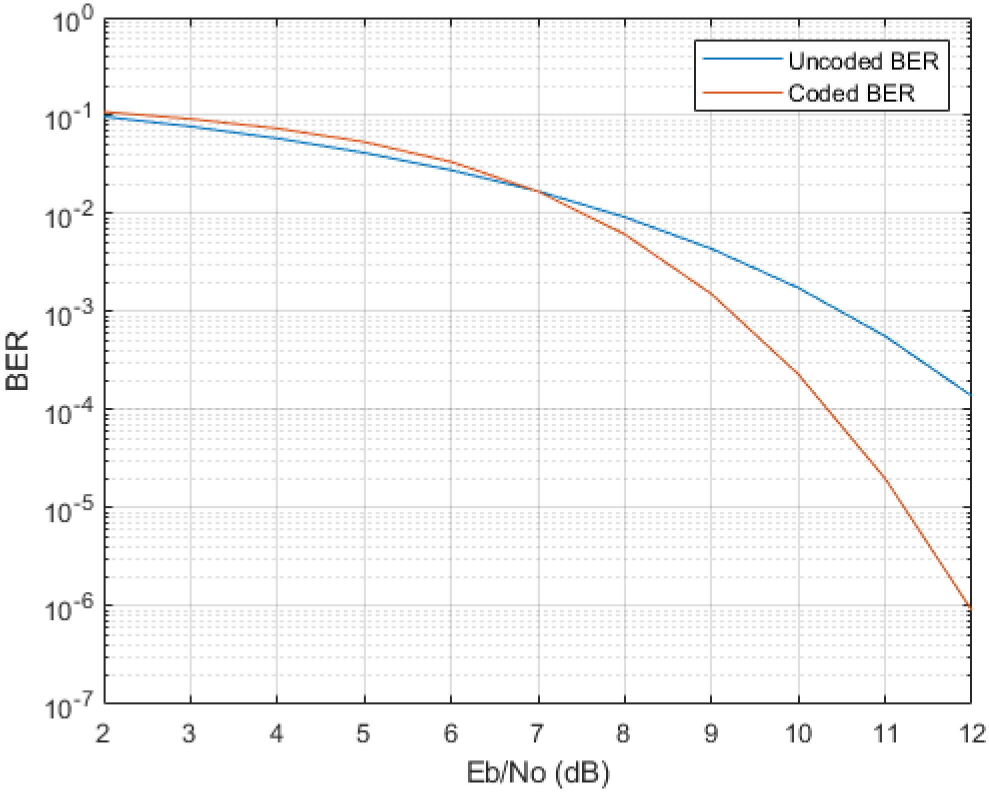
```
ebno = (2:12)';
uncodedBER = berawgn(ebno,'qam',16);
```

Estimate the coded BER for 16-QAM channel with a (15,11) Reed-Solomon code using hard decision decoding.

```
codedBER = bercoding(ebno,'RS','hard',15,11,'qam',16);
```

Plot the estimated BER curves.

```
semilogy(ebno,[uncodedBER codedBER])
grid
legend('Uncoded BER','Coded BER')
xlabel('Eb/No (dB)')
ylabel('BER')
```



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.

Alternatives

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

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` | `bertool` | `distspec`

Topics

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

Introduced before R2006a

berconfint

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

Introduced before R2006a

berfading

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, ...)
```

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

Estimate BER Performance of 16-QAM in Fading

Generate a vector of Eb/No values to evaluate.

```
EbNo = 8:2:20;
```

Initialize the BER results vector.

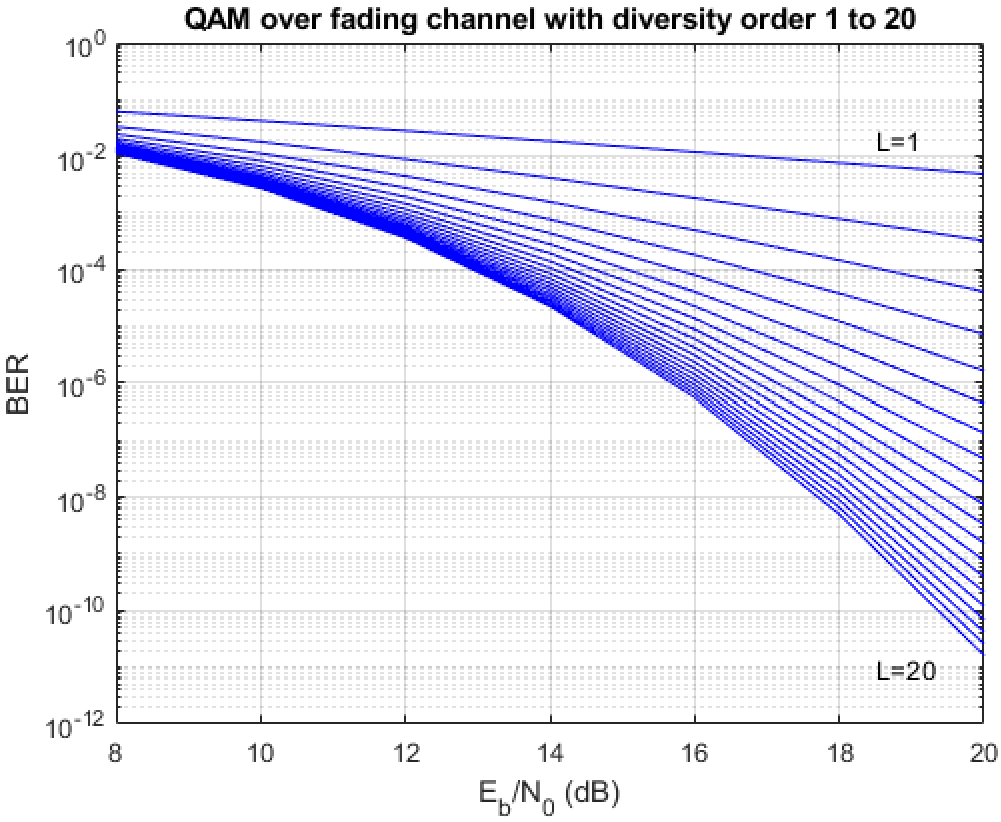
```
ber = zeros(length(EbNo),20);
```

Generate BER vs. Eb/No curves for 16-QAM in a fading channel. Vary the diversity order from 1 to 20.

```
for L = 1:20
    ber(:,L) = berfading(EbNo,'qam',16,L);
end
```

Plot the results.

```
semilogy(EbNo,ber,'b')
text(18.5, 0.02, sprintf('L=%d',1))
text(18.5, 1e-11, sprintf('L=%d',20))
title('QAM over fading channel with diversity order 1 to 20')
xlabel('E_b/N_0 (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.

Alternatives

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

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` | `bertool`

Topics

“Theoretical Performance Results”

Analytical Expressions Used in berfading

Introduced before R2006a

berfit

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 1-118 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.

Field	Description
`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 1-118 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 1

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

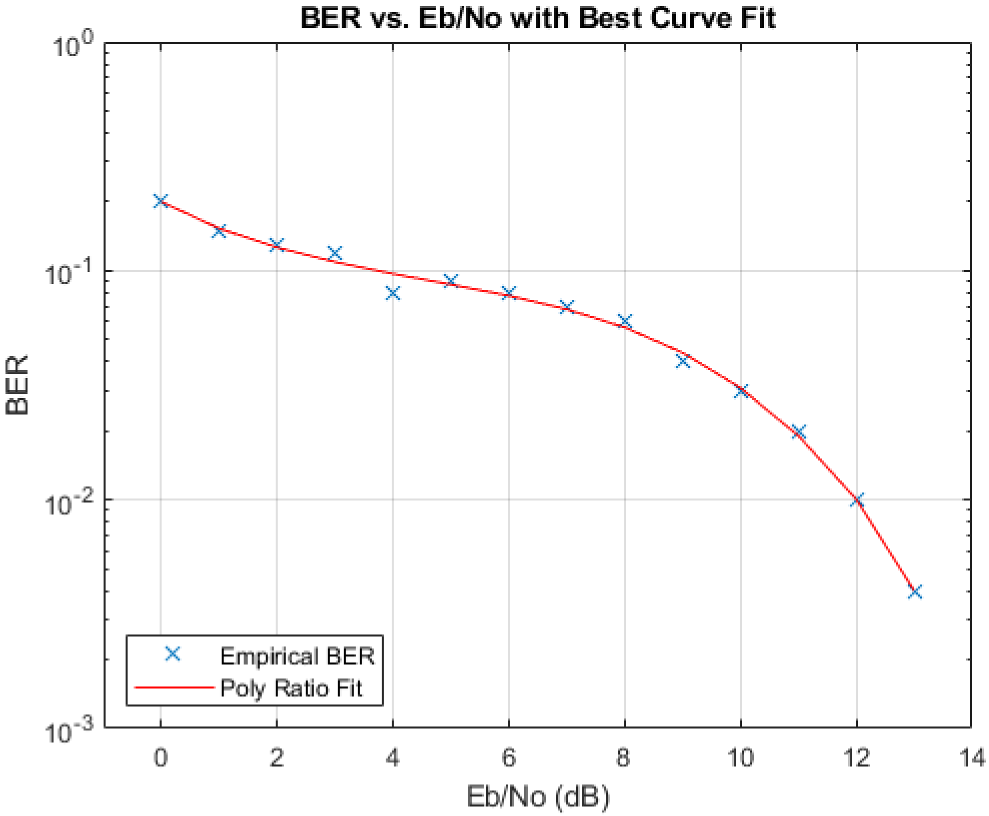
Examples

Bit Error Rate Curve Fitting

These examples illustrate the syntax of the `berfit` 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”.

Best fit for a sample set of data

```
EbN0 = 0:13;
berdata = [.2 .15 .13 .12 .08 .09 .08 .07 .06 .04 .03 .02 .01 .004];
berfit(EbN0,berdata);
```

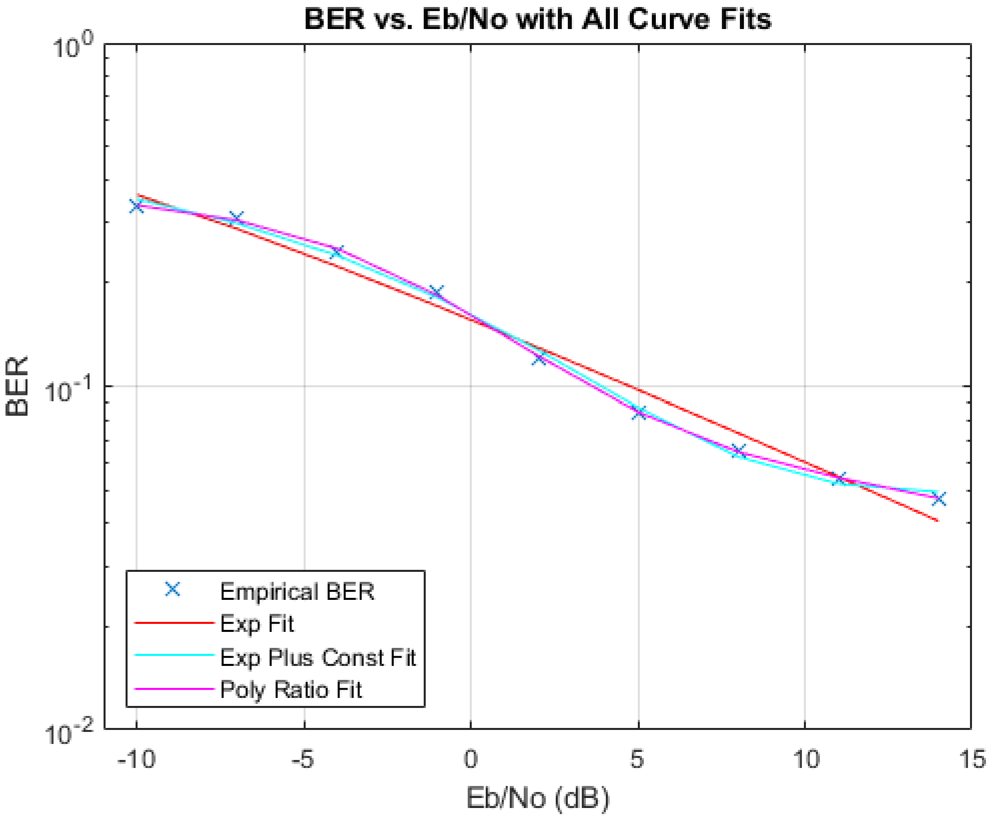


Plot the best fit. The curve connects the points created by evaluating the fit expression at the values in EbN0. To make the curve look smoother, use a syntax like `berfit(EbN0,berdata,[0:0.2:13])`. This alternative syntax uses more points when plotting the curve, but it does not change the fit expression.

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.

```
EbN0 = -10:3:15;
empBER = [0.3361 0.3076 0.2470 0.1878 0.1212 0.0845 0.0650 0.0540 0.0474];
figure; berfit(EbN0, empBER, [], [], 'all');
```



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 = 8; EbN0 = 3:10;
berdata = berfading(EbN0,'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
```

```
*** Trying exponential fit.
```

```
[fitber1,fitprops1] = berfit(EbN0,noisydata,EbN0,...
   options,'exp')
```

```
Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 2.749919

fitber1 = 1×8

    0.1247    0.0727    0.0376    0.0168    0.0064    0.0020    0.0005    0.0001 ⋯


fitprops1 = struct with fields:
       fitType: 'exp'
        coeffs: [4x1 double]
      sumSqErr: 2.7499
     exitState: 'The maximum number of function evaluations has been exceeded'
     funcCount: 10001
    iterations: 6193
```

```
disp('*** Trying polynomial ratio fit.') % Good fit
```

```
*** Trying polynomial ratio fit.
```

```
[fitber2,fitprops2] = berfit(EbN0,noisydata,EbN0,...
   options,'polyRatio')
```

```
fitber2 = 1×8

    0.1701    0.0874    0.0407    0.0169    0.0060    0.0016    0.0003    0.0001 ⋯


fitprops2 = struct with fields:
       fitType: 'polyRatio'
        coeffs: [6x1 double]
      sumSqErr: 2.3880
     exitState: 'The curve fit converged to a solution'
     funcCount: 554
    iterations: 331
```

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

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.

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

Introduced before R2006a

bersync

Bit error rate (BER) for imperfect synchronization

Syntax

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

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

Calculate Bit Error Rate (BER) for Imperfect Synchronization

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
```

```
ber = 3×3

   5.2073e-02   2.0536e-02   1.1160e-02
   1.8948e-02   7.9757e-04   4.9008e-06
   1.2501e-02   1.9091e-04   9.0060e-09
```

Switch back to default notation format.

```
format;
```

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)`

Condition	Behavior of Function
`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.

Alternatives

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

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` | `bertool`

Topics

"Theoretical Results"

Introduced before R2006a

bertool

Open bit error rate analysis GUI (BERTool)

Syntax

```
bertool
```

Description

`bertool` launches the Bit Error Rate Analysis Tool (BERTool). The BERTool application enables you to analyze the bit error rate (BER) performance of communications systems. BERTool computes the BER as a function of signal-to-noise ratio. It analyzes performance either with Monte-Carlo simulations of MATLAB functions and Simulink models or with theoretical closed-form expressions for selected types of communication systems. See “BERTool” to learn more.

Introduced before R2006a

bi2de

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.

`d = bi2de(b,flg)` converts a binary row vector to a decimal integer, where `flg` determines the position of the most significant digit.

`d = bi2de(b,p)` converts a base-`p` row vector `b` to a nonnegative decimal integer.

`d = bi2de(b,p,flg)` converts a base-`p` row vector to a decimal integer, where `flg` determines the position of the most significant digit.

Input Arguments

b — Binary input

row vector | matrix

Binary input specified as a row vector or matrix.

Example: `[0 1 0]`

Example: `[1 0 0; 1 0 1]`

Note `b` must represent an integer less than or equal to 2^{52}.

flg — MSB flag

'right-msb' | 'left-msb'

Character vector that determines whether the first column corresponds to the lowest-order or highest-order digit. If omitted, bi2de assumes 'right-msb'.

p — Base

positive integer scalar

The base of the row vector that is converted to a decimal. Specify as a positive integer greater than or equal to 2.

Example: 4

Output Arguments

d — Decimal output

scalar | vector

Decimal output converted from a base-p row vector b. Elements of d are nonnegative integers. If b is a matrix, each row represents a base-p number. In this case, the output d is a column vector in which each element is the decimal representation of the corresponding row of b.

Examples

Convert Binary Numbers to Decimals

Generate a matrix that contains binary representations of five random numbers between 0 and 15. Convert the binary numbers to decimal integers.

```
b = randi([0 1],5,4);
d = bi2de(b)
```

```
d = 5×1

     1
     5
    14
    11
    15
```

Convert a base-8 number to its decimal equivalent. Assign the most significant digit to the leftmost position. The output corresponds to $4(8^3)+2(8^2)+7(8^1)+1(8^0)=2233$.

```
d = bi2de([4 2 7 1],8,'left-msb')
```

```
d = 2233
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`de2bi`

Introduced before R2006a

bin2gray

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, matrix, or 3-D array. `modulation` is the modulation type and must be `'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

Binary to Gray Symbol Mapping

This example shows how to use the `bin2gray` and `gray2bin` functions to map integer inputs from a natural binary order symbol mapping to a Gray coded signal constellation and vice versa, assuming 16-QAM modulation. In addition, a visual representation of the difference between Gray and binary coded symbol mappings is shown.

Create a complete vector of 16-QAM integers.

```
x = (0:15)';
```

Convert the input vector from a natural binary order to a Gray encoded vector using `bin2gray`.

```
[y,mapy] = bin2gray(x,'qam',16);
```

Convert the Gray encoded symbols, `y`, back to a binary ordering using `gray2bin`.

```
z = gray2bin(y,'qam',16);
```

Verify that the original data, `x`, and the final output vector, `z` are identical.

```
isequal(x,z)
```

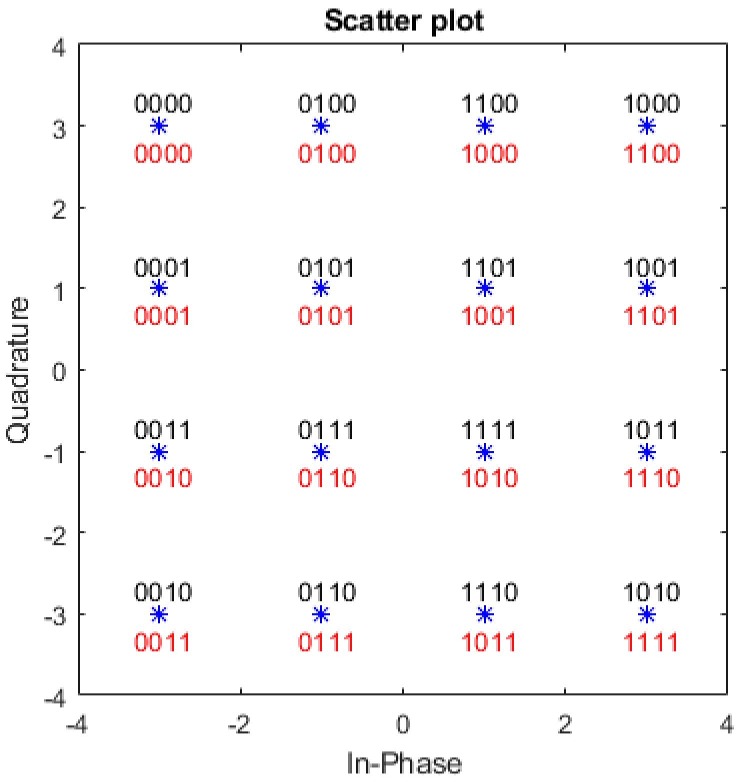
```
ans = logical
   1
```

To create a constellation plot showing the different symbol mappings, construct a 16-QAM modulator System object and use its associated `constellation` function to find the complex symbol values.

```
hMod = comm.RectangularQAMModulator;
symbols = constellation(hMod);
```

Plot the constellation symbols and label them using the Gray, `y`, and binary, `z`, output vectors. The binary representation of the Gray coded symbols is shown in black while the binary representation of the naturally ordered symbols is shown in red. Set the axes so that all points are displayed.

```
scatterplot(symbols,1,0,'b*');

for k = 1:16
    text(real(symbols(k))-0.3,imag(symbols(k))+0.3,...
        dec2base(mapy(k),2,4));

    text(real(symbols(k))-0.3,imag(symbols(k))-0.3,...
        dec2base(z(k),2,4),'Color',[1 0 0]);
end

axis([-4 4 -4 4])
```



Observe that only a single bit differs between adjacent constellation points when using Gray coding.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`gray2bin`

Topics

Gray Encoding a Modulated Signal

Introduced before R2006a

biterr

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

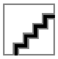
Syntax

```
[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.



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`
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

Bit Error Rate Computation

Create two binary matrices.

```
x = [0 0; 0 0; 0 0; 0 0]
```

```
x = 4×2

     0     0
     0     0
     0     0
     0     0
```

```
y = [0 0; 0 0; 0 0; 1 1]
```

```
y = 4×2

     0     0
     0     0
     0     0
     1     1
```

Determine the number of bit errors.

```
numerrs = biterr(x,y)
```

```
numerrs = 2
```

Determine the number of errors computed column-wise.

```
numerrs = biterr(x,y,[],'column-wise')
```

```
numerrs = 1×2

     1     1
```

Compute the number of row-wise errors.

```
numerrs = biterr(x,y,[],'row-wise')
numerrs = 4×1

     0
     0
     0
     2
```

Compute the number of overall errors. This has the same behavior as the default.

```
numerrs = biterr(x,y,[],'overall')
```

```
numerrs = 2
```

Estimate Bit Error Rate for 64-QAM in AWGN

Demodulate a noisy 64-QAM signal and estimate the bit error rate (BER) for a range of Eb/No values. Compare the BER estimate to theoretical values.

Set the simulation parameters.

```
M = 64;                 % Modulation order
k = log2(M);            % Bits per symbol
EbNoVec = (5:15)';      % Eb/No values (dB)
numSymPerFrame = 100;   % Number of QAM symbols per frame
```

Initialize the results vector.

```
berEst = zeros(size(EbNoVec));
```

The main processing loop executes the following steps:

- Generate binary data and convert to 64-ary symbols
- QAM modulate the data symbols
- Pass the modulated signal through an AWGN channel
- Demodulate the received signal
- Convert the demodulated symbols into binary data
- Calculate the number of bit errors

The `while` loop continues to process data until either 200 errors are encountered or 1e7 bits are transmitted.

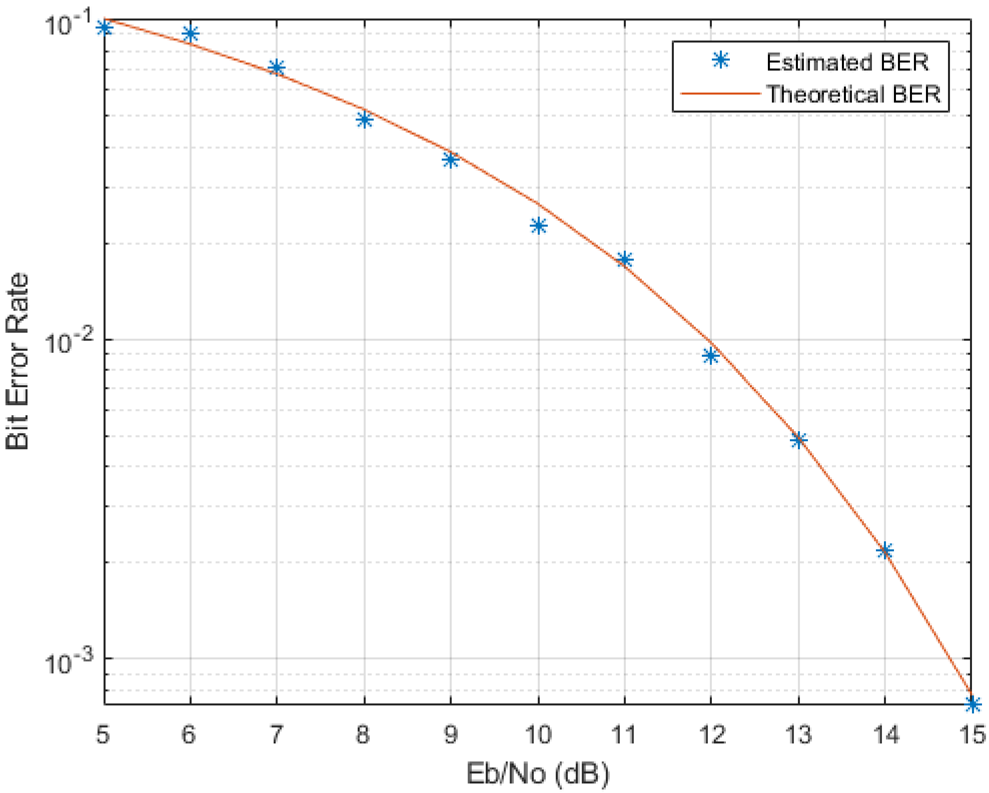
```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k);
    % Reset the error and bit counters
    numErrs = 0;
    numBits = 0;

    while numErrs < 200 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame,k);
        dataSym = bi2de(dataIn);

        % QAM modulate using 'Gray' symbol mapping
        txSig = qammod(dataSym,M);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal
        rxSym = qamdemod(rxSig,M);
        % Convert received symbols to bits
        dataOut = de2bi(rxSym,k);

        % Calculate the number of bit errors
        nErrors = biterr(dataIn,dataOut);

        % Increment the error and bit counters
        numErrs = numErrs + nErrors;
        numBits = numBits + numSymPerFrame*k;
    end

    % Estimate the BER
    berEst(n) = numErrs/numBits;
end
```

Determine the theoretical BER curve using `berawgn`.

```
berTheory = berawgn(EbNoVec,'qam',M);
```

Plot the estimated and theoretical BER data. The estimated BER data points are well aligned with the theoretical curve.

```
semilogy(EbNoVec,berEst,'*')
hold on
semilogy(EbNoVec,berTheory)
grid
legend('Estimated BER','Theoretical BER')
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



See Also

alignsignals | finddelay | symerr

Introduced before R2006a

bsc

Binary symmetric channel

Syntax

```
ndata = bsc(data,probability)
ndata = bsc(data,probability,streamhandle)
ndata = bsc(data,probability,seed)
[ndata,err] = bsc( ___ )
```

Description

`ndata = bsc(data,probability)` passes the binary input signal `data` through a binary symmetric channel having the specified error probability. The channel introduces a bit error and processes each element of the input `data` independently. `data` must be an array of binary numbers or a Galois array in GF(2). `probability` must be a scalar from 0 to 1.

`ndata = bsc(data,probability,streamhandle)` accepts a random stream handle to generate uniform noise samples by using `rand`. Providing a random stream handle or using the `reset` function on the default random stream object enables you to generate repeatable noise samples. For more information, see `RandStream`.

`ndata = bsc(data,probability,seed)` accepts a seed value, for initializing the uniform random number generator, `rand`. If you want to generate repeatable noise samples, then either reset the random stream input before calling `bsc` or use the same seed input.

`[ndata,err] = bsc( ___ )` returns an array containing the channel errors, using any of the preceding syntaxes.

Examples

Add Bit Errors to Bit Stream

Using the `bsc` function, introduce bit errors in the bits in a random matrix with probability 0.15.

```
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
```

```
numerrs = 1509
```

```
pcterrs = 0.1509
```

The output below is typical. For relatively small sets of data, the percentage of bit errors is not exactly 15% in most trials. If the size of the matrix `z` is large, the bit error percentage will be closer to the exact probability you specify.

Check for Errors After Decoding

Using the `bsc` function, introduce bit errors in the bits in a random matrix with probability 0.01. Use Viterbi decoder to decode message data.

Define trellis for Viterbi decoder. Generate and encode message data.

```
trel = poly2trellis([4 3],[4 5 17;7 4 2]);
msg = ones(10000,1);
```

Create objects for convolutional encoder, Viterbi decoder, and error rate calculator.

```
hEnc = comm.ConvolutionalEncoder(trel);
hVitDec = comm.ViterbiDecoder(trel, 'InputFormat','hard', 'TracebackDepth',...
    2, 'TerminationMethod', 'Truncated');
hErrorCalc = comm.ErrorRate;
```

Encode the message data. Introduce bit errors. Display the total number of errors.

```
code = hEnc(msg);
[ncode,err] = bsc(code,.01);
numchanerrs = sum(sum(err))
```

```
numchanerrs = 158
```

Decode the data and check the number of errors after decoding.

```
dcode = hVitDec(ncode);
berVec = hErrorCalc(msg, dcode);
ber = berVec(1)
```

```
ber = 0.0049
```

```
numsyserrs = berVec(2)
```

```
numsyserrs = 49
```

- “Design a Rate 2/3 Feedforward Encoder Using Simulink”

See Also

Functions
RandStream | awgn | gf | rand

Topics
“Design a Rate 2/3 Feedforward Encoder Using Simulink”

Introduced before R2006a

cdma2000ForwardReferenceChannels

Define cdma2000 forward reference channel

Syntax

```
cfg = cdma2000ForwardReferenceChannels(wv)
cfg = cdma2000ForwardReferenceChannels(wv,numchips)
cfg = cdma2000ForwardReferenceChannels(BSTM-RC,numchips,P,M)
cfg = cdma2000ForwardReferenceChannels(traffic,numchips,F-SCH-SPEC)
```

Description

`cfg = cdma2000ForwardReferenceChannels(wv)` returns a structure, `cfg`, that defines the cdma2000® forward link parameters given the input waveform identifier, `wv`. To generate a forward link reference channel waveform, pass this structure to the `cdma2000ForwardWaveformGenerator` function.

For all syntaxes, `cdma2000ForwardReferenceChannels` creates a configuration structure that is compliant with the cdma2000 physical layer specification [1].

`cfg = cdma2000ForwardReferenceChannels(wv,numchips)` specifies the number of chips to generate.

`cfg = cdma2000ForwardReferenceChannels(BSTM-RC,numchips,P,M)` returns the data structure for the `BSTM-RC` waveform identifiers, given the total traffic channel power, `P`, and the number of traffic channels, `M`. For more information on base station testing, see Table 6.5.2-1 of [2].

`cfg = cdma2000ForwardReferenceChannels(traffic,numchips,F-SCH-SPEC)` returns the data structure for the specified traffic channel, `traffic`, and the forward supplemental channel (F-SCH) and frame length combination, `F-SCH-SPEC`. If omitted, `F-SCH-SPEC` has a default value of the lowest F-SCH data rate allowable for a 20 ms frame length, given the radio configuration specified by `traffic`.

Examples

Generate Waveform for RC2 Forward Traffic Channels

Create a parameter structure, `config`, for all forward traffic channels (F-FCH and F-SCCH) that are supported by radio configuration 2.

```
config = cdma2000ForwardReferenceChannels('ALL-RC2')
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 1000
                  FPICH: [1x1 struct]
                 FAPICH: [1x1 struct]
                FTDPICH: [1x1 struct]
               FATDPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FSYNC: [1x1 struct]
                  FBCCH: [1x1 struct]
                  FCACH: [1x1 struct]
                  FCCCH: [1x1 struct]
                 FCPCCH: [1x1 struct]
                  FQPCH: [1x1 struct]
                   FFCH: [1x1 struct]
                  FOCNS: [1x1 struct]
                  FSCCH: [1x1 struct]
```

Examine the fields for the Forward Fundamental Channel (F-FCH). The data rate is 14,400 bps and the frame length is 20 ms.

```
config.FFCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC2'
              DataRate: 14400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 7
             EnableQOF: 'Off'
    PowerControlEnable: 'Off'
```

Generate the complex waveform using the corresponding waveform generator function.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

A waveform composed of the channels specified by each substructure of `config` is generated by `cdma2000ForwardWaveformGenerator`.

Generate CDMA200 Waveform Containing Sync Channel Message

Create a reference channel, specify the sync channel message as the data source, add the `SyncMessage` structure to the `FSYNC` substructure. Generate the waveform using this reference channel configuration.

Create a reference channel for testing a base station using radio configuration 3.

```
config = cdma2000ForwardReferenceChannels('BSTM-RC3');
```

Adjust the Forward Sync Channel (F-SYNC) settings. Set a relative channel power of 0.0 dB and specify the sync channel message as the data source.

```
config.FSYNC.Power = 0.0;
config.FSYNC.DataSource = 'SyncMessage';
```

Define the sync channel message structure (for P_REV = 6 (IS-2000-0)) and add it to the `config.FSYNC` substructure. Display the `FSYNC` structure.

```
sm = struct();
sm.P_REV = 6;                          % Protocol Revision field
sm.MIN_P_REV = 6;                      % Minimum Protocol Revision field
sm.SID = hex2dec('14B');               % System Identifier field
sm.NID = 1;                            % Network Identification field
sm.PILOT_PN = 0;                       % Pilot PN Offset field
sm.LC_STATE = hex2dec('20000000000');  % Long Code State field
sm.SYS_TIME = hex2dec('36AE0924C');    % System Time field
sm.LP_SEC = 0;                         % Leap Second field
sm.LTM_OFF = 0;                        % Local Time Offset field
sm.DAYLT = 0;                          % Daylight Savings Time Indicator field
sm.PRAT = 0;                           % Paging Channel Data Rate field
sm.CDMA_FREQ = hex2dec('2F6');         % CDMA Frequency field
sm.EXT_CDMA_FREQ = hex2dec('2F6');     % Extended CDMA Frequency field

config.FSYNC.SyncMessage = sm;

config.FSYNC
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    EnableCoding: 'On'
      DataSource: 'SyncMessage'
     SyncMessage: [1x1 struct]
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate F-CCCH Waveform

Create a structure for a 2000-chip forward common control channel (F-CCCH). Specify a 38,400 bps data rate, a 5 ms frame length, and an accompanying broadcast control channel (F-BCCH) with a 9600 bps data rate.

```
config   = cdma2000ForwardReferenceChannels('CONTROL-38400-5-9600',2000)
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2000
                  FPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FCCCH: [1x1 struct]
                  FBCCH: [1x1 struct]
```

Verify that the F-CCCH and F-BCCH data rates are 38,400 bps and 9600 bps, respectively.

```
config.FCCCH.DataRate
```

```
ans = 38400
```

```
config.FBCCH.DataRate
```

```
ans = 9600
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate Waveform for Base Station Testing

Create a reference channel for testing a base station using radio configuration 3. Specify the number of chips, the total power allocated to the individual channels, and the number of traffic channels. The FFCH substructure is a structure array whose dimensions are set by the number of traffic channels.

```
config = cdma2000ForwardReferenceChannels('BSTM-RC3',1000,-3,4)
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
```

```
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 1000
                  FPICH: [1x1 struct]
                  FSYNC: [1x1 struct]
                   FPCH: [1x1 struct]
                   FFCH: [1x4 struct]
```

Verify that the length of the FFCH substructure corresponds to the number of specified traffic channels, 4.

```
length(config.FFCH)
```

```
ans = 4
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate F-SCH Waveform

Create a traffic channel using radio configuration 7 composed of a 614,400 bps forward supplemental channel (F-SCH) having a 20 ms frame length. Set the number of chips to 5000.

```
config = cdma2000ForwardReferenceChannels('TRAFFIC-RC7-4800', ...
    5000,'F-SCH-614400-20')
```

```
config = struct with fields:
          SpreadingRate: 'SR3'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
```

```
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 5000
                  FPICH: [1x1 struct]
                   FFCH: [1x1 struct]
                   FSCH: [1x1 struct]
```

This channel uses spreading rate 3, `'SR3'`, which has a 3.75 MHz bandwidth.

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Input Arguments

`wv` — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector. The input typically identifies the channel type, radio configuration, data rate, and frame length. To specify `wv`, connect the substrings with hyphens, for example, `'CONTROL-19200-10-4800'`.

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
`wv`	`'FPICH-ONLY'`				Generates a waveform containing a pilot channel only.
	`'CONTROL'`	9600	20	4800 | 9600 | 19200	Character vector representing the forward common control
		19200	10 | 20		

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
		38400	5 | 10 | 20		channel (F-CCCH) data rate in bps, the frame length in ms, and the forward broadcast control channel (F-BCCH) data rate in bps. Specify 'CONTROL-9600-20-9600' to create a structure variable, wv, with a 9600 bps F-CCCH data rate, a 20 ms frame length, and a 9600 bps F-BCCH data rate.
	'TRAFFIC'	RC1	1200 | 2400 | 4800 | 9600	N/A	Character vector representing the radio configuration and the forward fundamental channel (F-FCH) data rate in bps.
		RC2 | RC5 | RC8 | RC9	1800 | 3600 | 7200 | 14400		

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
		RC3 | RC4 | RC6 | RC7	1500 | 2700 | 4800 | 9600		Specify 'TRAFFIC-RC9-14400' to create a channel with radio configuration 9 having a 14400 bps F-FCH data rate.
	'BSTM'	RC1 | RC2 | RC3 | RC4 | RC5 | RC6 | RC7 | RC8 | RC9	N/A	N/A	Models for testing the base station transmitter. Specify 'BSTM-RC1' to create a structure for base station testing with radio configuration 1.

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
	'ALL'	RC1 | RC2 | RC3 | RC4 | RC5 | RC6 | RC7 | RC8 | RC9	N/A	N/A	Returns all channels that are supported for the specified radio configuration. Specify 'ALL-RC4' to create a structure containing all traffic channels for radio configuration 4.

Example: 'CONTROL-9600-20-9600'

Example: 'TRAFFIC-RC9-7200'

Example: 'ALL-RC5'

Data Types: char

numchips — Number of chips

1000 (default) | positive integer scalar

Number of chips, specified as a positive integer.

Example: 1024

Data Types: double

BSTM-RC — BSTM reference channel type

'BSTM-RC1' | 'BSTM-RC2' | 'BSTM-RC3' | 'BSTM-RC4' | 'BSTM-RC5' | 'BSTM-RC6' | 'BSTM-RC7' | 'BSTM-RC8' | 'BSTM-RC9'

BSTM reference channel type, specified as a character vector. For more information, see Table 6.5.2-1 of [2].

Example: 'BSTM-RC8'

Data Types: char

P — Power budget allocated to traffic channels

0 (default) | real scalar

Power budget allocated to all traffic channels, specified in decibels as a real scalar.

Example: 5

Data Types: double

M — Number of traffic channels

6 (default) | positive integer scalar

Number of traffic channels, specified as a positive integer.

Example: 8

Data Types: double

traffic — Traffic configuration

character vector

Traffic channel configuration, specified as a character vector. The table shows the supported traffic channel configurations.

Radio Configuration	Traffic Channel Configuration			
1	'TRAFFIC-RC1-1200'	'TRAFFIC-RC1-2400'	'TRAFFIC-RC1-4800'	'TRAFFIC-RC1-9600'
2	'TRAFFIC-RC2-1800'	'TRAFFIC-RC2-3600'	'TRAFFIC-RC2-7200'	'TRAFFIC-RC2-14400'
3	'TRAFFIC-RC3-1500'	'TRAFFIC-RC3-2700'	'TRAFFIC-RC3-4800'	'TRAFFIC-RC3-9600'
4	'TRAFFIC-RC4-1500'	'TRAFFIC-RC4-2700'	'TRAFFIC-RC4-4800'	'TRAFFIC-RC4-9600'
5	'TRAFFIC-RC5-1800'	'TRAFFIC-RC5-3600'	'TRAFFIC-RC5-7200'	'TRAFFIC-RC5-14400'

Radio Configuration	Traffic Channel Configuration			
6	'TRAFFIC-RC6-1500'	'TRAFFIC-RC6-2700'	'TRAFFIC-RC6-4800'	'TRAFFIC-RC6-9600'
7	'TRAFFIC-RC7-1500'	'TRAFFIC-RC7-2700'	'TRAFFIC-RC7-4800'	'TRAFFIC-RC7-9600'
8	'TRAFFIC-RC8-1800'	'TRAFFIC-RC8-3600'	'TRAFFIC-RC8-7200'	'TRAFFIC-RC8-14400'
9	'TRAFFIC-RC9-1800'	'TRAFFIC-RC9-3600'	'TRAFFIC-RC9-7200'	'TRAFFIC-RC9-14400'

Example: 'TRAFFIC-RC6-4800' is a traffic channel that uses radio configuration 6 with a 4800 bps data rate.

Data Types: char

F-SCH-SPEC — Forward supplemental channel data rate and frame length
character vector

Forward supplemental channel data rate and frame length, specified as a character vector. The supported data rate and frame length combinations are summarized in the table.

<table>
<tr><th rowspan="2">Radio Configuration</th><th colspan="3">Frame Length</th></tr>
<tr><th>20 ms</th><th>40 ms</th><th>80 ms</th></tr>
<tr><td><b>3</b> | <b>4</b> | <b>6</b> | <b>7</b></td><td>'F-SCH-1500-20' | 'F-SCH-2700-20' | 'F-SCH-4800-20' | 'F-SCH-9600-20' | 'F-SCH-19200-20' | 'F-SCH-38400-20' | 'F-SCH-76800-20' | 'F-SCH-153600-20'</td><td>'F-SCH-1350-40' | 'F-SCH-2400-40' | 'F-SCH-4800-40' | 'F-SCH-9600-40' | 'F-SCH-19200-40' | 'F-SCH-38400-40' | 'F-SCH-76800-40'</td><td>'F-SCH-1200-80' | 'F-SCH-2400-80' | 'F-SCH-4800-80' | 'F-SCH-9600-80' | 'F-SCH-19200-80' | 'F-SCH-38400-80'</td></tr>
<tr><td><b>4</b> | <b>6</b> | <b>7</b></td><td>'F-SCH-307200-20'</td><td>'F-SCH-153600-40'</td><td>'F-SCH-76800-80'</td></tr>
</table>

Radio Configuration	Frame Length		
	20 ms	40 ms	80 ms
7	'F-SCH-614400-20'	'F-SCH-307200-40'	'F-SCH-153600-80'
5 | 8 | 9	'F-SCH-1800-20' | 'F-SCH-3600-20' | 'F-SCH-7200-20' | 'F-SCH-14400-20' | 'F-SCH-28800-20' | 'F-SCH-57600-20' | 'F-SCH-115200-20' | 'F-SCH-230400-20'	'F-SCH-1800-40' | 'F-SCH-3600-40' | 'F-SCH-7200-40' | 'F-SCH-14400-40' | 'F-SCH-28800-40' | 'F-SCH-57600-40' | 'F-SCH-115200-40'	'F-SCH-1800-80' | 'F-SCH-3600-80' | 'F-SCH-7200-80' | 'F-SCH-14400-80' | 'F-SCH-28800-80' | 'F-SCH-57600-80'
8 | 9	'F-SCH-460800-20'	'F-SCH-230400-40'	'F-SCH-115200-80'
9	'F-SCH-1036800-20'	'F-SCH-518400-40'	'F-SCH-259200-80'

For more data rate information for the cdma2000 forward links, see tables 3.1.3.1.3-2 and 3.1.3.1.3-4 of [1].

Example: `'F-SCH-460800-20'` is a supplemental channel with a 460,800 bps data rate and a 20 ms frame length.

Data Types: `char`

Output Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
SpreadingRate	'SR1' | 'SR3'	Spreading rate of the waveform. SR1 corresponds to a 1.2288 Mcps carrier. SR3 corresponds to a 3.6864 Mcps carrier. SR3 supports direct sequence spreading only.
Diversity	'NTD' | 'OTD' | 'STS'	Transmit diversity type (applicable only for SR1), where NTD is no transmit diversity, OTD is orthogonal transmit diversity, and STS is space time spreading
QOF	'QOF1' | 'QOF2' | 'QOF3'	Quasi-orthogonal function type
PNOffset	Nonnegative scalar integer	PN offset of the base station
LongCodeState	Positive scalar integer	Initial long code state
PowerNormalization	'Off' | 'NormalizeTo0dB' | 'NoiseFillTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients, used only when the FilterType field is set to 'Custom'
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
FPICH	Structure	See **FPICH Substructure**. Optional.
FAPICH	Structure	See **FAPICH Substructure**. Optional.
FTDPICH	Structure	See **FTDPICH Substructure**. Optional.

Parameter Field	Values	Description
FATDPICH	Structure	See **FATDPICH Substructure**. Optional.
FSYNC	Structure	See **FSYNC Substructure**. Optional.
FPCH	Structure	See **FPCH Substructure**. Optional.
FCCCH	Structure	See **FCCCH Substructure**. Optional.
FCACH	Structure	See **FCACH Substructure**. Optional.
FQPCH	Structure	See **FQPCH Substructure**. Optional.
FCPCCH	Structure	See **FCPCCH Substructure**. Optional.
FBCCH	Structure	See **FBCCH Substructure**. Optional.
FFCH	Structure	See **FFCH Substructure**. Optional.
FDCCH	Structure	See **FDCCH Substructure**. Optional.
FSCCH	Structure	See **FSCCH Substructure**. Optional.
FSCH	Structure	See **FSCH Substructure**. Optional.
FOCNS	Structure	See **FOCNS Substructure**. Optional.

FPICH Substructure

Include the FPICH substructure in the cfg structure to configure the forward pilot channel (F-PICH). The FPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FAPICH Substructure

Include the FAPICH substructure in the cfg structure to configure the forward auxiliary pilot channel (F-APICH). The FAPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256 | 512	Walsh code length

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FTDPICH Substructure

Include the FTDPICH substructure in the cfg structure to configure the forward transmit diversity pilot Channel (F-TDPICH). The FTDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FATDPICH

Include the FATDPICH substructure in the cfg structure to configure the forward auxiliary transmit diversity pilot channel (F-ATDPICH). The FATDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256, 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FSYNC Substructure

Include the FSYNC substructure in the cfg structure to configure the forward sync channel (F-SYNC). The FSYNC substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or 'SyncMessage'. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or a 'SyncMessage' character vector.
SyncMessage	Structure	See **SyncMessage Substructure**. Optional.

SyncMessage Substructure

If the DataSource field of the FSYNC substructure is set to 'SyncMessage', add the SyncMessage substructure to the cfg.FSYNC substructure to configure the sync channel message. The SyncMessage substructure contains these fields.

Parameter Field	Typical Value	Description
P_REV	6	Protocol revision field
MIN_P_REV	6	Minimum protocol revision field
SID	hex2dec('14B')	System identifier field
NID	1	Network identification field
PILOT_PN	0	Pilot PN offset field
LC_STATE	hex2dec('20000000000')	Long code state field
SYS_TIME	hex2dec('36AE0924C')	System time field
LP_SEC	0	Leap second field
LTM_OFF	0	Local time offset field
DAYLT	0	Daylight savings time indicator field

Parameter Field	Typical Value	Description
PRAT	0	Paging channel data rate field
CDMA_FREQ	hex2dec('2F6')	CDMA frequency field
EXT_CDMA_FREQ	hex2dec('2F6')	Extended CDMA frequency field

FPCH Substructure

Include the FPCH substructure in the cfg substructure to configure the forward paging channel (F-PCH). The FPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600	Data rate (bps)
LongCodeMask	Positive scalar integer	Long code identifier
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ 7	Walsh code number
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or a paging message character vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'. Paging message options include 'PagingMessage1', 'PagingMessage2', and 'PagingMessage3'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or one of three paging messages. For a description of paging message contents see footnote 1.

Parameter Field	Values	Description
1	When the DataSource enumeration specifies one of the paging message options, simulated paging message data is used as input to the F-PCH physical channel: • 'PagingMessage1' — Streams a 7680 bit sequence (800 ms at fullrate) of paging message contents onto the channel that includes the General Page Message, the CDMA Channel List Message, the Extended System Parameter Message, the Extended Neighbor List Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a nonsequential pattern. • 'PagingMessage2' — Streams a 2304 bit sequence (240 ms at fullrate) of paging message contents onto the channel that includes a truncated version of the full 'PagingMessage1' content. • 'PagingMessage3' — Streams an 864 bit sequence (90 ms at fullrate) of paging message contents onto the channel that includes the Neighbor List Message, the CDMA Channel List Message, the General Page Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a sequential pattern. For more information on the F-PCH contents, refer to 3GPP2 C.S0004, Table 3.1.2.3.1.1.2–1.	

FCCCH Substructure

Include the FCCCH substructure in the cfg structure to configure the forward common control channel (F-CCCH). The FCCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number

Parameter Field	Values	Description
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	`'On'` | `'Off'`	Enable or disable channel coding
DataSource	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCACH Substructure

Include the `FCACH` substructure in the `cfg` structure to configure the forward common assignment channel (F-CACH). The `FCACH` substructure contains these fields.

Parameter Field	Values	Description
Enable	`'On'` | `'Off'`	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
CodingType	`'conv'` | `'turbo'`	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	`'On'` | `'Off'`	Enable or disable channel coding
DataSource	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FQPCH Substructure

Include the `FQPCH` substructure in the `cfg` structure to configure the forward quick paging channel (F-QPCH). The `FQPCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`DataRate`	`2400` | `4800`	Data rate (bps)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCPCCH Substructure

Include the `FCPCCH` substructure in the `cfg` structure to configure the forward common power control channel (F-CPCCH). The `FCPCCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 63	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FBCCH Substructure

Include the FBCCH substructure in the cfg structure to configure the forward broadcast control channel (F-BCCH). The FBCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600 | 19200	Data rate (bps)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 127	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FFCH Substructure

Include the FFCH substructure in the cfg structure to configure the forward fundamental traffic channel (F-FCH). The FFCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' through 'RC9'	Radio configuration channel
DataRate	1200 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to F-FCH)

Parameter Field	Values	Description
PowerControlDataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

FDCCH Substructure

Include the FDCCH substructure in the cfg structure to configure the forward dedicated control channel (F-DCCH). The FDCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' through 'RC9'	Radio configuration channel
DataRate	9600 | 14400	Data rate (bps)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCCH Substructure

Include the FSCCH substructure in the cfg structure to configure the forward supplemental code channel (F-SCCH). The FSCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' | 'RC2'	Radio configuration channel
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCH Substructure

Include the FSCH substructure in the cfg structure to configure the forward supplemental channel (F-SCH). The FSCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' | 'RC4' | 'RC5' | 'RC6' | 'RC7' | 'RC8' | 'RC9'	Radio configuration channel

Parameter Field	Values	Description
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 307200	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
CodingType	'Conv' | 'Turbo'	Channel coding type, convolutional or turbo
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FOCNS Substructure

Include the FOCNS substructure in the cfg structure to configure orthogonal channel noise source information. The FOCNS substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256	Walsh code length

Parameter Field	Values	Description
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ `WalshLength` – 1	Walsh code number

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

[2] 3GPP2 C.S0010–C v2.0. “Recommended Minimum Performance Standards for cdma2000 Spread Spectrum Base Stations.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

[3] 3GPP2 C.S0004–F v1.0. “Signaling Link Access Control (LAC) Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`cdma2000ForwardWaveformGenerator` | `cdma2000ReverseReferenceChannels`

Introduced in R2015b

cdma2000ForwardWaveformGenerator

Generate cdma2000 forward link waveform

Syntax

```
[waveform1,waveform2] = cdma2000ForwardWaveformGenerator(cfg)
```

Description

`[waveform1,waveform2] = cdma2000ForwardWaveformGenerator(cfg)` returns the cdma2000 forward link baseband primary waveform, `waveform1`, and the forward link diversity waveform, `waveform2`, as defined by the parameter definition structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties the function uses to generate a cdma2000 waveform. You can generate `cfg` by using the `cdma2000ForwardReferenceChannels` function. The top-level parameters of `cfg` are `SpreadingRate`, `Diversity`, `QOF`, `PNOffset`, `LongCodeState`, `PowerNormalization`, `CustomFilterCoefficients`, `OversamplingRatio`, `FilterType`, `InvertQ`, `EnableModulation`, `ModulationFrequency`, and `NumChips`. To enable specific channels, add their associated substructures, for example, the forward paging channel, `FPCH`.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all combinations of spreading rate, radio configuration, frame length, and data rate are supported. To ensure that the input argument is valid, use the `cdma2000ForwardReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate Waveform for RC2 Forward Traffic Channels

Create a parameter structure, `config`, for all forward traffic channels (F-FCH and F-SCCH) that are supported by radio configuration 2.

```
config = cdma2000ForwardReferenceChannels('ALL-RC2')
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 1000
                  FPICH: [1x1 struct]
                 FAPICH: [1x1 struct]
                FTDPICH: [1x1 struct]
               FATDPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FSYNC: [1x1 struct]
                  FBCCH: [1x1 struct]
                  FCACH: [1x1 struct]
                  FCCCH: [1x1 struct]
                 FCPCCH: [1x1 struct]
                  FQPCH: [1x1 struct]
                   FFCH: [1x1 struct]
                  FOCNS: [1x1 struct]
                  FSCCH: [1x1 struct]
```

Examine the fields for the Forward Fundamental Channel (F-FCH). The data rate is 14,400 bps and the frame length is 20 ms.

```
config.FFCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC2'
```

```
              DataRate: 14400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 7
             EnableQOF: 'Off'
    PowerControlEnable: 'Off'
```

Generate the complex waveform using the corresponding waveform generator function.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

A waveform composed of the channels specified by each substructure of `config` is generated by `cdma2000ForwardWaveformGenerator`.

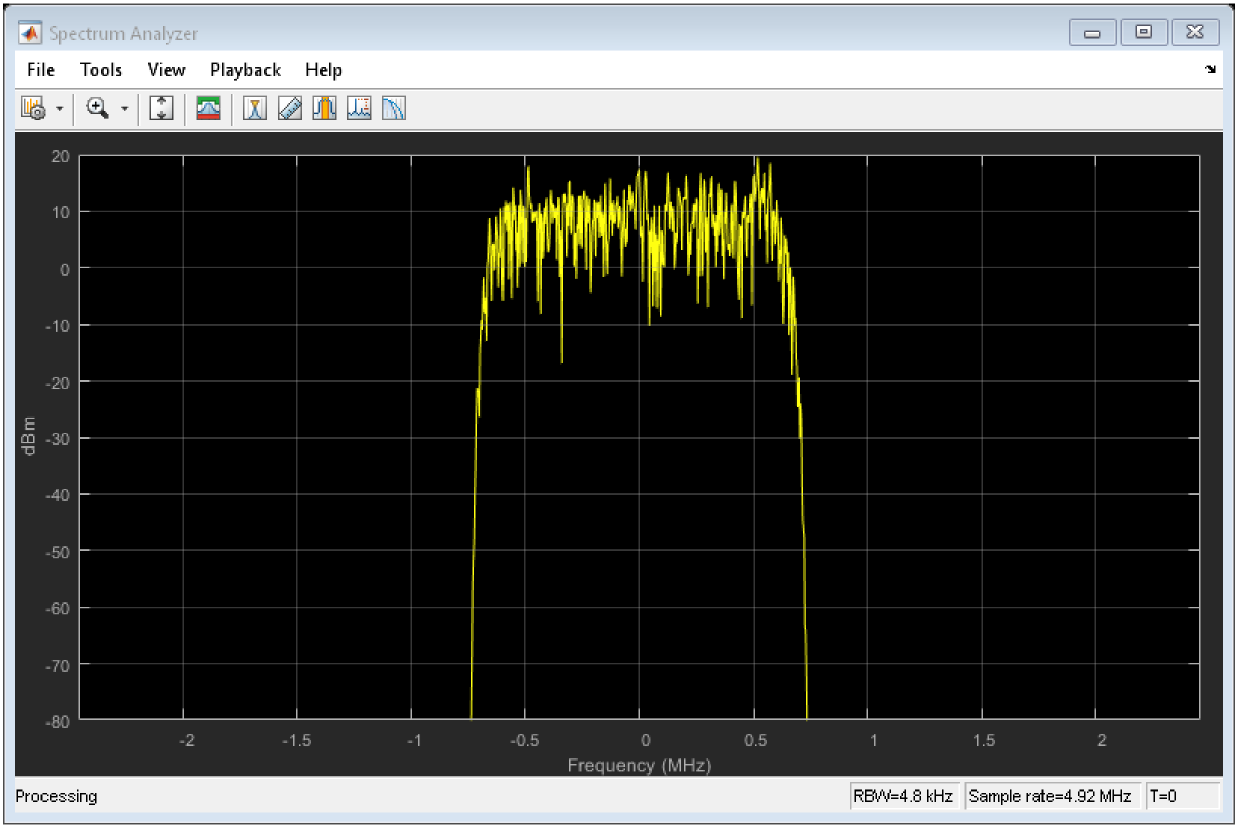
Generate Forward Traffic Channel for RC4

Configure a cdma2000 forward link supporting a 307.2 kbps forward supplemental channel (F-SCH) using radio configuration 4.

```
config = cdma2000ForwardReferenceChannels('TRAFFIC-RC4-4800',5000, ...
    'F-SCH-307200-20');
```

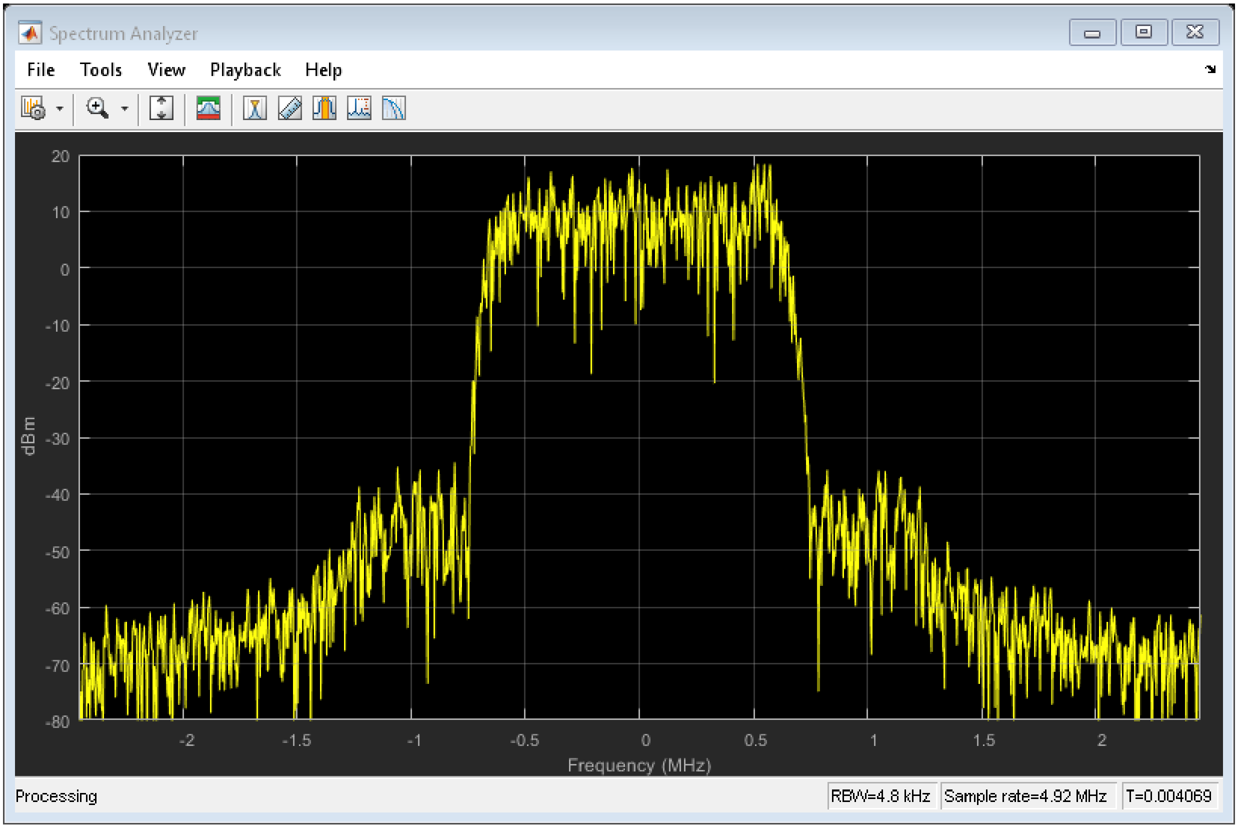
Generate the waveform and plot its spectrum. The sample rate is equal to the product of the chip rate and the oversampling ratio. RC4 uses spreading rate 1, which is equivalent to a 1.2288 Mcps chip rate.

```
wv = cdma2000ForwardWaveformGenerator(config);
fs = 1.2288e6 * config.OversamplingRatio;

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Change the filter type to `'cdma2000Short'` and plot the spectrum.

```
config.FilterType = 'cdma2000Short';
wv = cdma2000ForwardWaveformGenerator(config);
step(sa,wv)
```



The `'cdma2000Short'` filter does not provide as much out-of-band attenuation as does the `'cdma2000Long'` filter.

Generate cdma2000 Waveform with Two Forward Supplemental Channels

Create a parameter structure that specifies a forward traffic channel. Use it to generate a forward channel waveform.

Create a parameter structure specifying a traffic channel consisting of a 4800 bps fundamental channel, 5000 chips, and a 614.4 kbps supplemental channel (F-SCH) having a 20 ms frame duration.

```
cfg = cdma2000ForwardReferenceChannels('TRAFFIC-RC7-4800', ...
    5000,'F-SCH-614400-20');
```

Based on the first F-SCH, create a second F-SCH.

```
cfg(2).FSCH = cfg.FSCH;
```

Set the data rate of the second F-SCH to 38.4 kbps. Set the frame duration to 40 ms.

```
cfg(2).FSCH.DataRate = 38400;
cfg(2).FSCH.FrameLength = 40;
cfg.FSCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC7'
              DataRate: 614400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 2
             EnableQOF: 'Off'
            CodingType: 'conv'


ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC7'
              DataRate: 38400
           FrameLength: 40
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 2
             EnableQOF: 'Off'
            CodingType: 'conv'
```

Set the Walsh code of the second F-SCH so that it is not identical to the Walsh code of the first F-SCH.

```
cfg(2).FSCH.WalshCode = 3;
```

Generate the forward link waveform.

```
wv = cdma2000ForwardWaveformGenerator(cfg);
```

Input Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
`SpreadingRate`	`'SR1'` | `'SR3'`	Spreading rate of the waveform. `SR1` corresponds to a 1.2288 Mcps carrier. `SR3` corresponds to a 3.6864 Mcps carrier. SR3 supports direct sequence spreading only.
`Diversity`	`'NTD'` | `'OTD'` | `'STS'`	Transmit diversity type (applicable only for `SR1`), where `NTD` is no transmit diversity, `OTD` is orthogonal transmit diversity, and `STS` is space time spreading
`QOF`	`'QOF1'` | `'QOF2'` | `'QOF3'`	Quasi-orthogonal function type
`PNOffset`	Nonnegative scalar integer	PN offset of the base station
`LongCodeState`	Positive scalar integer	Initial long code state
`PowerNormalization`	`'Off'` | `'NormalizeTo0dB'` | `'NoiseFillTo0dB'`	Power normalization of the waveform
`NumChips`	Positive scalar integer	Number of chips in the waveform
`FilterType`	`'cdma2000Long'` | `'cdma2000Short'` | `'Off'` | `'Custom'`	Type of output filtering

Parameter Field	Values	Description
CustomFilterCoefficients	Real vector	Custom filter coefficients, used only when the FilterType field is set to 'Custom'
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
FPICH	Structure	See **FPICH Substructure**. Optional.
FAPICH	Structure	See **FAPICH Substructure**. Optional.
FTDPICH	Structure	See **FTDPICH Substructure**. Optional.
FATDPICH	Structure	See **FATDPICH Substructure**. Optional.
FSYNC	Structure	See **FSYNC Substructure**. Optional.
FPCH	Structure	See **FPCH Substructure**. Optional.
FCCCH	Structure	See **FCCCH Substructure**. Optional.
FCACH	Structure	See **FCACH Substructure**. Optional.
FQPCH	Structure	See **FQPCH Substructure**. Optional.
FCPCCH	Structure	See **FCPCCH Substructure**. Optional.
FBCCH	Structure	See **FBCCH Substructure**. Optional.
FFCH	Structure	See **FFCH Substructure**. Optional.
FDCCH	Structure	See **FDCCH Substructure**. Optional.
FSCCH	Structure	See **FSCCH Substructure**. Optional.
FSCH	Structure	See **FSCH Substructure**. Optional.
FOCNS	Structure	See **FOCNS Substructure**. Optional.

FPICH Substructure

Include the FPICH substructure in the cfg structure to configure the forward pilot channel (F-PICH). The FPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FAPICH Substructure

Include the FAPICH substructure in the cfg structure to configure the forward auxiliary pilot channel (F-APICH). The FAPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256 | 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FTDPICH Substructure

Include the FTDPICH substructure in the cfg structure to configure the forward transmit diversity pilot Channel (F-TDPICH). The FTDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FATDPICH

Include the FATDPICH substructure in the cfg structure to configure the forward auxiliary transmit diversity pilot channel (F-ATDPICH). The FATDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel

Parameter Field	Values	Description
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256, 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FSYNC Substructure

Include the FSYNC substructure in the cfg structure to configure the forward sync channel (F-SYNC). The FSYNC substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or 'SyncMessage'. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or a 'SyncMessage' character vector.
SyncMessage	Structure	See **SyncMessage Substructure**. Optional.

SyncMessage Substructure

If the DataSource field of the FSYNC substructure is set to 'SyncMessage', add the SyncMessage substructure to the cfg.FSYNC substructure to configure the sync channel message. The SyncMessage substructure contains these fields.

Parameter Field	Typical Value	Description
P_REV	6	Protocol revision field

Parameter Field	Typical Value	Description
MIN_P_REV	6	Minimum protocol revision field
SID	hex2dec('14B')	System identifier field
NID	1	Network identification field
PILOT_PN	0	Pilot PN offset field
LC_STATE	hex2dec('20000000000')	Long code state field
SYS_TIME	hex2dec('36AE0924C')	System time field
LP_SEC	0	Leap second field
LTM_OFF	0	Local time offset field
DAYLT	0	Daylight savings time indicator field
PRAT	0	Paging channel data rate field
CDMA_FREQ	hex2dec('2F6')	CDMA frequency field
EXT_CDMA_FREQ	hex2dec('2F6')	Extended CDMA frequency field

FPCH Substructure

Include the FPCH substructure in the cfg substructure to configure the forward paging channel (F-PCH). The FPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600	Data rate (bps)
LongCodeMask	Positive scalar integer	Long code identifier
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ 7	Walsh code number
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
`DataSource`	Cell array: `{'PN Type', RN Seed}`, binary vector, or a paging message character vector. Standard PN types are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`. Paging message options include `'PagingMessage1'`, `'PagingMessage2'`, and `'PagingMessage3'`.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or one of three paging messages. For a description of paging message contents see footnote 1.

1 When the `DataSource` enumeration specifies one of the paging message options, simulated paging message data is used as input to the F-PCH physical channel:

- `'PagingMessage1'` — Streams a 7680 bit sequence (800 ms at fullrate) of paging message contents onto the channel that includes the General Page Message, the CDMA Channel List Message, the Extended System Parameter Message, the Extended Neighbor List Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a nonsequential pattern.
- `'PagingMessage2'` — Streams a 2304 bit sequence (240 ms at fullrate) of paging message contents onto the channel that includes a truncated version of the full `'PagingMessage1'` content.
- `'PagingMessage3'` — Streams an 864 bit sequence (90 ms at fullrate) of paging message contents onto the channel that includes the Neighbor List Message, the CDMA Channel List Message, the General Page Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a sequential pattern.

For more information on the F-PCH contents, refer to 3GPP2 C.S0004, Table 3.1.2.3.1.1.2–1.

FCCCH Substructure

Include the FCCCH substructure in the cfg structure to configure the forward common control channel (F-CCCH). The FCCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On'|'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCACH Substructure

Include the FCACH substructure in the cfg structure to configure the forward common assignment channel (F-CACH). The FCACH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
CodingType	'conv' | 'turbo'	Type of error correction coding

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FQPCH Substructure

Include the FQPCH substructure in the cfg structure to configure the forward quick paging channel (F-QPCH). The FQPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	2400 | 4800	Data rate (bps)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCPCCH Substructure

Include the FCPCCH substructure in the cfg structure to configure the forward common power control channel (F-CPCCH). The FCPCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 63	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FBCCH Substructure

Include the FBCCH substructure in the cfg structure to configure the forward broadcast control channel (F-BCCH). The FBCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600 | 19200	Data rate (bps)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 127	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FFCH Substructure

Include the FFCH substructure in the cfg structure to configure the forward fundamental traffic channel (F-FCH). The FFCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' through 'RC9'	Radio configuration channel
DataRate	1200 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to F-FCH)
PowerControlDataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

FDCCH Substructure

Include the FDCCH substructure in the cfg structure to configure the forward dedicated control channel (F-DCCH). The FDCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
RadioConfiguration	'RC3' through 'RC9'	Radio configuration channel
DataRate	9600 | 14400	Data rate (bps)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCCH Substructure

Include the FSCCH substructure in the cfg structure to configure the forward supplemental code channel (F-SCCH). The FSCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' | 'RC2'	Radio configuration channel
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCH Substructure

Include the FSCH substructure in the cfg structure to configure the forward supplemental channel (F-SCH). The FSCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' | 'RC4' | 'RC5' | 'RC6' | 'RC7' | 'RC8' | 'RC9'	Radio configuration channel
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 307200	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
CodingType	'Conv' | 'Turbo'	Channel coding type, convolutional or turbo
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

Parameter Field	Values	Description
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FOCNS Substructure

Include the FOCNS substructure in the cfg structure to configure orthogonal channel noise source information. The FOCNS substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number

Output Arguments

waveform1 — Modulated baseband waveform comprising the primary physical channels

complex vector array

Modulated baseband waveform comprising the primary cdma2000 physical channels, returned as a complex vector array.

`waveform2` — Modulated baseband waveform comprising the diversity physical channels

complex vector array

Modulated baseband waveform comprising the diversity cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

[2] 3GPP2 C.S0004–F v1.0. “Signaling Link Access Control (LAC) Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`cdma2000ForwardReferenceChannels` | `cdma2000ReverseWaveformGenerator`

Introduced in R2015b

cdma2000ReverseReferenceChannels

Define cdma2000 reverse reference channel

Syntax

```
cfg = cdma2000ReverseReferenceChannels(wv)
cfg = cdma2000ReverseReferenceChannels(wv,numchips)
cfg = cdma2000ReverseReferenceChannels(traffic,numchips,R-SCH-SPEC)
```

Description

`cfg = cdma2000ReverseReferenceChannels(wv)` returns a structure, `cfg`, that defines the cdma2000 reverse link parameters given the input waveform identifier, `wv`. Pass the structure to the `cdma2000ReverseWaveformGenerator` function to generate a reverse link reference channel waveform.

For all syntaxes, `cdma2000ReverseReferenceChannels` creates a configuration structure that is compliant with the physical layer specification for cdma2000 systems described in [1].

`cfg = cdma2000ReverseReferenceChannels(wv,numchips)` specifies the number of chips to generate.

`cfg = cdma2000ReverseReferenceChannels(traffic,numchips,R-SCH-SPEC)` returns `cfg` for the specified traffic channel, `traffic`, and the reverse supplemental channel (R-SCH) and frame length combination, `R-SCH-SPEC`.

Examples

Generate Reverse Common Control Channel Waveform

Generate the structure corresponding to the reverse common control channel (R-CCCH) having a 19,200 bps data rate and 10 ms frames.

```
config = cdma2000ReverseReferenceChannels('R-CCCH-19200-10');
```

Verify that the R-CCCH substructure is configured for the correct data rate and frame duration.

```
config.RCCCH
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 19200
     FrameLength: 10
       WalshCode: 1
```

Generate the reverse channel waveform using the corresponding waveform generator function, `cdma2000ReverseWaveformGenerator`.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate Reverse Channels for RC1 and RC6

Create a configuration structure to generate all possible channels associated with radio configuration 1 in which the number of chips is specified as 2500.

```
config = cdma2000ReverseReferenceChannels('ALL-RC1',2500)
```

```
config = struct with fields:
     RadioConfiguration: 'RC1'
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2500
                   RFCH: [1x1 struct]
                   RACH: [1x1 struct]
                  RSCCH: [1x1 struct]
```

The structure contains substructures corresponding to the R-FCH, R-ACH, and R-SCCH channels.

Call the function again using radio configuration 6.

```
config = cdma2000ReverseReferenceChannels('ALL-RC6',2500)
```

```
config = struct with fields:
     RadioConfiguration: 'RC6'
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2500
                   RFCH: [1x1 struct]
                  RPICH: [1x1 struct]
                  REACH: [1x1 struct]
                  RCCCH: [1x1 struct]
                  RDCCH: [1x1 struct]
                  RSCH1: [1x1 struct]
                  RSCH2: [1x1 struct]
```

The channels supported by RC6 differ from those supported by RC1. They include R-FCH, R-PICH, R-EACH, R-CCCH, R-DCCH, R-SCH1, and R-SCH2.

Create the waveform corresponding to the set of RC6 channels.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate Reverse Supplemental Channel

Create a configuration structure using radio configuration 3 with a reverse fundamental channel (R-FCH). Specify a 2700 bps data rate and a reverse supplemental channel (R-SCH) having a 76,800 bps data rate and an 80 ms frame length.

```
config = cdma2000ReverseReferenceChannels('TRAFFIC-RC3-2700',2000, ...
    'R-SCH-76800-80');
```

Verify that the R-FCH data rate is 2700 bps and the first R-SCH data rate is 76,800 bps with an 80 ms frame length.

```
config.RFCH.DataRate
```

```
ans = 2700
```

```
config.RSCH1.DataRate
```

```
ans = 76800
```

```
config.RSCH1.FrameLength
```

```
ans = 80
```

Generate the corresponding waveform.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector. The input typically identifies the channel type, radio configuration, data rate, and frame length. To specify wv, connect the substrings with hyphens, for example, 'TRAFFIC-RC2-3600'.

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
wv	'R-PICH-ONLY'			Generates a waveform containing a pilot channel only.
	'R-CCCH'	9600	20	Character vector representing the Reverse
		19200	10 | 20	

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
		38400	5 | 10 | 20	Common Control Channel (R-CCCH) data rate in bps and the frame length in ms. Specify 'R-CCCH-9600-20' to create a structure variable, wv, with a 9600 bps R-CCCH data rate and a 20 ms frame length.
	'TRAFFIC'	RC1	1200 | 2400 | 4800 | 9600	Character vector representing the radio configuration and the Reverse Fundamental Channel (R-FCH) data rate in bps. Specify 'TRAFFIC-RC6-14400', corresponds to radio configuration 6 with a 14400 bps R-FCH data rate.
		RC2 | RC4 | RC6	1800 | 3600 | 7200 | 14400	
		RC3 | RC5 | RC6	1500 | 2700 | 4800 | 9600	
	'R-EACH'	9600	20	Reverse Enhanced Access Channel waveforms. Specify 'R-
		19200	10 | 20	

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
		38400	5 | 10 | 20	EACH-38400-5' to create a structure corresponding to an R-EACH channel with a 38400 bps data rate and a 5 ms frame length.
	'R-PICH-R-FCH'			Specify tests for the mobile transmitter in accordance with [2].
	'ALL'	RC1 | RC2 | RC3 | RC4 | RC5 | RC6	N/A	Returns all channels that are supported for the specified radio configuration. Specify 'ALL-RC4' to create a structure containing all traffic channels for radio configuration 4.

Example: 'R-CCCH-9600-20' is a R-CCH channel having a 9600 bps data rate and a 20 ms frame length.

Example: 'R-EACH-38400-5' is a R-EACH channel having a 38,400 bps data rate and a 5 ms frame length.

Data Types: char

numchips — Number of chips

1000 (default) | positive integer scalar

Number of chips, specified as a positive integer.

Example: `2048`

Data Types: `double`

`traffic` — Traffic configuration

character vector

Traffic channel configuration, specified as a character vector. The table shows the valid configurations.

Radio Configuration	Traffic Channel Configuration			
1	`'TRAFFIC-RC1-1200'`	`'TRAFFIC-RC1-2400'`	`'TRAFFIC-RC1-4800'`	`'TRAFFIC-RC1-9600'`
2	`'TRAFFIC-RC2-1800'`	`'TRAFFIC-RC2-3600'`	`'TRAFFIC-RC2-7200'`	`'TRAFFIC-RC2-14400'`
3	`'TRAFFIC-RC3-1500'`	`'TRAFFIC-RC3-2700'`	`'TRAFFIC-RC3-4800'`	`'TRAFFIC-RC3-9600'`
4	`'TRAFFIC-RC4-1800'`	`'TRAFFIC-RC4-3600'`	`'TRAFFIC-RC4-7200'`	`'TRAFFIC-RC4-14400'`
5	`'TRAFFIC-RC5-1500'`	`'TRAFFIC-RC5-2700'`	`'TRAFFIC-RC5-4800'`	`'TRAFFIC-RC5-9600'`
6	`'TRAFFIC-RC6-1800'`	`'TRAFFIC-RC6-3600'`	`'TRAFFIC-RC6-7200'`	`'TRAFFIC-RC6-14400'`

Example: `'TRAFFIC-RC4-1800'` is a traffic channel using radio configuration 4 and having an R-FCH with an 1800 bps data rate .

Data Types: `char`

`R-SCH-SPEC` — Reverse Supplemental Channel data rate and frame length

character vector

Specify the R-SCH data rate and frame length as a character vector. If omitted, `R-SCH-SPEC` defaults to the lowest R-SCH data rate allowable for a 20 ms frame length given the radio configuration specified by `traffic`. The table summarizes the supported data rate and frame length combinations.

Radio Configuration	Frame Length		
	20 ms	40 ms	80 ms
3 | 5	'R-SCH-1500-20' | 'R-SCH-2700-20' | 'R-SCH-4800-20' | 'R-SCH-9600-20' | 'R-SCH-19200-20' | 'R-SCH-38400-20' | 'R-SCH-76800-20' | 'R-SCH-153600-20' | 'R-SCH-307200-20'	'R-SCH-1350-40' | 'R-SCH-2400-40' | 'R-SCH-4800-40' | 'R-SCH-9600-40' | 'R-SCH-19200-40' | 'R-SCH-38400-40' | 'R-SCH-76800-40'| 'R-SCH-153600-40'	'R-SCH-1350-80' | 'R-SCH-2400-80' | 'R-SCH-4800-80' | 'R-SCH-9600-80' | 'R-SCH-19200-80' | 'R-SCH-38400-80' | 'R-SCH-76800-80'
5	'R-SCH-614400-20'	'R-SCH-307200-40'	'R-SCH-153600-80'
4 | 6	'R-SCH-1800-20' | 'R-SCH-3600-20' | 'R-SCH-7200-20' | 'R-SCH-14400-20' | 'R-SCH-28800-20' | 'R-SCH-57600-20' | 'R-SCH-115200-20' | 'R-SCH-230400-20'	'R-SCH-1800-40' | 'R-SCH-3600-40' | 'R-SCH-7200-40' | 'R-SCH-14400-40' | 'R-SCH-28800-40' | 'R-SCH-57600-40' | 'R-SCH-115200-40'	'R-SCH-1800-80' | 'R-SCH-3600-80' | 'R-SCH-7200-80' | 'R-SCH-14400-80' | 'R-SCH-28800-80' | 'R-SCH-57600-80'
6	'R-SCH-460800-20' | 'R-SCH-1036800-20'	'R-SCH-230400-40' | 'R-SCH-518400-40'	'R-SCH-115200-80' | 'R-SCH-259200-80'

Additional data rate information for the cdma2000 reverse links is given in Tables 2.1.3.1.3-1 and 2.1.3.1.3-2 of [1].

Example: `'R-SCH-153600-20'` is an R-SCH having a 153,600 bps data rate and a 20 ms frame length.

Data Types: `char`

Output Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
RadioConfiguration	'RC1' | 'RC2' | 'RC3' | 'RC4' | 'RC5' | 'RC6'	Radio configuration of the reverse channel. The spreading rate of the waveform is derived from the radio configuration. Spreading rate 1, SR1, corresponds to a 1.2288 Mcps carrier and is associated with RC1 through RC4. Spreading rate 3, SR3, corresponds to a 3.6864 Mcps carrier and is associated with RC5 and RC6.
PowerNormalization	'Off' | 'NormalizeTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients used only when the `FilterType` field is set to `'Custom'`
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when `EnableModulation` is `'On'`)
RPICH	Structure	See **RPICH Substructure**. Optional.
RACH	Structure	See **RACH Substructure**. Optional.

Parameter Field	Values	Description
REACH	Structure	See **REACH Substructure**. Optional.
RCCCH	Structure	See **RCCCH Substructure**. Optional.
RDCCH	Structure	See **RDCCH Substructure**. Optional.
RFCH	Structure	See **RFCH Substructure**. Optional.
RSCCH	Structure	See **RSCCH Substructure**. Optional.
RSCH1	Structure	See **RSCH1 Substructure**. Optional.
RSCH2	Structure	See **RSCH2 Substructure**. Optional.

RPICH Substructure

Include the `RPICH` substructure in the `cfg` structure to configure the Reverse Pilot Channel (R-PICH). The `RPICH` substructure contains the following fields.

Parameter Field	Values	Description
Enable	`'On'` | `'Off'`	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
PowerControlEnable	`'On'` | `'Off'`	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to R-PICH)
PowerControlDataSource	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Power control subchannel data source

RACH Substructure

Include the `RACH` substructure in the `cfg` structure to configure the Reverse Access Channel (R-ACH). The `RACH` substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

REACH Substructure

Include the REACH substructure in the cfg structure to configure the Reverse Enhanced Access Channel (R-EACH). The REACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

RCCCH Substructure

Include the RCCCH substructure in the cfg structure to configure the Reverse Common Control Channel (R-CCCH). The RCCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error control coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RDCCH Substructure

Include the RDCCH substructure in the cfg structure to configure the Reverse Dedicated Control Channel (R-DCCH). The RDCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
FrameLength	5 | 20	Frame length (ms)

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RFCH Substructure

Include the RFCH substructure in the cfg structure to configure the Reverse Fundamental Traffic Channel (R-FCH). The RFCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCCH Substructure

Include the `RSCCH` substructure in the `cfg` structure to configure the Reverse Supplemental Code Channel (R-SCCH). The `RSCCH` substructure contains the following fields.

Parameter Field	Values	Description
Enable	`'On'` | `'Off'`	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	`'On'` | `'Off'`	Enable or disable channel coding
DataSource	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH1 Substructure

Include the `RSCH1` substructure in the `cfg` structure to configure the Reverse Supplemental Channel 1 (R-SCH 1). The `RSCH1` substructure contains the following fields.

Parameter Field	Values	Description
Enable	`'On'` | `'Off'`	Enable or disable the channel

Parameter Field	Values	Description
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	2 | 4	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH2 Substructure

Include the RSCH2 substructure in the cfg structure to configure the Reverse Supplemental Channel 2 (R-SCH 2). The RSCH2 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel

Parameter Field	Values	Description
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	4 | 8	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

Data Types: struct

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

[2] 3GPP2 C.S0011–E v2.0. “Recommended Minimum Performance Standards for cdma2000 Spread Spectrum Mobile Stations.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`cdma2000ForwardReferenceChannels` | `cdma2000ReverseWaveformGenerator`

Introduced in R2015b

cdma2000ReverseWaveformGenerator

Generate cdma2000 reverse link waveform

Syntax

```
waveform = cdma2000ReverseWaveformGenerator(cfg)
```

Description

`waveform = cdma2000ReverseWaveformGenerator(cfg)` returns the cdma2000 reverse link baseband waveform, `waveform` as defined by the parameter configuration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties used by the function to generate a cdma2000 waveform. You can generate the input argument by using the `cdma2000ReverseReferenceChannels` function. The top-level parameters of `cfg` are `RadioConfiguration`, `LongCodeState`, `PowerNormalization`, `OversamplingRatio`, `FilterType`, `InvertQ`, `EnableModulation`, `ModulationFrequency`, and `NumChips`. To enable specific channels, add their associated substructures, for example, the reverse dedicated control channel, `RDCCH`.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all combinations of spreading rate, radio configuration, frame length, and data rate are supported. To ensure that the input argument is valid, use the `cdma2000ReverseReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate Reverse Common Control Channel Waveform

Generate the structure corresponding to the reverse common control channel (R-CCCH) having a 19,200 bps data rate and 10 ms frames.

```
config = cdma2000ReverseReferenceChannels('R-CCCH-19200-10');
```

Verify that the R-CCCH substructure is configured for the correct data rate and frame duration.

```
config.RCCCH
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
     LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 19200
     FrameLength: 10
       WalshCode: 1
```

Generate the reverse channel waveform using the corresponding waveform generator function, `cdma2000ReverseWaveformGenerator`.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate R-SCH Channels for RC5

Create a configuration structure for a reverse channel having an R-FCH with a 4800 bps data rate and two R-SCHs. Specify that each R-SCH have a 153,600 bps data rate using RC5.

```
config = cdma2000ReverseReferenceChannels('TRAFFIC-RC5-4800',5000, ...
    'R-SCH-153600-40');
```

Determine the sample rate. Because RC5 corresponds to SR3, the chip rate is 3.6864 Mcps. Multiply by the oversampling ratio to obtain the sample rate.

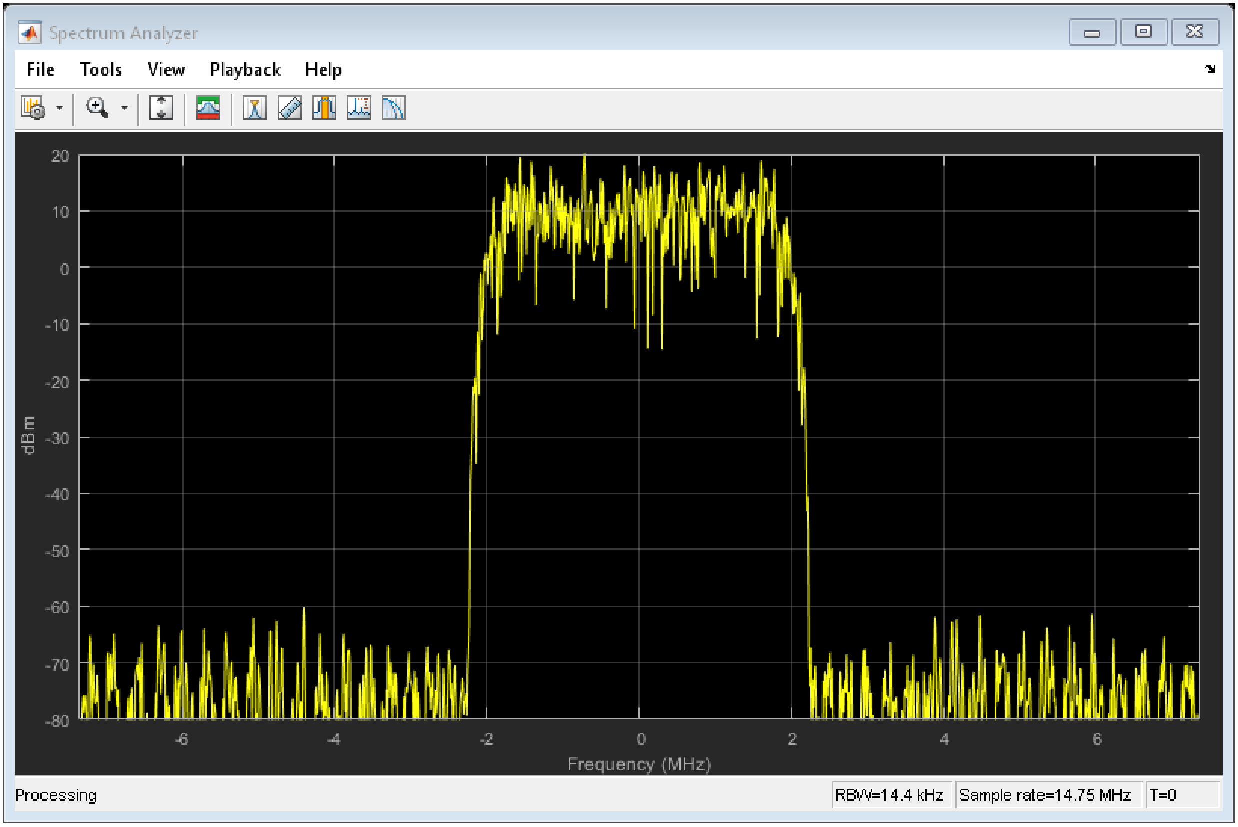
```
fs = 3.6864e6*config.OversamplingRatio;
```

Generate the reverse link waveform.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Plot the spectrum of the resultant waveform.

```
sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Processing

RBW=14.4 kHz | Sample rate=14.75 MHz | T=0

Generate cdma2000 Waveform with Two Reverse Supplemental Channels

Create a parameter structure specifying a reverse traffic channel containing a pair of supplemental channels and generate the corresponding waveform.

Create a parameter structure specifying a traffic channel consisting of a 14,400 bps fundamental channel, 2000 chips, and a 57,600 bps supplemental channel (R-SCH) pair having a 40 ms frame duration.

```
cfg = cdma2000ReverseReferenceChannels('TRAFFIC-RC4-14400',2000,'F-SCH-57600-40');
```

Create a second R-SCH pair by copying the R-SCH fields from the existing pair.

```
cfg(2).RSCH1 = cfg.RSCH1;
cfg(2).RSCH2 = cfg.RSCH2;
```

Set the data rate of the second R-SCH pair to 28,800 bps.

```
cfg(2).RSCH1.DataRate = 28800;
cfg(2).RSCH2.DataRate = 28800;
```

Set the Walsh codes of the second pair so that they differ from the first pair.

```
cfg(2).RSCH1.WalshCode = 4;
cfg(2).RSCH2.WalshCode = 5;
```

Verify that the data rates are set correctly and that no two supplemental channels share the same Walsh code.

```
cfg.RSCH1
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 57600
     FrameLength: 40
     WalshLength: 2
       WalshCode: 0


ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 28800
     FrameLength: 40
     WalshLength: 2
       WalshCode: 4
```

```
cfg.RSCH2
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 57600
     FrameLength: 40
     WalshLength: 2
       WalshCode: 1


ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 28800
     FrameLength: 40
     WalshLength: 2
       WalshCode: 5
```

Generate the reverse link waveform.

```
wv = cdma2000ReverseWaveformGenerator(cfg);
```

Input Arguments

cfg — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
RadioConfiguration	'RC1' | 'RC2' | 'RC3' | 'RC4' | 'RC5' | 'RC6'	Radio configuration of the reverse channel. The spreading rate of the waveform is derived from the radio configuration. Spreading rate 1, SR1, corresponds to a 1.2288 Mcps carrier and is associated with RC1 through RC4. Spreading rate 3, SR3, corresponds to a 3.6864 Mcps carrier and is associated with RC5 and RC6.
PowerNormalization	'Off' | 'NormalizeTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients used only when the FilterType field is set to 'Custom'
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
RPICH	Structure	See **RPICH Substructure**. Optional.
RACH	Structure	See **RACH Substructure**. Optional.
REACH	Structure	See **REACH Substructure**. Optional.
RCCCH	Structure	See **RCCCH Substructure**. Optional.
RDCCH	Structure	See **RDCCH Substructure**. Optional.
RFCH	Structure	See **RFCH Substructure**. Optional.
RSCCH	Structure	See **RSCCH Substructure**. Optional.
RSCH1	Structure	See **RSCH1 Substructure**. Optional.
RSCH2	Structure	See **RSCH2 Substructure**. Optional.

RPICH Substructure

Include the RPICH substructure in the cfg structure to configure the Reverse Pilot Channel (R-PICH). The RPICH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to R-PICH)
PowerControlDataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

RACH Substructure

Include the RACH substructure in the cfg structure to configure the Reverse Access Channel (R-ACH). The RACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

REACH Substructure

Include the REACH substructure in the cfg structure to configure the Reverse Enhanced Access Channel (R-EACH). The REACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

RCCCH Substructure

Include the RCCCH substructure in the cfg structure to configure the Reverse Common Control Channel (R-CCCH). The RCCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error control coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RDCCH Substructure

Include the RDCCH substructure in the cfg structure to configure the Reverse Dedicated Control Channel (R-DCCH). The RDCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RFCH Substructure

Include the `RFCH` substructure in the `cfg` structure to configure the Reverse Fundamental Traffic Channel (R-FCH). The `RFCH` substructure contains the following fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`DataRate`	`1200` | `1500` | `1800` | `2400` | `2700` | `3600` | `4800` | `7200` | `9600` | `14400`	Data rate (bps)
`FrameLength`	`5` | `10` | `20`	Frame length (ms)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 15	Walsh code number
`LongCodeMask`	42-bit binary number	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCCH Substructure

Include the RSCCH substructure in the cfg structure to configure the Reverse Supplemental Code Channel (R-SCCH). The RSCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH1 Substructure

Include the RSCH1 substructure in the cfg structure to configure the Reverse Supplemental Channel 1 (R-SCH 1). The RSCH1 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	2 | 4	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH2 Substructure

Include the RSCH2 substructure in the cfg structure to configure the Reverse Supplemental Channel 2 (R-SCH 2). The RSCH2 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	4 | 8	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

Output Arguments

waveform — Modulated baseband waveform comprising the physical channels
complex vector array

Modulated baseband waveform comprising the cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`cdma2000ForwardWaveformGenerator` | `cdma2000ReverseReferenceChannels`

Introduced in R2015b

cma

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

Examples

Create a Linear Equalizer using CMA

Use the constant modulus algorithm (CMA) to create an adaptive equalizer object.

Set the number of weights and the step size for the equalizer.

```
nWeights = 1;
stepSize = 0.1;
```

Create an adaptive algorithm object using the `cma` function.

```
alg = cma(stepSize);
```

Construct a linear equalizer using the algorithm object.

```
eqObj = lineareq(nWeights,alg)
```

```
eqObj =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Constant Modulus'
                nWeights: 1
             nSampPerSym: 1
                SigConst: [-1 1]
                StepSize: 0.1000
           LeakageFactor: 1
                 Weights: 0
            WeightInputs: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```

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

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

Topics

“Equalization”

Introduced before R2006a

comm_links

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`

Introduced before R2006a

commlib

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`

Introduced before R2006a

commscope

(To be removed) Package of communications scope classes

Note `commscope.ScatterPlot` has been removed. Use `comm.ConstellationDiagram` instead.

`commscope.eyediagram` will be removed in a future release. Use `comm.EyeDiagram` instead.

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, for example `help commscope.eyediagram`.

See Also

`comm.ConstellationDiagram` | `comm.EyeDiagram`

Introduced in R2007b

commscope.eyediagram

(To be removed) Eye diagram analysis

Note `commscope.eyediagram` will be removed in a future release. Use `comm.EyeDiagram` instead.

Syntax

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

Description

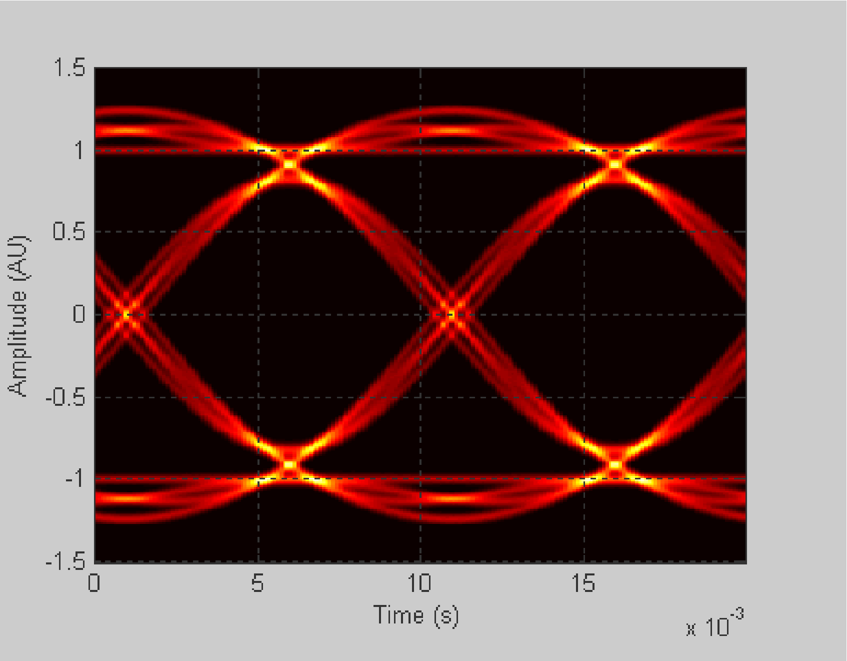
`h = commscope.eyediagram` constructs an eye diagram object, `h`, with default properties. This syntax is equivalent to:

```
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');
```

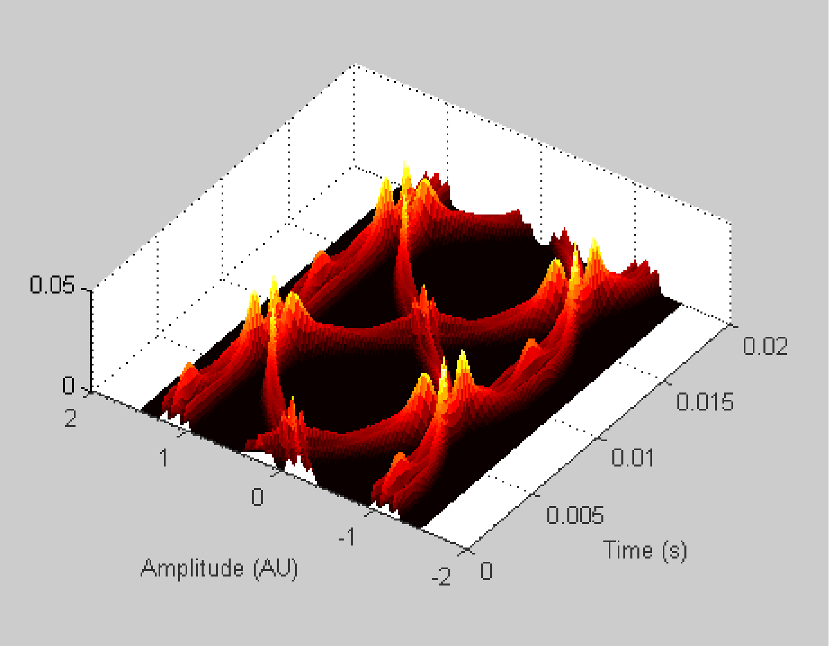
`h = commscope.eyediagram(property1,value1,...)` constructs an eye diagram object, `h`, with properties as specified by property/value pairs.

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.

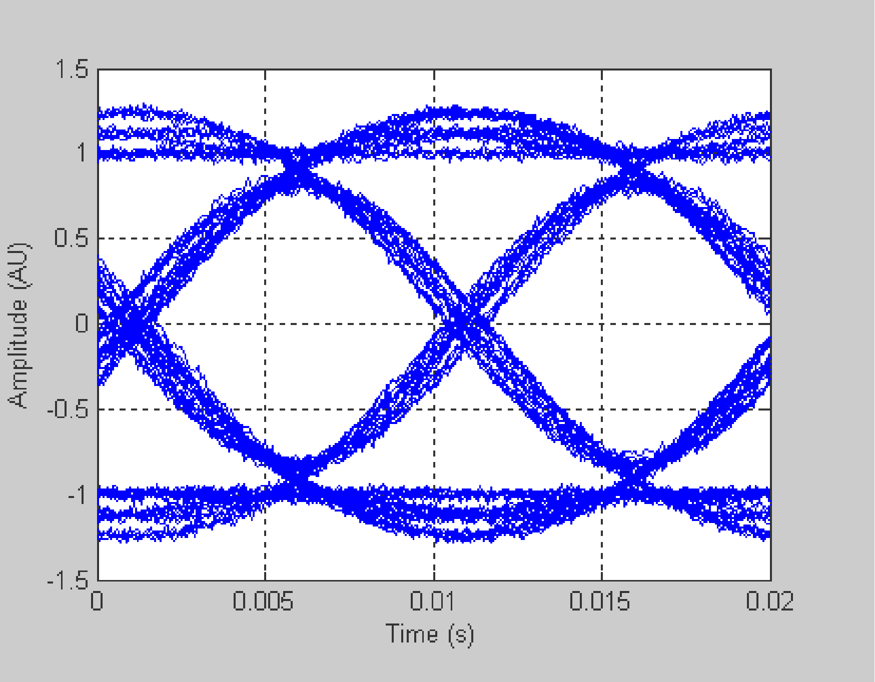
You can change the plot type by setting the `PlotType` property. The following plots are examples of each type.



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.
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.

Property	Description
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).
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 1-238 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:

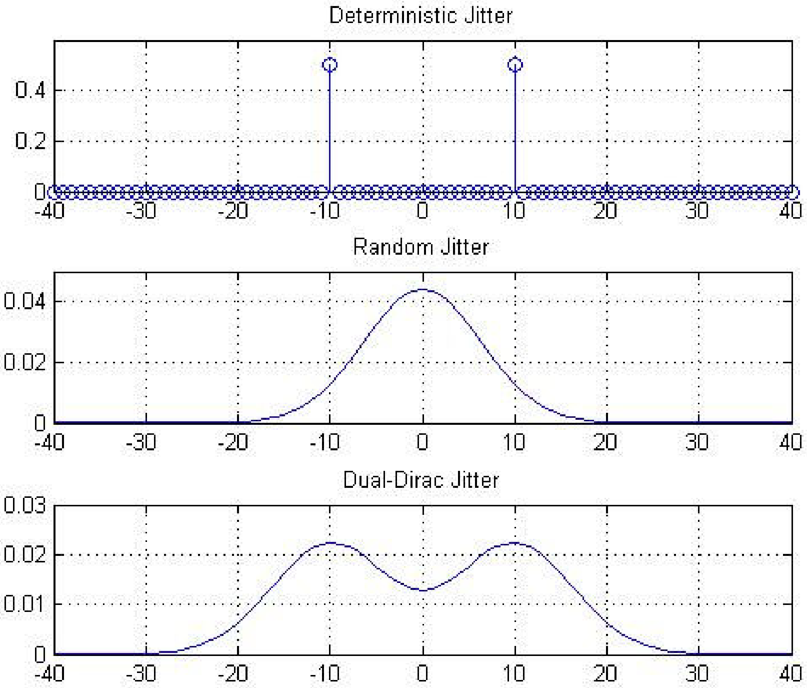
- Amplitude Measurements
  - Eye Amplitude
  - Eye Crossing Amplitude
  - Eye Crossing Percentage
  - Eye Height
  - Eye Level
  - Eye SNR
  - Quality Factor
  - Vertical Eye Opening
- Time Measurements
  - Deterministic Jitter
  - Eye Crossing Time
  - Eye Delay
  - Eye Fall Time

- 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-Dirac 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-Dirac model. A dual-Dirac 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.



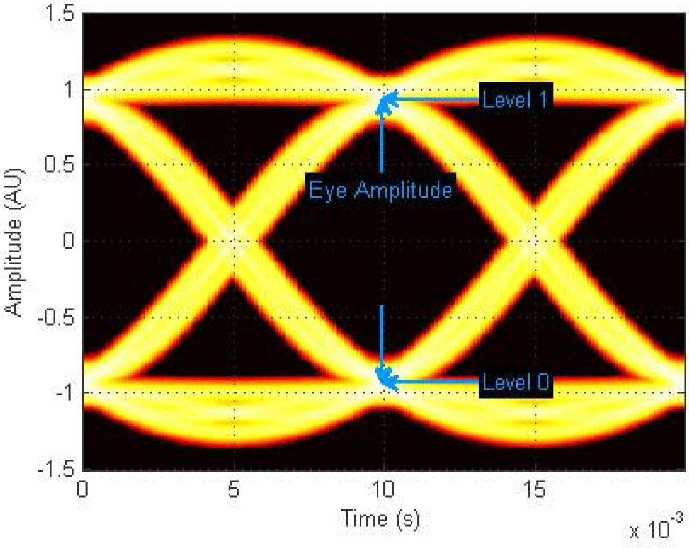
The dual-Dirac model is described in [5] in more detail. The amplitude of the two Dirac functions may not be the same. In such a case, the analyze method estimates these amplitudes, ρ_L and ρ_R.

Amplitude Measurements

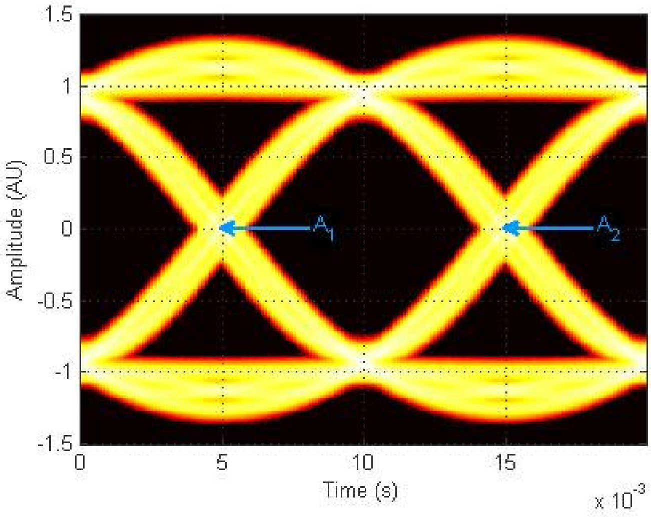
You can use the vertical histogram to obtain a variety of amplitude measurements. For complex signals, measurements are done on both in-phase and the quadrature components, unless otherwise specified.

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

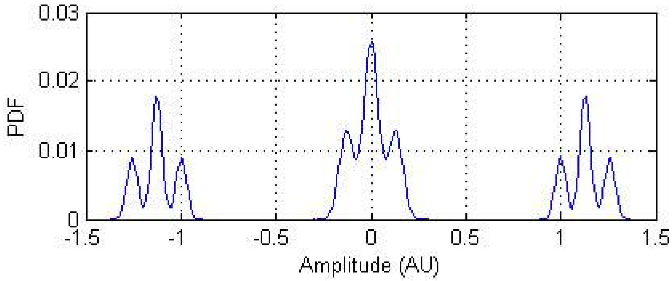
Eye Amplitude, measured in Amplitude Units (AU), is defined as the distance between two neighboring eye levels. For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye amplitude is the difference of these two values, as shown in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



Eye crossing 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.



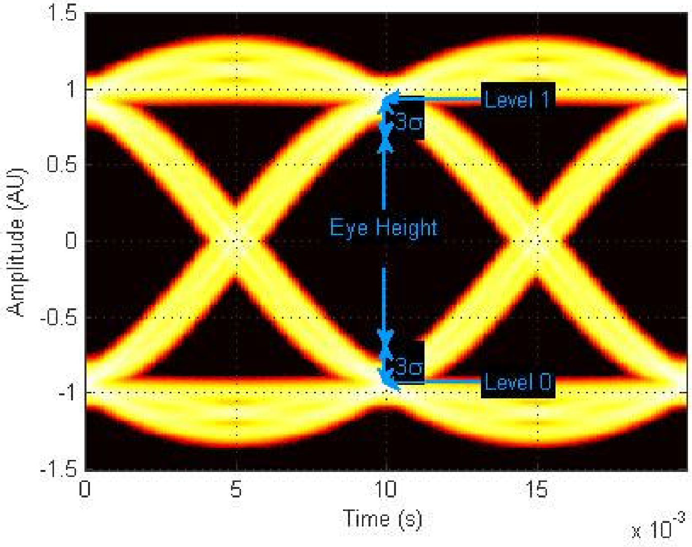
The next figure shows the vertical histogram at the first eye crossing time.



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

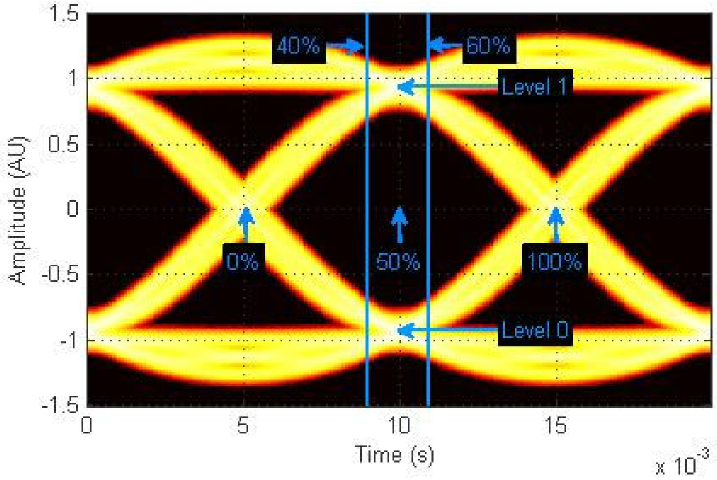
Eye Height, measured in Amplitude Units (AU), is defined as the 3σ distance between two neighboring eye levels.

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.

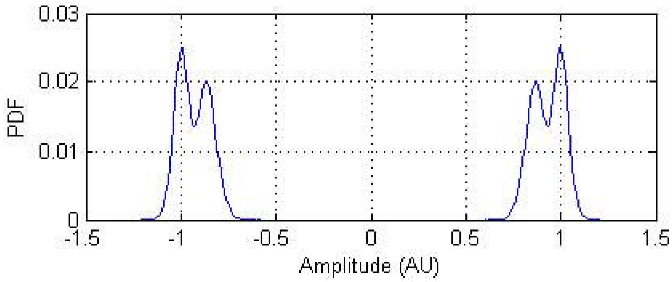


Eye Level is the amplitude level used to represent data bits, measured in Amplitude Units (AU).

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 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:

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

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.

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.

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 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]:

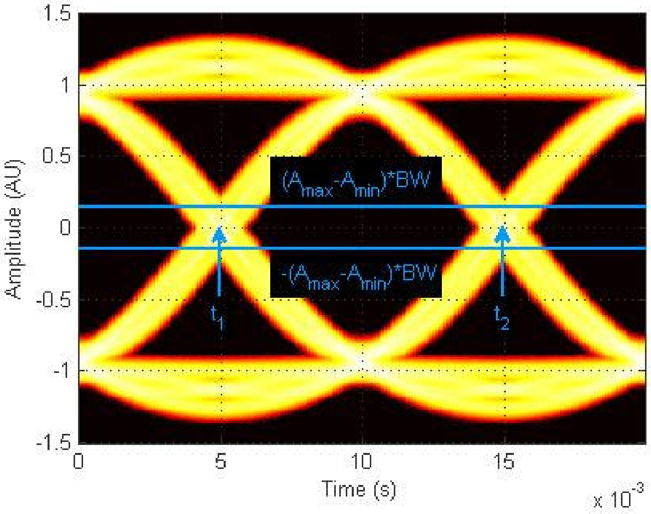
$$DJ = \mu_L - \mu_R$$

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

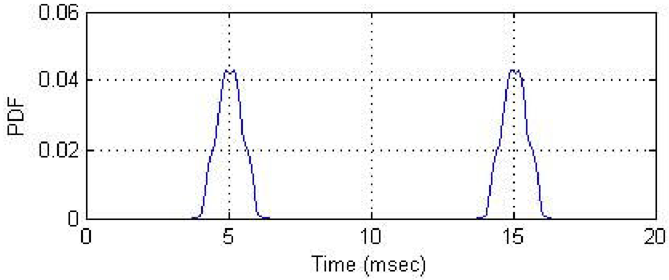
Eye crossing times are calculated as the mean of the horizontal histogram for each crossing point, around the reference amplitude level. This value is measured in seconds.

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., $A_{\text{total}} = A_{\text{max}} — A_{\text{min}}$) and BW is the crossing band width, shown in the following figure.



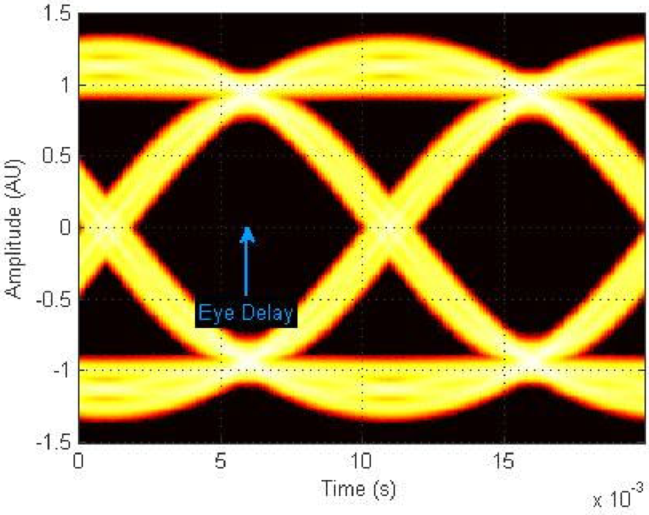
The following figure shows the average PDF in this region. Because this example assumes two symbols per trace, there are two crossing points.



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).

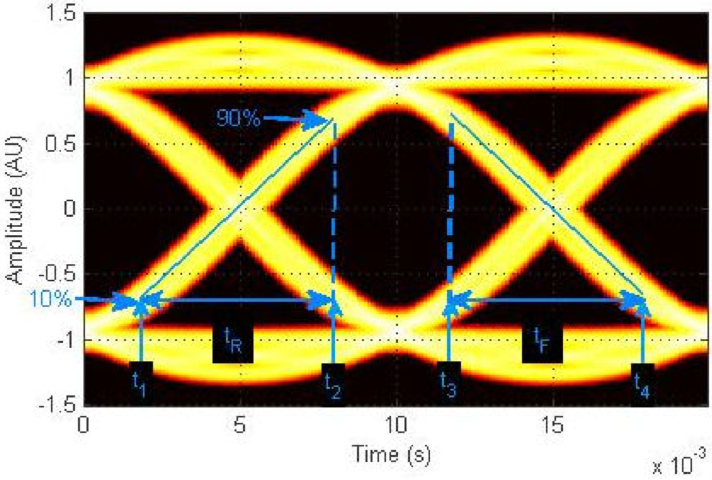
To avoid the time-wrapping or a warning, add a half-symbol duration delay to the current value in the MeasurementDelay property of the eye diagram object. This additional delay repositions the eye in the approximate center of the scope.

Eye Delay 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.

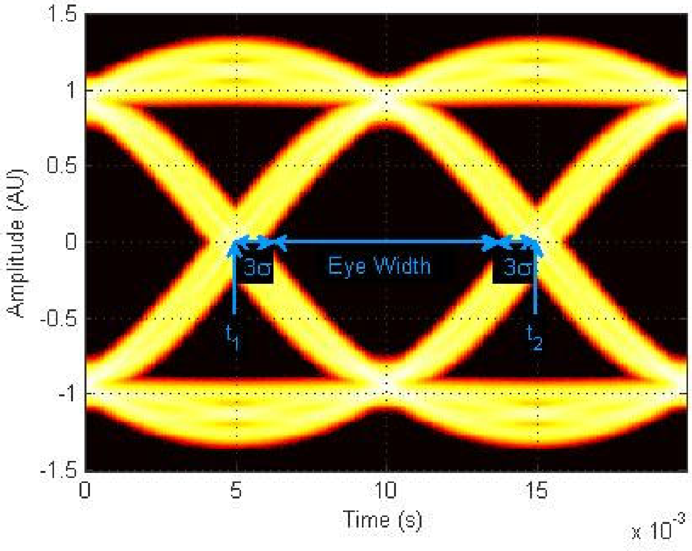


Eye Fall Time is the mean time between the high and low threshold values defined by the AmplitudeThreshold property of the eyemeasurementsetup object. The previous figure shows the fall time calculated from 10% to 90% of the eye amplitude.

Eye Rise Time 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 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 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 is the difference between the extreme data points of the histogram.

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 eyemeasurementsetup object.

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

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.

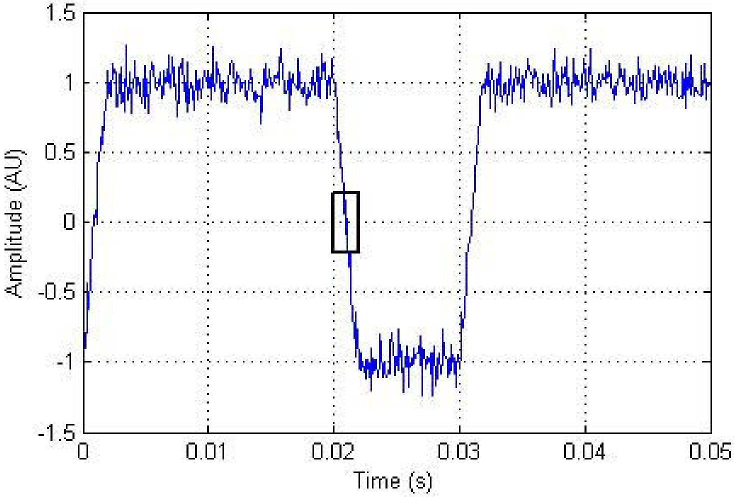
Amplitude Threshold

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.

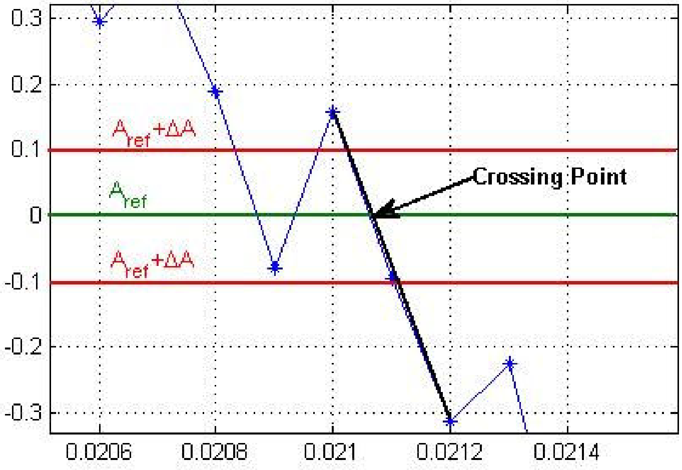
Jitter Hysteresis

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

`comm.EyeDiagram`

Introduced in R2007b

commscope.ScatterPlot

(Removed) Create Scatter Plot scope

Note `commscope.ScatterPlot` has been removed. Use `comm.ConstellationDiagram` instead.

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.

Property	Description
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.
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.

Property	Description
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 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).

See Also

`comm.ConstellationDiagram` | `scatterplot`

Topics

“Scatter Plots and Constellation Diagrams”

Introduced in R2009a

commsrc.combinedjitter

Construct combined jitter generator object

Syntax

```
combJitt = commsrc.combinedjitter
combJitt = commsrc.combinedjitter(Name,Value)
```

Description

`combJitt = commsrc.combinedjitter` constructs a default combined jitter generator object, `combJitt`, with with all jitter components disabled.

Use the object to generate jitter samples that include any combination of random, periodic, and Dirac components.

`combJitt = commsrc.combinedjitter(Name,Value)` creates a combined jitter generator object 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

A combined jitter generator object includes these properties. You can edit all properties, except those explicitly noted.

Property	Description
`Type`	Type of object, `Combined Jitter Generator`. This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz. Default value 1e4.

Property	Description
`RandomJitter`	Variable to enable the random jitter generator. Specify as either `'off'` (default) or `'on'`.
`RandomStd`	Standard deviation of the random jitter generator in seconds. Applies when `RandomJitter` is `'on'`. Default value 1e-4.
`PeriodicJitter`	Variable to enable the periodic jitter generator. Specify as either `'off'` (default) or `'on'`.
`PeriodicNumber`	Number of sinusoidal components. The `PeriodicNumber` must be a finite positive scalar integer. Applies when `PeriodicJitter` is `'on'`. Default value 1.
`PeriodicAmplitude`	Amplitude of each sinusoidal component of the periodic jitter in seconds. Applies when `PeriodicJitter` is `'on'`. Default value 5e-4.
`PeriodicFrequencyHz`	Frequency of each sinusoidal component of the periodic jitter measured in Hz. Applies when `PeriodicJitter` is `'on'`. Default value is 200.
`PeriodicPhase`	Phase of each sinusoidal component of the periodic jitter in radians. Applies when `PeriodicJitter` is `'on'`. Default value 0.
`DiracJitter`	Variable to enable the Dirac jitter generator. Specify as either `'off'` (default) or `'on'`.
`DiracNumber`	Number of Dirac components. The `DiracNumber` must be a finite positive scalar integer. Applies when `DiracJitter` is `'on'`. Default value 2.
`DiracDelta`	Time delay of each Dirac component in seconds. Applies when `DiracJitter` is `'on'`. Default value [-5.e-4 5.e-4].
`DiracProbability`	Probability of each Dirac component represented as a vector of length `DiracNumber`. The sum of the probabilities must equal one. Applies when `DiracJitter` is `'on'`. Default value [0.5 0.5].

Object Functions

A combined jitter generator object has three object functions, as described in this section.

generate

This object function generates jitter samples based on the jitter generator object. It has one input argument, which is the number of samples in a frame. Its output is a single-column vector of length N. You can call this object function using this syntax:

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

where combJitt is the generator object, N is the number of output samples, and x is a real single-column vector.

reset

This object function resets the internal states of the combined jitter generator. You can call this object function using this syntax:

```
reset(combJitt)
```

where combJitt is the generator object.

disp

Display the properties of the combined generator object, combJitt. You can call this object function using this syntax:

```
disp(combJitt)
```

where combJitt is the generator object.

Examples

Generate Combined Random and Periodic Jitter

Generate 500 jitter samples composed of random and periodic components.

Create a `commsrc.combinedjitter` object configured to apply a combination of random and periodic jitter components. Use name-value pairs to enable `RandomJitter` and `PeriodicJitter, and to assign jitter settings.` Set the standard deviation of the random jitter to `2e-4 seconds,` the periodic jitter amplitude to 5e-4 seconds, and the periodic jitter frequency to `2 Hz`.

```
numSamples = 500;
combJitt = commsrc.combinedjitter(...
    'RandomJitter','on', ...
    'RandomStd',2e-4, ...
    'PeriodicJitter','on', ...
    'PeriodicAmplitude',5e-4, ...
    'PeriodicFrequencyHz',200)
```

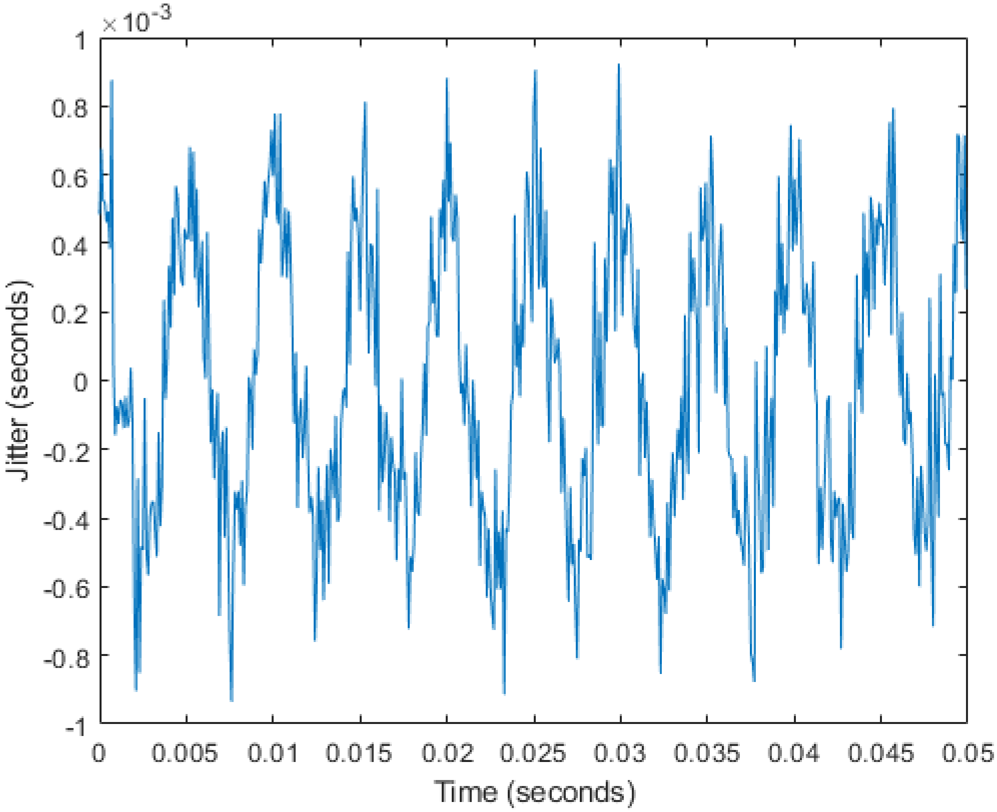
```
combJitt =
                   Type: 'Combined Jitter Generator'
      SamplingFrequency: 10000
           RandomJitter: 'on'
              RandomStd: 2.0000e-04
         PeriodicJitter: 'on'
         PeriodicNumber: 1
      PeriodicAmplitude: 5.0000e-04
    PeriodicFrequencyHz: 200
          PeriodicPhase: 0
            DiracJitter: 'off'
```

Use the `generate` method to create the combined jitter samples.

```
y = generate(combJitt,numSamples);
x = [0:numSamples-1];
```

Plot the jitter samples. You can see the Gaussian and periodic nature of the combined jitter.

```
plot(x/combJitt.SamplingFrequency,y)
xlabel('Time (seconds)')
ylabel('Jitter (seconds)')
```



Display commsrc.combinedjitter Object Settings

Create a `commsrc.combinedjitter` object. Display the default object property values.

```
combJitt = commsrc.combinedjitter;
disp(combJitt)
```

```
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Create a `commsrc.combinedjitter` object with random, periodic, and Dirac jitters enabled. Display the object property values.

```
combJitt = commsrc.combinedjitter('RandomJitter','on', ...
    'PeriodicJitter','on','DiracJitter','on');
disp(combJitt)
```

```
                   Type: 'Combined Jitter Generator'
      SamplingFrequency: 10000
           RandomJitter: 'on'
              RandomStd: 1.0000e-04
         PeriodicJitter: 'on'
         PeriodicNumber: 1
      PeriodicAmplitude: 5.0000e-04
    PeriodicFrequencyHz: 200
          PeriodicPhase: 0
            DiracJitter: 'on'
            DiracNumber: 2
             DiracDelta: [-5.0000e-04 5.0000e-04]
       DiracProbability: [0.5000 0.5000]
```

Generate Non-return-to-zero Pattern Signal

Generate a binary non-return-to-zero (NRZ) signal utilizing the pattern generator object. View the NRZ signal with and without jitter applied to the signal.

Initialize system parameters.

```
Fs = 10000;            % Sample rate
Rs = 50;               % Symbol rate (Sps)
sps = Fs/Rs;           % Number of samples per symbol
Trise = 1/(5*Rs);      % Rise time of the NRZ signal
Tfall = 1/(5*Rs);      % Fall time of the NRZ signal
frameLen = 100;        % Number of symbols in a frame
spt = 200;               % Number of samples per trace on eye diagram
```

Create a pattern generator object with no jitter component assigned.

```
src = commsrc.pattern('SamplingFrequency', Fs, ...
                      'SamplesPerSymbol', sps, ...
                      'RiseTime', Trise, ...
                      'FallTime', Tfall)
```

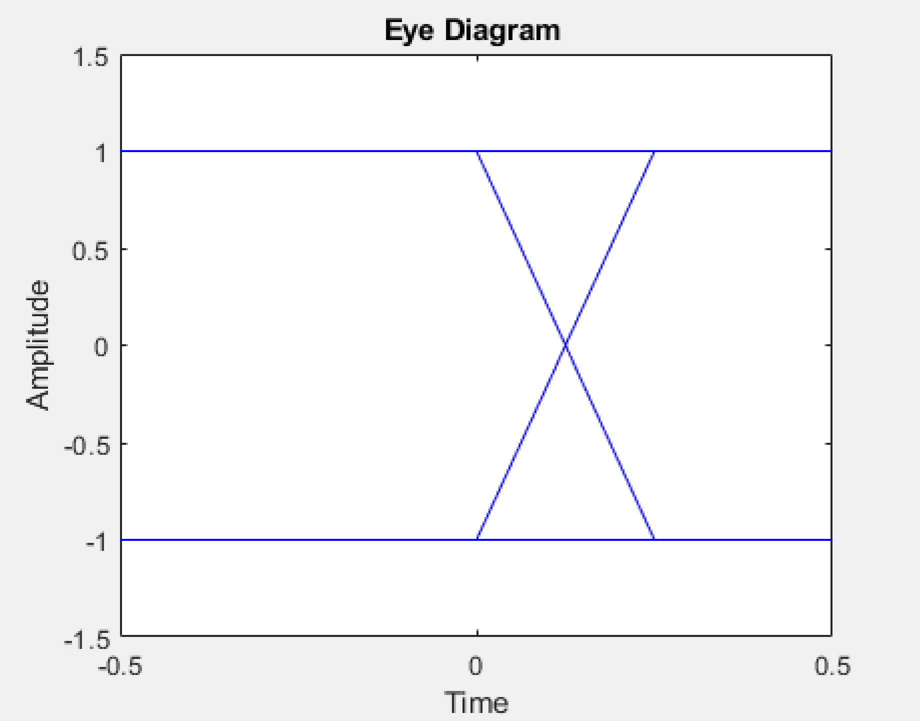
```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

```
src.Jitter
```

```
ans =
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Generate an NRZ signal and view the eye diagram of the signal.

```
message = generate(src,frameLen);
eyediagram(message,spt)
```



Add inter-symbol-interference (ISI) to an NRZ signal. ISI is modeled by two equal amplitude Dirac functions. Create a combined jitter object with Dirac jitter and assign it to the pattern generator object.

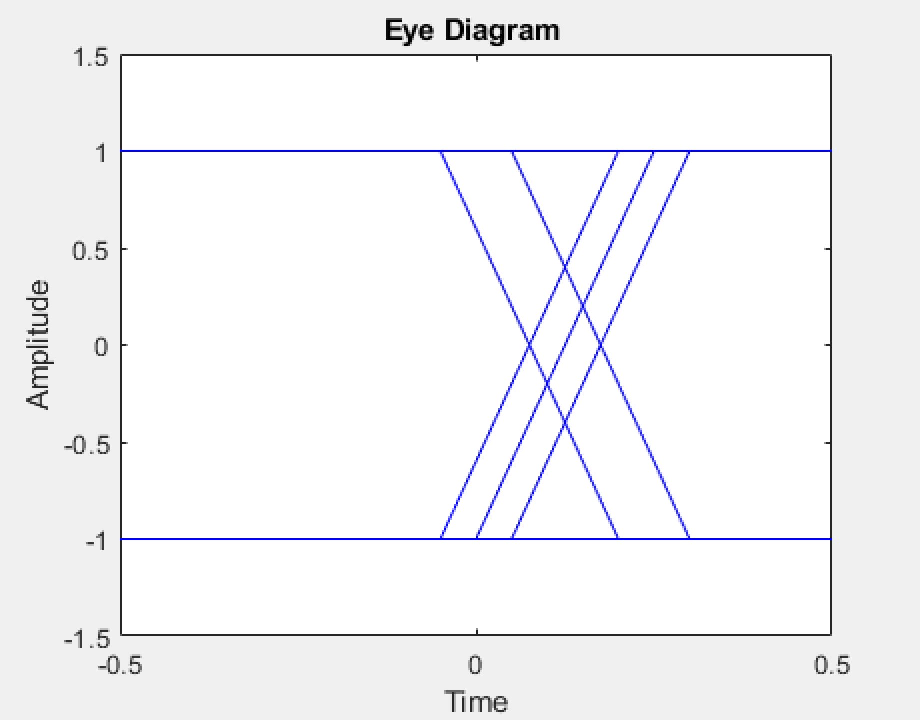
```
jitterSrc = commsrc.combinedjitter('DiracJitter','on', ...
    'DiracDelta',0.05/Rs*[-1 1]);
src.Jitter = jitterSrc
```

```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
```

```
              Jitter: [1x1 commsrc.combinedjitter]
```

Generate an NRZ signal that has jitter added to it and view the eye diagram of the signal.

```
reset(src);
message = generate(src, frameLen);
eyediagram(message,spt)
```



- “Eye Diagram Measurements”

See Also

Functions
`commsrc.pattern`

Topics

"Eye Diagram Measurements"

Introduced in R2015a

commsrc.pattern

Construct pattern generator object

Syntax

```
h = commsrc.pattern
h = commsrc.pattern(Name,Value)
```

Description

`h = commsrc.pattern` constructs a pattern generator object, `h`.

The pattern generator object produces modulated data patterns. The object can be used to inject jitter into modulated signals.

`h = commsrc.pattern(Name,Value)` creates a combined jitter generator object 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

A pattern generator object includes these properties. You can edit all properties, except those explicitly noted.

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 non-return-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, specify as 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, specify as 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 “Object Functions” on page 1-271. Applies when PulseType is 'RZ'.
FallTime	Fall time of the pulse in seconds, specified as a percentage from 10 to 90.
DataPattern	The bit sequence the object uses, specified as 'PRBS5', 'PRBS6', ..., 'PRBS15', 'PRBS23', 'PRBS31', and'User Defined'.
UserDataPattern	User-defined bit pattern consisting of a vector of ones and zeroes. Applies when DataPattern is 'User Defined'.

Property	Description
`Jitter`	Jitter characteristics, specified as a `commsrc.combinedjitter` object. Use this property to configure Random, Periodic and Dual Dirac Jitter.

Object Functions

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

generate

This object function 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 column vector. You can call the object function using this syntax:

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

where `h` is the handle to the object, `N` is the number of output symbols, and `x` is a column vector whose length is `N` multiplied by `h.SamplesPerSymbol`.

reset

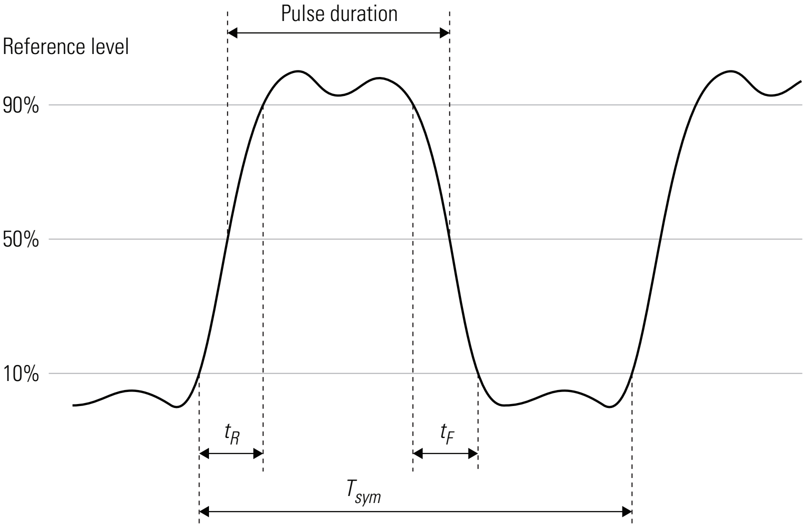
This object function resets the pattern generator to its default state. The property values do not reset unless they relate to the state of the object. This object function has no input arguments.

idealtostd181

This object function converts the ideal pulse specifications to IEEE STD-181 specifications. The ideal 0% to 100% span rise time (`tr`) and fall time (`tf`) are converted to 10% to 90% spans with a 50% pulse width duration (`pw`). Call the `idealtostd181` object function using this syntax:

```
h = stdstd181toideal(tr,tf,pw)
```

The object function sets the appropriate properties. The IEEE STD-181 Return-to-Zero (RZ) signal parameters are shown in this figure.

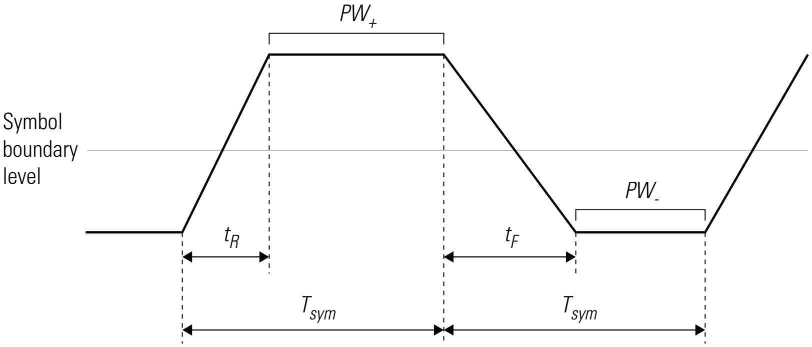


std181toideal

The `std181toideal` object function converts the IEEE STD-181 pulse specifications, stored in the pattern generator, to ideal pulse specifications. The function converts the rise and fall times from 10% – 90% span to 0% – 100% span, and converts the 50% pulse duration to pulse width. Call the `std181toideal` object function using this syntax:

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

where `h` is the pattern generator object handle, `tr` is the ideal 0% – 100% rise time, `tf` is the ideal 0% – 100% fall time, and `pw` is the ideal pulse width. The ideal pulse non-return-to-zero (NRZ) signal parameters are shown in this figure.



Use the property values for IEEE STD-181 specifications.

computedcd

The `computedcd` object function 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. Call the `computedcd` object function using this syntax:

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

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

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

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

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

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.

Examples

Display commsrc.pattern Object Settings

Create a `commsrc.pattern` object. Display the default object property values.

```
h = commsrc.pattern;
disp(h)
```

```
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
      SamplesPerSymbol: 100
           SymbolRate: 100
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0
             FallTime: 0
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

Generate Non-return-to-zero Pattern Signal

Generate a binary non-return-to-zero (NRZ) signal utilizing the pattern generator object. View the NRZ signal with and without jitter applied to the signal.

Initialize system parameters.

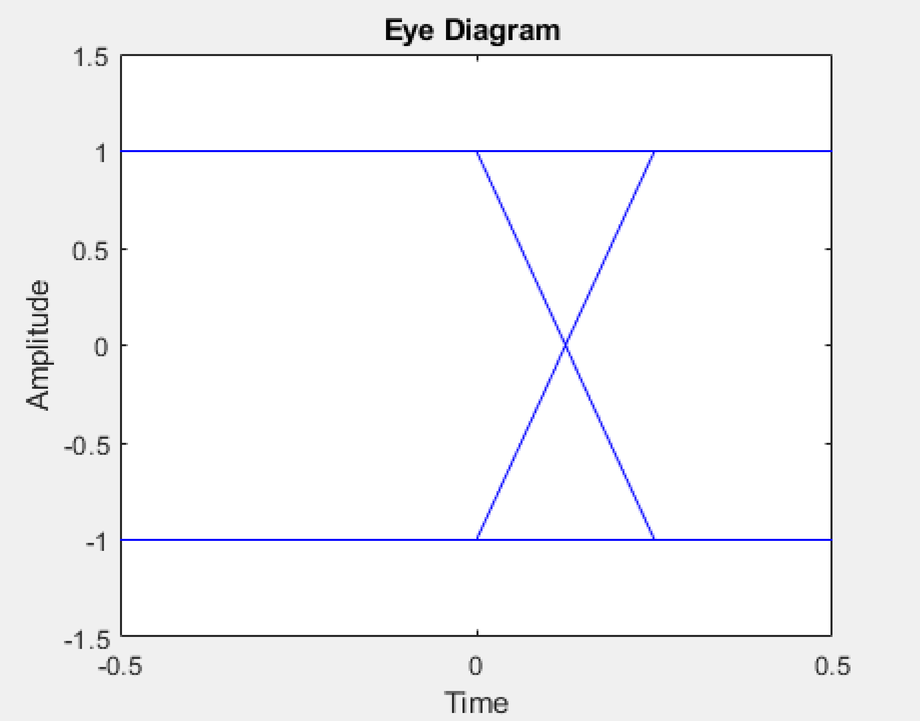
```
Fs = 10000;             % Sample rate
Rs = 50;                % Symbol rate (Sps)
sps = Fs/Rs;            % Number of samples per symbol
Trise = 1/(5*Rs);       % Rise time of the NRZ signal
Tfall = 1/(5*Rs);       % Fall time of the NRZ signal
frameLen = 100;         % Number of symbols in a frame
spt = 200;                % Number of samples per trace on eye diagram
```

Create a pattern generator object with no jitter component assigned.

```
src = commsrc.pattern('SamplingFrequency', Fs, ...
                      'SamplesPerSymbol', sps, ...
                      'RiseTime', Trise, ...
                      'FallTime', Tfall)

src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]


src.Jitter

ans =
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Generate an NRZ signal and view the eye diagram of the signal.

```
message = generate(src,frameLen);
eyediagram(message,spt)
```



Add inter-symbol-interference (ISI) to an NRZ signal. ISI is modeled by two equal amplitude Dirac functions. Create a combined jitter object with Dirac jitter and assign it to the pattern generator object.

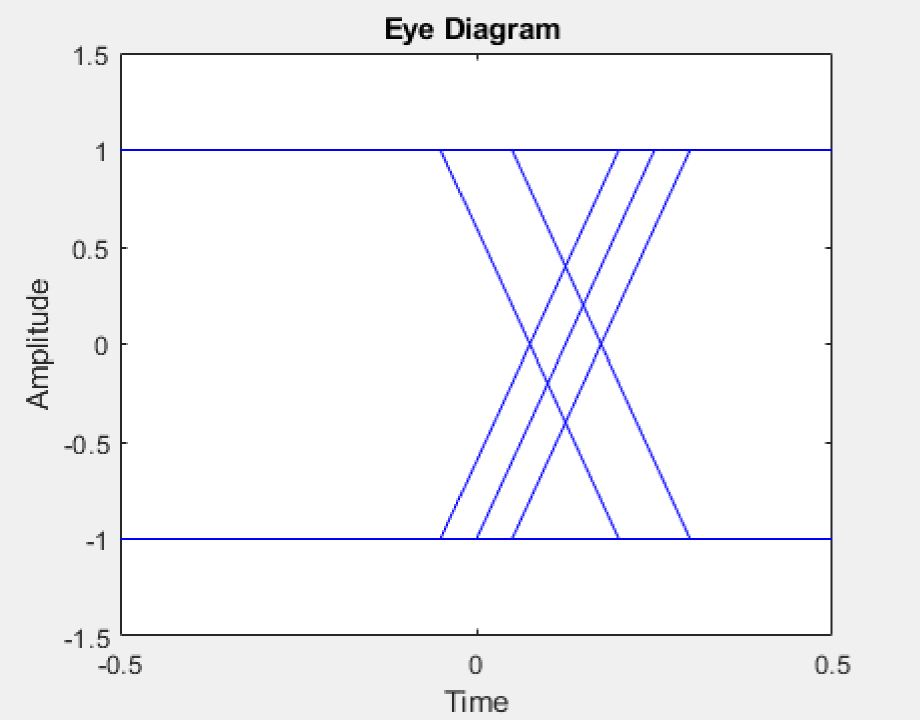
```
jitterSrc = commsrc.combinedjitter('DiracJitter','on', ...
    'DiracDelta',0.05/Rs*[-1 1]);
src.Jitter = jitterSrc
```

```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
```

```
                Jitter: [1x1 commsrc.combinedjitter]
```

Generate an NRZ signal that has jitter added to it and view the eye diagram of the signal.

```
reset(src);
message = generate(src, frameLen);
eyediagram(message,spt)
```



- “Eye Diagram Measurements”

References

[1] IEEE Standard for Transitions, Pulses, and Related Waveforms, STD-181-2011. Piscataway, NJ. 6 September 2011.

See Also

Functions
`commsrc.combinedjitter`

Topics

"Eye Diagram Measurements"

Introduced in R2008b

commsrc.pn

Create PN sequence generator object

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

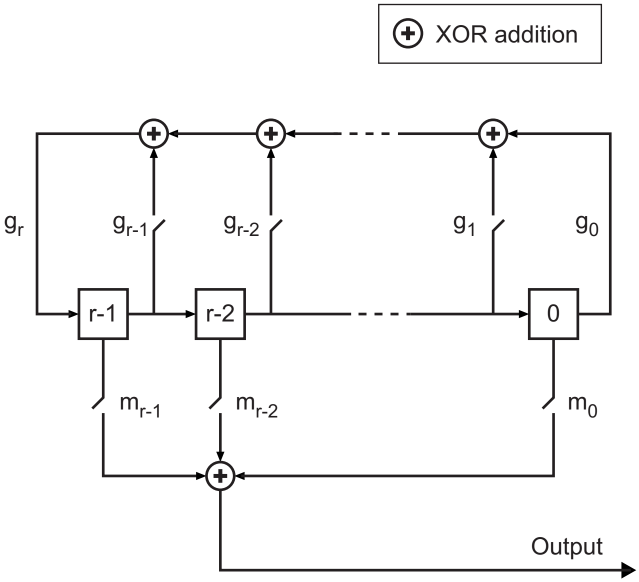
Every time this property is set, it will reset the entire object. In addition to changing the polynomial values, 'CurrentStates', 'InitialStates', and 'Mask' will be set to their default values ('NumBitsOut' will remain the same), and no warnings will be issued.

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*, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \ldots + g_0$. 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 `'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]`

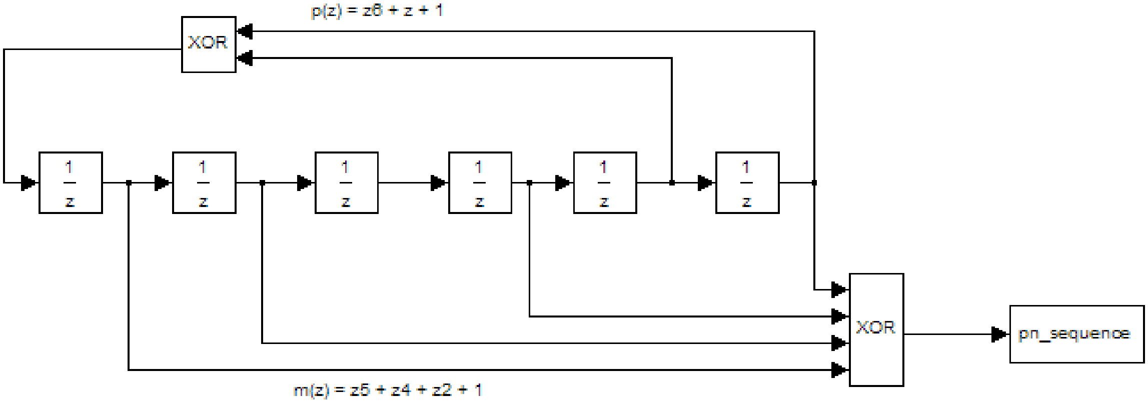
r	Generator Polynomial	r	Generator Polynomial
12	[12 11 8 6 0]	31	[31 28 0]
13	[13 12 10 9 0]	32	[32 31 30 10 0]
14	[14 13 8 4 0]	33	[33 20 0]
15	[15 14 0]	34	[34 15 14 1 0]
16	[16 15 13 4 0]	35	[35 2 0]
17	[17 14 0]	36	[36 11 0]
18	[18 11 0]	37	[37 12 10 2 0]
19	[19 18 17 14 0]	38	[38 6 5 1 0]
20	[20 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]
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 PN Sequence Generator

Set up a PN sequence generator. Define the polynomial in binary vector format or exponential vector format.

This figure defines a PN sequence generator with a generator polynomial $p(z) = z^6 + z + 1$.



Define this PN sequence generator as follows:

```
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])
```

```
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])
```

```
h =
          GenPoly: [1 0 0 0 0 1 1]
    InitialStates: [0 0 0 0 0 1]
    CurrentStates: [0 0 0 0 0 1]
             Mask: [1 1 0 1 0 1]
       NumBitsOut: 1
```

General commsrc.pn Use

Typical `commsrc.pn` is used to output pseudorandom data streams.

Construct a PN object.

```
h = commsrc.pn('Shift',0);
```

Output 10 PN bits.

```
set(h,'NumBitsOut',10);
generate(h)
```

```
ans = 10×1

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

Output 10 more PN bits.

```
generate(h)
```

```
ans = 10×1

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

Reset the object to the initial shift register state values.

```
reset(h);
```

Output 4 PN bits.

```
set(h,'NumBitsOut',4);
generate(h)
```

```
ans = 4×1

     1
     0
     0
     0
```

Copied commsrc.pn Object Behavior

When a `commsrc.pn` object is copied, its states are also copied. Subsequent outputs from the copied object are likely to be different from the initial outputs from the original object.

Construct a PN object and output a sequence from it.

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

```
ans = 5×1

     1
     0
     0
     0
     0
```

Make a copy of `h.` Generate a sequence from the copied object. Because the copy was made after the state of `h` changed, the initial sequence generated by `g` is different from the initial sequence generated from `h`.

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

```
ans = 5×1

     0
     1
     0
     0
     0
```

But if g is reset, it generates the same sequence that h generated.

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

```
ans = 5×1

     1
     0
     0
     0
     0
```

See Also

mask2shift | shift2mask

Introduced in R2009a

commtest.ErrorRate

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.
FrameLength	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 `NumTransmissions` test input and calls its `getInput` method, the error rate test console returns the value stored in `FrameLength`. 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 `DiscreteRandomSource` test input and calls its `getInput` method, the test console generates and returns a frame of symbols. The length of the frame of symbols matches the `FrameLength` property. This property defaults to 500.

Property	Description
IterationMode	Specify how the object determines simulation points. • If set to `Combinatorial`, the object performs simulations for all possible combinations of registered test parameter sweep values. • If set to `Indexed`, 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: (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)

Property	Description
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.

Property	Description
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`. • 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.

Property	Description
	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.
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.
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.

Property	Description
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 parpool 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:

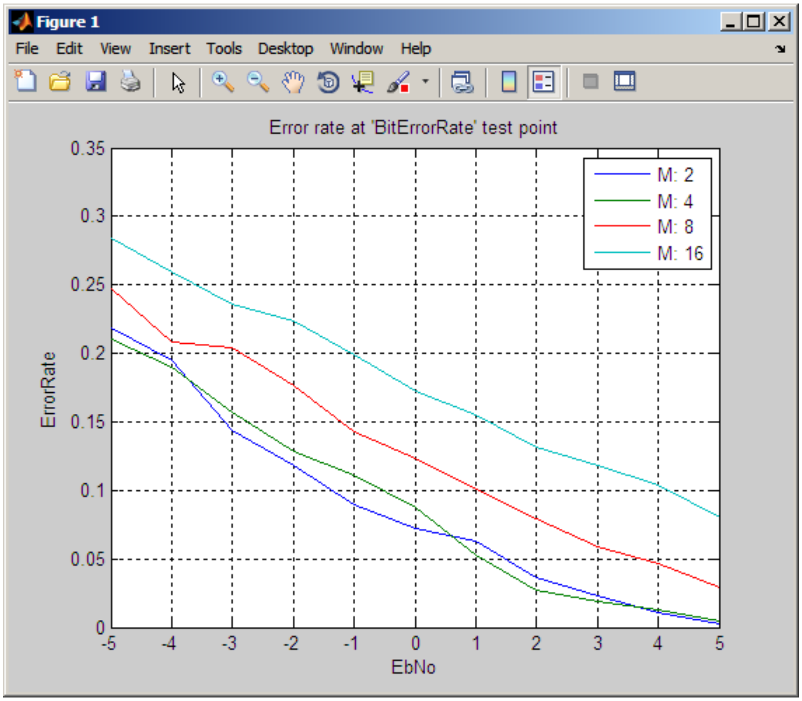


Figure 1
File Edit View Insert Tools Desktop Window Help
Error rate at 'BitErrorRate' test point
ErrorRate
EbNo
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-5
-4
-3
-2
-1
0
1
2
3
4
5
M: 2
M: 4
M: 8
M: 16

See Also

`testconsole.Results`

Topics

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

Introduced in R2009b

compand

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 µ= 255 and A = 87.6.

Examples

μ-Law Compression and Expansion

Generate a data sequence.

```
data = 2:2:12;
```

Compress the input sequence using an μ-law compander. The typical value for μ is 255. The data ranges between 8.1 and 12 instead of between 2 and 12.

```
compressed = compand(data,255,max(data),'mu/compressor')
```

```
compressed = 1×6

    8.1644    9.6394   10.5084   11.1268   11.6071   12.0000
```

Expand the compressed signal. The expanded sequence is nearly identical to the original.

```
expanded = compand(compressed,255,max(data),'mu/expander')
```

```
expanded = 1×6

    2.0000    4.0000    6.0000    8.0000   10.0000   12.0000
```

A-Law Compression and Expansion

Generate a data sequence.

```
data = 1:5;
```

Compress the input sequence using an A-law compander. The typical value for A is 87.5. The data ranges between 3.5 and 5 instead of between 1 and 5.

```
compressed = compand(data,87.6,max(data),'a/compressor')
```

```
compressed = 1×5
```

```
    3.5296    4.1629    4.5333    4.7961    5.0000
```

Expand the compressed signal. The expanded sequence is nearly identical to the original.

```
expanded = compand(compressed,87.6,max(data),'a/expander')
```

```
expanded = 1×5

    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

`dpcmdeco` | `dpcmenco` | `quantiz`

Topics

"Compand a Signal"

Introduced before R2006a

convdeintrlv

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`

Topics

"Interleaving"

Introduced before R2006a

convenc

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

Create Convolutional Codes

Encode five two-bit symbols using a rate 2/3 convolutional code.

```
data = randi([0 1],10,1);
trellis1 = poly2trellis([5 4],[23 35 0; 0 5 13]);
code1 = convenc(data,poly2trellis([5 4],[23 35 0; 0 5 13]));
```

Verify that the encoded output is 15 bits, 3/2 times the length of the input sequence, `data`.

```
length(code1)
```

```
ans = 15
```

Define the encoder's trellis structure explicitly and then use `convenc` to encode 10 one-bit symbols.

```
trellis2 = 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),trellis2);
```

Use the final and initial state arguments when invoking `convenc`. Encode part of `data` , recording final state for later use.

```
[code3,fstate] = convenc(data(1:6),trellis1);
```

Encode the rest of `data`, using `fstate` as an input argument.

```
code4 = convenc(data(7:10),trellis1,fstate);
```

Verify that the `[code3; code4]` matches `code1`.

```
isequal(code1,[code3; code4])
```

```
ans = logical
   1
```

Trellis Structure for a 1/2 Feedforward Convolutional Encoder

Create a trellis structure for a rate 1/2 feedforward convolutional code and use it to encode and decode a random bit stream.

Create a trellis in which the constraint length is 7 and the code generator is specified as a cell array of polynomial character vectors.

```
trellis = poly2trellis(7,{'1 + x^3 + x^4 + x^5 + x^6', ...
    '1 + x + x^3 + x^4 + x^6'})
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 64
          nextStates: [64x2 double]
             outputs: [64x2 double]
```

Generate random binary data, convolutionally encode the data, and decode the data using the Viterbi algorithm.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
decodedData = vitdec(codedData,trellis,34,'trunc','hard');
```

Verify that there are no bit errors in the decoded data.

```
biterr(data,decodedData)
```

```
ans = 0
```

Estimate BER for Hard and Soft Decision Viterbi Decoding

Estimate bit error rate (BER) performance for hard-decision and soft-decision Viterbi decoders in AWGN. Compare the performance to that of an uncoded 64-QAM link.

Set the simulation parameters.

```
clear; close all
rng default
M = 64;                 % Modulation order
```

```
k = log2(M);            % Bits per symbol
EbNoVec = (4:10)';       % Eb/No values (dB)
numSymPerFrame = 1000;   % Number of QAM symbols per frame
```

Initialize the BER results vectors.

```
berEstSoft = zeros(size(EbNoVec));
berEstHard = zeros(size(EbNoVec));
```

Set the trellis structure and traceback length for a rate 1/2, constraint length 7, convolutional code.

```
trellis = poly2trellis(7,[171 133]);
tbl = 32;
rate = 1/2;
```

The main processing loops performs these steps:

- Generate binary data.
- Convolutionally encode the data.
- Apply QAM modulation to the data symbols. Specify unit average power for the transmitted signal.
- Pass the modulated signal through an AWGN channel.
- Demodulate the received signal using hard decision and approximate LLR methods. Specify unit average power for the received signal.
- Viterbi decode the signals using hard and unquantized methods.
- Calculate the number of bit errors.

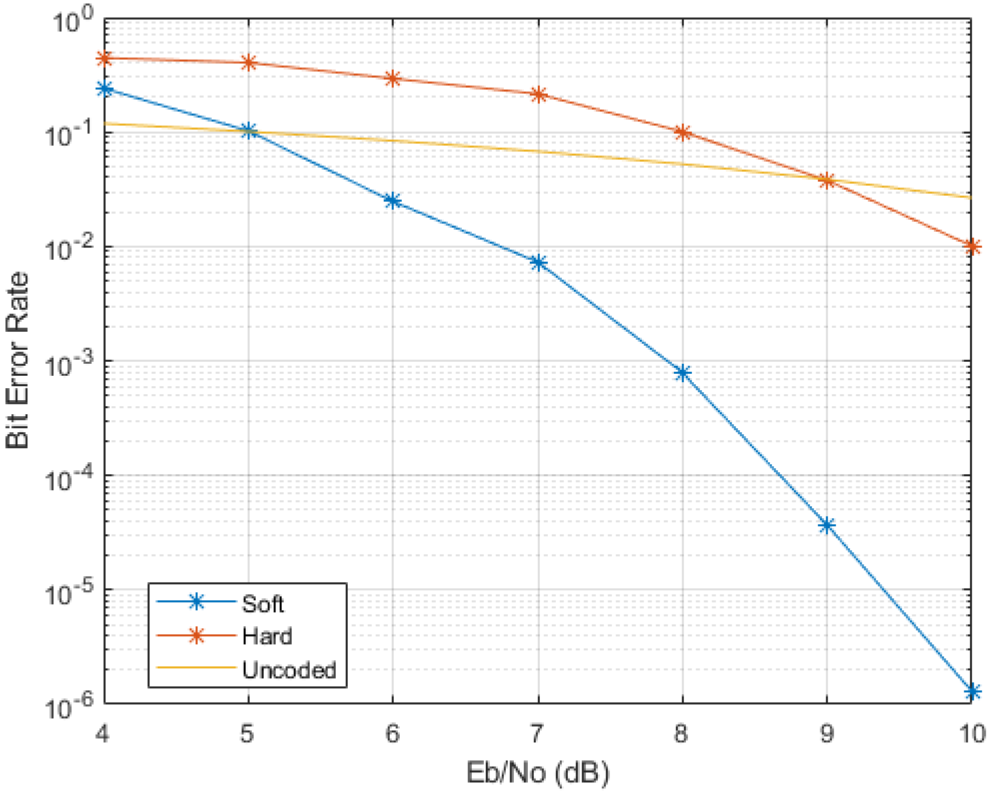
The `while` loop continues to process data until either 100 errors are encountered or 1e7 bits are transmitted.

```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k*rate);
    % Noise variance calculation for unity average signal power.
    noiseVar = 10.^(-snrdB/10);
    % Reset the error and bit counters
    [numErrsSoft,numErrsHard,numBits] = deal(0);

    while numErrsSoft < 100 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame*k,1);
        % Convolutionally encode the data
        dataEnc = convenc(dataIn,trellis);

        % QAM modulate
        txSig = qammod(dataEnc,M,'InputType','bit','UnitAveragePower',true);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal using hard decision (bit) and
        % soft decision (approximate LLR) approaches.
        rxDataHard = qamdemod(rxSig,M,'OutputType','bit','UnitAveragePower',true);
        rxDataSoft = qamdemod(rxSig,M,'OutputType','approxllr', ...
            'UnitAveragePower',true,'NoiseVariance',noiseVar);

        % Viterbi decode the demodulated data
        dataHard = vitdec(rxDataHard,trellis,tbl,'cont','hard');
        dataSoft = vitdec(rxDataSoft,trellis,tbl,'cont','unquant');

        % Calculate the number of bit errors in the frame. Adjust for the
        % decoding delay, which is equal to the traceback depth.
        numErrsInFrameHard = biterr(dataIn(1:end-tbl),dataHard(tbl+1:end));
        numErrsInFrameSoft = biterr(dataIn(1:end-tbl),dataSoft(tbl+1:end));

        % Increment the error and bit counters
        numErrsHard = numErrsHard + numErrsInFrameHard;
        numErrsSoft = numErrsSoft + numErrsInFrameSoft;
        numBits = numBits + numSymPerFrame*k;

    end

    % Estimate the BER for both methods
    berEstSoft(n) = numErrsSoft/numBits;
    berEstHard(n) = numErrsHard/numBits;
end
```

Plot the estimated hard and soft BER data. Plot the theoretical performance for an uncoded 64-QAM channel.

```
semilogy(EbNoVec,[berEstSoft berEstHard],'-*')
hold on
semilogy(EbNoVec,berawgn(EbNoVec,'qam',M))
legend('Soft','Hard','Uncoded','location','best')
grid
```

```
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



As expected, the soft decision decoding produces the best results.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Convolutional Codes”

Introduced before R2006a

convintrlv

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` | `helintrlv` | `muxintrlv`

Topics

“Interleaving”

Introduced before R2006a

convmtx

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 | gf

Topics

“Signal Processing Operations in Galois Fields”

Introduced before R2006a

cosets

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 1-320.

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

`gf` | `minpol`

Introduced before R2006a

crc.detector

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','ReflectI
nput',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.
`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.

Property	Description
`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`

Introduced in R2008a

crc.generator

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.
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.

Property	Description
`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`

Introduced in R2008a

cyclgen

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. Alternatively, you can specify `pol` as a polynomial character vector. For more inofrmation, see “Character Representation of Polynomials”

`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

Parity Check and Generator Matrices for Binary Cyclic Codes

Create parity check and generator matrices for a binary cyclic code having codeword length 7 and message length 4.

Create the generator polynomial using `cyclpoly`.

```
pol = cyclpoly(7,4);
```

Create the parity check and generator matrices. The parity check matrix `parmat` has a 3-by-3 identity matrix embedded in its leftmost columns.

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

```
parmat = 3×7

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


genmat = 4×7

     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
```

Create a parity check matrix in which the code is not systematic. The matrix `parmatn` does not have an embedded 3-by-3 identity matrix.

```
parmatn = cyclgen(7,pol,'nonsys')
```

```
parmatn = 3×7

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

Create the parity check and generator matrices for a (7,3) binary cyclic code. As this is a systematic code, there is a 4-by-4 identity matrix in the leftmost columns of `parmat2`.

```
parmat2 = cyclgen(7,'1 + x^2 + x^3 + x^4')
```

```
parmat2 = 4×7

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

See Also

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

Topics

“Block Codes”

Introduced before R2006a

cyclpoly

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	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

Cyclic Code Generator Polynomials

Create [15,4] cyclic code generator polynomials.

Use the input `'all' to show` all possible generator polynomials for a [15,4] cyclic code. Use the input `'max' to show` that $1+x+x^2+x^3+x^5+x^7+x^8+x^{11}$ is one such polynomial that has the largest number of nonzero terms.

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

```
c1 = 3×12

     1     1     0     0     0     1     1     0     0     0     1     1 ⋯
     1     0     0     1     1     0     1     0     1     1     1     1
     1     1     1     1     0     1     0     1     1     0     0     1
```

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

```
c2 = 1×12

     1     1     1     1     0     1     0     1     1     0     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.

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`

Topics

"Block Codes"

Introduced before R2006a

de2bi

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 in which each row is the binary form of the corresponding element in `d`.

`b = de2bi(d,n)` has an output with `n` columns.

`b = de2bi(d,n,p)` converts a nonnegative decimal integer `d` to a base-`p` row vector.

`b = de2bi(d,[],p)` specifies the base, `p` .

`b = de2bi(d,...,flg)` uses `flg` to determine whether the first column of `b` contains the lowest-order or highest-order digits.

Input Arguments

d — Decimal input

scalar | vector | matrix

Decimal input which can be a scalar, vector, or matrix. Specify elements as nonnegative integers. If `d` is a matrix, it is treated like the column vector `d(:)`.

Example: 4

Example: [10; 5]

Note To ensure an accurate conversion, d must be less than or equal to 2^{52}.

n — Number of output columns

positive integer scalar

The number of output columns specified as a positive scalar. If necessary, the binary representation of d is padded with extra zeros.

Example: 3

p — Base

positive integer scalar

An integer that specifies the base of the output b. Specify as an integer greater than or equal to 2. The first column of b is the lowest base-p digit. The output is padded with extra zeros if necessary so that it has n columns. If d is a nonnegative decimal vector, the output b is a matrix in which each row is the base-p form of the corresponding element in d. If d is a matrix, `de2bi` treats it like the vector `d(:)`.

Example: 8

`flg` — MSB flag

`'right-msb'` | `'left-msb'`

Character vector that determines whether the first column of b contains the lowest-order or highest-order digits. If omitted, `de2bi` assumes `'right-msb'`.

Output Arguments

b — Binary output

vector | matrix

Binary representation of d in the form of a row vector or matrix.

Examples

Convert Decimals to Binary Numbers

Convert decimals 1 through 10 into their equivalent binary representations.

```
d = (1:10)';
b = de2bi(d);
[d b]
```

```
ans = 10×5

     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
```

Convert 3 and 9 into binary numbers. Each value is represented by a four-element row.

```
b = de2bi([3 9])
```

```
b = 2×4

     1     1     0     0
     1     0     0     1
```

Repeat the conversion with the number of columns set to 5. The output is now padded with zeros in the fifth column.

```
bb = de2bi([3 9],5)
```

```
bb = 2×5

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

Convert the decimals 1 through 6 to their base 3 equivalents. Set the leftmost bit as the most significant digit.

```
d = (1:6)';
t = de2bi(d,[],3,'left-msb');
[d t]
```

```
ans = 6×3

     1     0     1
     2     0     2
     3     1     0
     4     1     1
     5     1     2
     6     2     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bi2de`

Introduced before R2006a

decode

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 1-440 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 polynomial character vector or 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 polynomial character vector or 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

Encoding and Decoding with Linear Block Codes

Encode and decode corrupted data using three types of linear block codes.

Hamming Code

Set the code parameters.

```
n = 15;                 % Code length
k = 11;                 % Message length
```

Create a binary message having length `k`.

```
data = randi([0 1],k,1);
```

Encode the message.

```
encData = encode(data,n,k,'hamming/binary');
```

Introduce an error in the 4th bit of the encoded sequence.

```
encData(4) = ~encData(4);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'hamming/binary');
```

```
numerr = biterr(data,decData)
```

```
numerr = 0
```

Linear Block Code

Set the code parameters.

```
n = 7;                % Code length
k = 3;                % Message length
```

Create a binary message having length k.

```
data = randi([0 1],k,1);
```

Create a cyclic generator polynomial. Then, create a parity-check matrix and convert it into a generator matrix.

```
pol = cyclpoly(n,k);
parmat = cyclgen(n,pol);
genmat = gen2par(parmat);
```

Encode the message sequence by using the generator matrix.

```
encData = encode(data,n,k,'linear/binary',genmat);
```

Introduce an error in the 3rd bit of the encoded sequence.

```
encData(3) = ~encData(3);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'linear/binary',genmat);
```

```
Single-error patterns loaded in decoding table.  8 rows remaining.
2-error patterns loaded.  1 rows remaining.
3-error patterns loaded.  0 rows remaining.
```

```
numerr = biterr(data,decData)
```

```
numerr = 0
```

Cyclic Code

Set the code parameters.

```
n = 15;                % Code length
k = 5;                 % Message length
```

Create a binary message having length k.

```
data = randi([0 1],k,1);
```

Create a generator polynomial for a cyclic code. Create a parity-check matrix by using the generator polynomial.

```
gpol = cyclpoly(n,k);
parmat = cyclgen(n,gpol);
```

Create a syndrome decoding tabled by using the parity-check matrix.

```
trt = syndtable(parmat);
```

```
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.
```

Encode the data by using the generator polynomial.

```
encData = encode(data,n,k,'cyclic/binary',gpol);
```

Introduce errors in the 4th and 7th bits of the encoded sequence.

```
encData(4) = ~encData(4);
encData(7) = ~encData(7);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'cyclic/binary',gpol,trt);
```

```
numerr = biterr(data,decData)
```

```
numerr = 0
```

Algorithms

Depending on the decoding method, decode relies on such lower-level functions as hammgen, syndtable, and cyclgen.

See Also

`cyclpoly` | `encode` | `gen2par` | `syndtable`

Topics

"Block Codes"

Introduced before R2006a

deintrlv

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`

Topics

"Interleaving"

Introduced before R2006a

dfe

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.
`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".

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,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))

For example illustrating relationships among properties, see *Linked Properties of an Equalizer Object* in “Accessing Properties of an Equalizer”.

Examples

Decision Feedback Equalization with LMS Adaptation

Equalize a signal using a decision feedback equalizer with least mean square (LMS) adaptation.

Set Up Transmitter

Create a QPSK modulated transmission signal containing random message data. Pass the signal through an arbitrary channel filter to add signal distortion.

```
M = 4; % Alphabet size for modulation
msg = randi([0 M-1],2500,1); % Random message
hMod = comm.QPSKModulator('PhaseOffset',0);
modmsg = hMod(msg); % Modulate using QPSK
chan = [.986; .845; .237; .123+.31i]; % Channel coefficients
filtmsg = filter(chan,1,modmsg); % Introduce channel distortion
```

Set Up Equalizer

Create a DFE object that has 5 forward taps, 3 feedback taps. Specify the least mean square algorithm inline when creating the equalizer object. Initialize additional equalizer properties.

```
dfeObj = dfe(5,3,lms(0.01));
% Set the signal constellation
dfeObj.SigConst = hMod((0:M-1)')';
% Maintain continuity between calls to equalize
dfeObj.ResetBeforeFiltering = 0;
% Define initial coefficients to help convergence
dfeObj.Weights = [0 1 0 0 0 0 0 0];
```

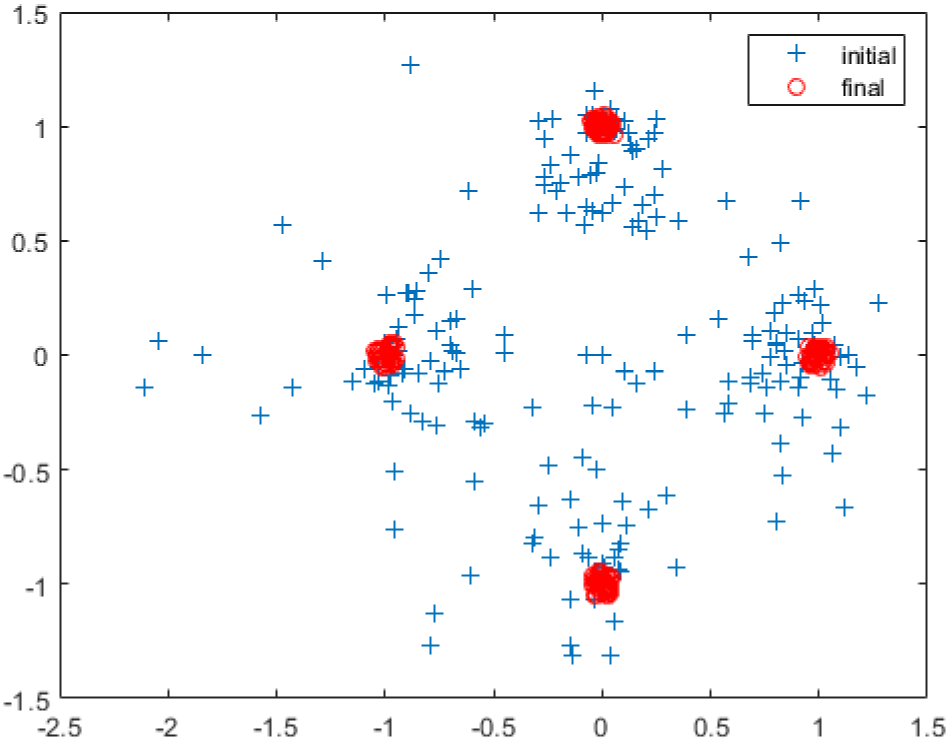
Equalize Received Signal

```
eqRxSig = equalize(dfeObj,filtmsg);
```

Plot Results

Compare the first 200 equalized symbols (`initial`) to the remaining equalized signal (`final`).

```
initial = eqRxSig(1:200);
plot(real(initial),imag(initial),'+')
hold on
final = eqRxSig(end-200:end);
plot(real(final),imag(final),'ro')
legend('initial', 'final')
```



Equalization of the received signal converges within approximately 200 samples.

Apply a Decision Feedback Equalizer (DFE) to An 8-PSK Modulated Signal

Apply a decision feedback equalizer (DFE) to an 8-PSK modulated signal impaired by a frequency selective channel. The DFE uses 600 training symbols.

Create a PSK modulator System object™ and set the modulation order to 8.

```
modulator = comm.PSKModulator('ModulationOrder',8);
```

Create a column vector of 8-ary random integer symbols. Seed the random number generator, `rng`, to produce a predictable sequence of numbers.

```
rng(12345);
data = randi([0 7],5000,1);
```

Use the modulator System object to modulate the random data.

```
modData = modulator(data);
```

Create a Rayleigh channel System object to define a static frequency selective channel with four taps. Pass the modulated data through the channel object.

```
chan = comm.RayleighChannel('SampleRate',1000, ...
    'PathDelays',[0 0.002 0.004 0.008],'AveragePathGains',[0 -3 -6 -9]);
rxSig = chan(modData);
```

Create a DFE equalizer that has 10 feed forward taps and five feedback taps. The equalizer uses the LMS update method with a step size of 0.01.

```
numFFTaps = 10;
numFBTaps = 5;
equalizerDFE = dfe(numFFTaps,numFBTaps,lms(0.01));
```

Set the SigConst property of the DFE equalizer to match the 8-PSK modulator reference constellation. The reference constellation is determined by using the constellation method. For decision directed operation, the DFE must use the same signal constellation as the transmission scheme.

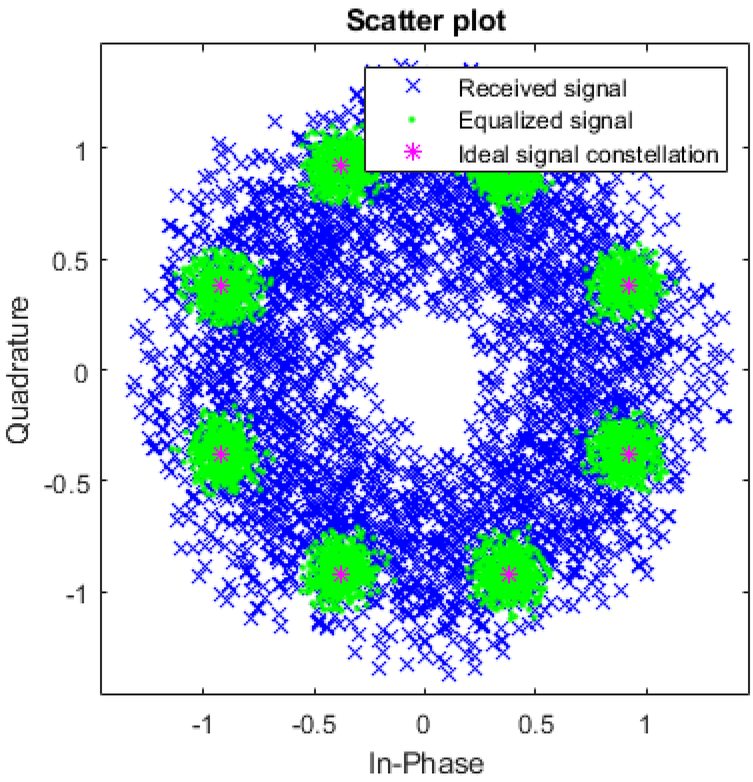
```
equalizerDFE.SigConst = constellation(modulator).';
```

Equalize the signal to remove the effects of channel distortion. Use the first 600 symbols to train the equalizer.

```
trainlen = 600;
[eqSig,detectedSig] = equalize(equalizerDFE,rxSig, ...
    modData(1:trainlen));
```

Plot the received signal, equalizer output after training, and the ideal signal constellation.

```
hScatter = scatterplot(rxSig,1,trainlen,'bx');
hold on
scatterplot(eqSig,1,trainlen,'g.',hScatter);
scatterplot(equalizerDFE.SigConst,1,0,'m*',hScatter);
legend('Received signal','Equalized signal',...
    'Ideal signal constellation');
hold off
```



Create a PSK demodulator System object. Use the object to demodulate the received signal before and after equalization.

```
demod = comm.PSKDemodulator('ModulationOrder',8);
demodSig = demod(rxSig);
demodEqualizedSig = demod(detectedSig);
```

Compute the error rates for the two demodulated signals and compare the results.

```
errorCalc = comm.ErrorRate;
nonEqualizedSER = errorCalc(data(trainlen+1:end), ...
    demodSig(trainlen+1:end));
reset(errorCalc)
equalizedSER = errorCalc(data(trainlen+1:end), ...
    demodEqualizedSig(trainlen+1:end));
disp('Symbol error rates with and without equalizer:')
```

```
Symbol error rates with and without equalizer:
```

```
disp([equalizedSER(1) nonEqualizedSER(1)])
```

```
         0    0.5225
```

The equalizer helps eliminate the distortion introduced by the frequency selective channel and reduces the error rate.

See Also

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

Topics

"Adaptive Equalization"
"Equalization"

Introduced before R2006a

dftmtx

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` | `gf` | `ifft`

Topics

"Signal Processing Operations in Galois Fields"

Introduced before R2006a

distspec

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

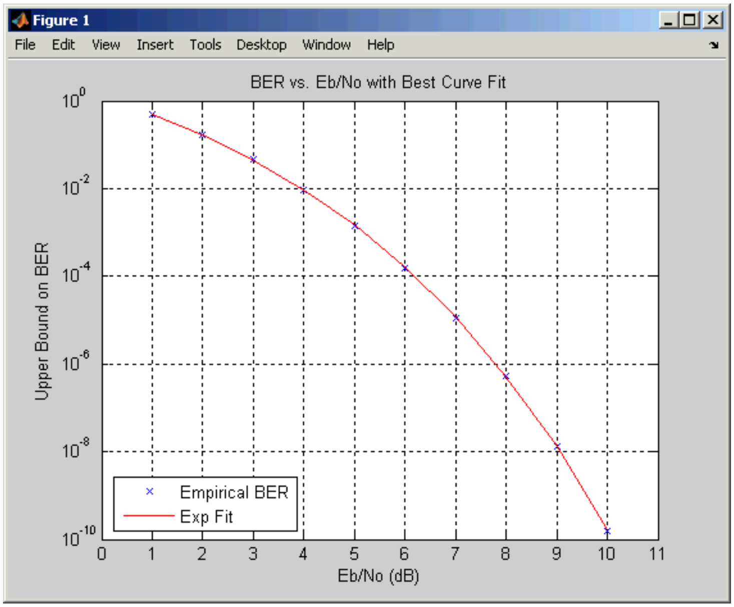
The example below performs these tasks:

- 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`

Introduced before R2006a

doppler

Construct Doppler spectrum structure

Syntax

```
s = doppler(specType)
s = doppler(specType, fieldValue)
s = doppler('BiGaussian', Name,Value)
```

Description

`s = doppler(specType)` constructs a Doppler spectrum structure of type `specType` for use with a fading channel System object. The returned structure, `s`, has default values for its dependent fields.

`s = doppler(specType, fieldValue)` constructs a Doppler spectrum structure of type `specType` for use with a fading channel System object. The returned structure, `s`, has its dependent field specified to `fieldValue`.

`s = doppler('BiGaussian', Name,Value)` constructs a BiGaussian Doppler spectrum structure for use with a fading channel System object. The returned structure, `s`, has dependent fields specified by `Name,Value` pair arguments.

Examples

Construct a Flat Doppler Spectrum Structure

Construct a flat Doppler structure variable for use with channel objects such as `comm.RayleighChannel`.

Invoke the `doppler` function to create a flat Doppler structure variable.

```
s = doppler('Flat')
```

```
s = struct with fields:
    SpectrumType: 'Flat'
```

Create a Bell Doppler Structure Variable

Use the `doppler` function to create a Doppler structure variable having the Bell spectrum.

```
s = doppler('Bell')
```

```
s = struct with fields:
    SpectrumType: 'Bell'
     Coefficient: 9
```

Construct a Rounded Doppler Spectrum Structure with Specified Polynomial

Specify the coefficients of the Doppler spectrum structure variable.

Construct a Rounded Doppler spectrum structure with coefficients `a0`, `a2`, and `a4` set to `2`, `6`, and `1`, respectively.

```
s = doppler('Rounded', [2, 6, 1])
```

```
s = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [2 6 1]
```

Construct a BiGaussian Doppler Spectrum Structure with Specified Field Values

Use the `doppler` function to create a Doppler spectrum structure with the parameters specified for a BiGaussian spectrum.

```
s = doppler('BiGaussian','NormalizedCenterFrequencies', ...
    [.1 .85],'PowerGains',[1 2])
```

```
s = struct with fields:
                    SpectrumType: 'BiGaussian'
    NormalizedStandardDeviations: [0.7071 0.7071]
     NormalizedCenterFrequencies: [0.1000 0.8500]
                      PowerGains: [1 2]
```

The `NormalizedStandardDeviations` field is set to the default value. The `NormalizedCenterFrequencies`, and `PowerGains` fields are set to the values specified from the input arguments.

Input Arguments

`specType` — Spectrum type of Doppler spectrum structure for use with fading channel System object

'Jakes' | 'Flat' | 'Rounded' | 'Bell' | 'Asymmetric Jakes' | 'Restricted Jakes' | 'Gaussian' | 'BiGaussian'

The spectrum type of a Doppler spectrum structure for use with a fading channel System object. Specify this value as a character vector.

The analytical expression for each Doppler spectrum type is described in the "Algorithms" on page 1-367 section.

Data Types: `char`

`fieldValue` — Value of dependent field of Doppler spectrum structure

scalar | vector

The value of the dependent field of the Doppler spectrum structure, specified as a scalar or vector of built-in data type. If you do not specify `fieldValue` , the dependent fields of the spectrum type use the default values.

Spectrum Type	Dependent Field	Description	Default Value
Jakes	—	—	—
Flat	—	—	—

Spectrum Type	Dependent Field	Description	Default Value
Rounded	Polynomial	1-by-3 vector of real finite values, representing the polynomial coefficients, a0, a2 and a4	[1 -1.72 0.785]
Bell	Coefficient	Nonnegative, finite, real scalar representing the Bell spectrum coefficient	9
Asymmetric Jakes	NormalizedFrequencyInterval	1-by-2 vector of real values between –1 and 1, inclusive, representing the minimum and maximum normalized Doppler shifts	[0 1]
Restricted Jakes	NormalizedFrequencyInterval	1-by-2 vector of real values between 0 and 1, inclusive, representing the minimum and maximum normalized Doppler shifts	[0 1]
Gaussian	NormalizedStandardDeviation	Normalized standard deviation of the Gaussian Doppler spectrum, specified as a positive, finite, real scalar	0.7071
BiGaussian	NormalizedStandardDeviations	Normalized standard deviations of the BiGaussian Doppler spectrum, specified as a positive, finite, real 1-by-2 vector	[0.7071 0.7071]

Spectrum Type	Dependent Field	Description	Default Value
	`NormalizedCenterFreqencies`	Normalized center frequencies of the BiGaussian Doppler spectrum specified as a real 1-by-2 vector whose elements fall between –1 and 1	`[0 0]`
	`PowerGains`	Linear power gains of the BiGaussian Doppler spectrum specified as a real nonnegative 1-by-2 vector	`[0.5 0.5]`

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

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `s=doppler('BiGaussian', 'NormalizedStandardDeviations', [.8 .75], 'NormalizedCenterFrequencies', [-.8 0], 'PowerGains', [.6 .6])`

`NormalizedStandardDeviations` — Normalized standard deviations of first and second Gaussian functions

`[1/sqrt(2) 1/sqrt(2)]` (default) | 1-by-2 vector

The normalized standard deviation of the first and second Gaussian functions. You can specify this value as a 1-by-2 vector of positive, finite, real values, of built-in data types.

When you do not specify this dependent field, the default value is `[1/sqrt(2) 1/sqrt(2)]`.

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

`NormalizedCenterFrequencies` — Normalized center frequencies of first and second Gaussian functions
`[0 0]` (default) | 1-by-2 vector

The normalized center frequencies of the first and second Gaussian functions. You can specify this value as a 1-by-2 vector of real values between –1 and 1, of built-in data types.

When you do not specify this dependent field, the default value is `[0 0]`.

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

`PowerGains` — Power gains of first and second Gaussian functions
`[0.5 0.5]` (default) | 1-by-2 vector

The power gains of the first and second Gaussian functions. You can specify this value as a 1-by-2 nonnegative, finite, real vector of built-in data types.

When you do not specify this dependent field, the default value is `[0.5 0.5]`.

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

Algorithms

The following algorithms represent the analytical expressions for each Doppler spectrum type. In each case, f_d denotes the maximum Doppler shift (`MaximumDopplerShift` property) of the associated fading channel System object.

Jakes

The theoretical `Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{1}{\pi f_d \sqrt{1-(f/f_d)^2}}, \quad |f| \le f_d$$

Flat

The theoretical `Flat` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{1}{2f_d},\ |f| \le f_d$$

Rounded

The theoretical `Rounded` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = C_r \left[a_0 + a_2 \left(\frac{f}{f_d} \right)^2 + a_4 \left(\frac{f}{f_d} \right)^4 \right],\ |f| \le f_d$$

where

$$C_r = \frac{1}{2f_d \left[a_0 + \frac{a_2}{3} + \frac{a_4}{5} \right]}$$

and you can specify [a_0, a_2, a_4] in the dependent field, `polynomial`.

Bell

The theoretical `Bell` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{C_b}{1+A \left(\frac{f}{f_d} \right)^2}$$

$$|f| \le f_d$$

where

$$C_b = \frac{\sqrt{A}}{\pi f_d}$$

You can specify *A* in the dependent field, `coefficient`.

Asymmetric Jakes

The theoretical `Asymmetric Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{A_a}{\pi f_d \sqrt{1-(f/f_d)^2}}, \quad -f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$$

$$A_a = \frac{1}{\frac{1}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

where you can specify $f_{\min} / f_d$ and $f_{\max} / f_d$ in the dependent field, `NormalizedFrequencyInterval`.

Restricted Jakes

The theoretical `Restricted Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{A_r}{\pi f_d \sqrt{1-(f/f_d)^2}}, \; 0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$$

where

$$A_r = \frac{1}{\frac{2}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

where you can specify $f_{\min} / f_d$ and $f_{\max} / f_d$ in the dependent field, `NormalizedFrequencyInterval`.

Gaussian

The theoretical `Gaussian` Doppler spectrum, *S(f)* has the analytic formula

$$S_G(f) = \frac{1}{\sqrt{2\pi\sigma_G^2}} \exp\left(-\frac{f^2}{2\sigma_G^2}\right)$$

You can specify σ_G / f_d in the dependent field, `NormalizedStandardDeviation`.

BiGaussian

The theoretical `BiGaussian` Doppler spectrum, *S(f)* has the analytic formula

$$S_G(f) = A_G\left[\frac{C_{G1}}{\sqrt{2\pi\sigma_{G1}^2}} \exp\left(-\frac{(f-f_{G1})^2}{2\sigma_{G1}^2}\right) + \frac{C_{G2}}{\sqrt{2\pi\sigma_{G2}^2}} \exp\left(-\frac{(f-f_{G2})^2}{2\sigma_{G2}^2}\right)\right]$$

where $A_G = \dfrac{1}{C_{G1}+C_{G2}}$ is a normalization coefficient.

You can specify σ_{G1}/f_d and σ_{G2}/f_d in the `NormalizedStandardDeviations` dependent field.

You can specify f_{G1}/f_d and f_{G2}/f_d in the `NormalizedCenterFrequencies` dependent field.

C_{G1} and C_{G2} are power gains that you can specify in the `PowerGains` dependent field.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

MIMO Channel | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2007a

doppler.ajakes

(To be removed) Construct asymmetrical Doppler spectrum object

Syntax

```
dop = doppler.ajakes(freqminmaxajakes)
dop = doppler.ajakes
```

Note `doppler.ajakes` will be removed in a future release. Use `doppler('Asymmetric Jakes', ...)` instead.

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 $-1 \le f_{\min,norm} \le f_{norm} \le f_{\max,norm} \le 1$, 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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2007b

doppler.bell

(To be removed) Construct bell-shaped Doppler spectrum object

Syntax

```
doppler.bell
doppler.bell(coeffbell)
```

Note `doppler.ajakes` will be removed in a future release. Use `doppler('Bell', ...)` instead.

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2009a

doppler.bigaussian

(To be removed) Construct bi-Gaussian Doppler spectrum object

Note `doppler.bigaussian` will be removed in a future release. Use `doppler('BiGaussian', ...)` instead.

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)
`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:

$$S_G(f) = A_G \left[\frac{C_{G1}}{\sqrt{2\pi\sigma_{G1}^2}} \exp\left(-\frac{(f - f_{G1})^2}{2\sigma_{G1}^2} \right) + \frac{C_{G2}}{\sqrt{2\pi\sigma_{G2}^2}} \exp\left(-\frac{(f - f_{G2})^2}{2\sigma_{G2}^2} \right) \right]$$

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007b

doppler.flat

(To be removed) Construct flat Doppler spectrum object

Note `doppler.flat` will be removed in a future release. Use `doppler('Flat')` instead.

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.

Examples

```
%% Create a Rayleigh Channel with Flat Doppler Spectrum
% This example shows how to create a Rayleigh channel object with a flat Doppler spectr
%%
% Set the sample time and maximum Doppler shift.
ts = 1e-6; % sec
fd = 50; % Hz
% Create the Rayleigh channel object.
chan = rayleighchan(ts,fd);
% Observe that the default Doppler spectrum property, SpectrumType, is 'Jakes'.
chan.DopplerSpectrum

ans =

    SpectrumType: 'Jakes'

% Change the Doppler spectrum property of the channel by using doppler.flat.
chan.DopplerSpectrum = doppler.flat
chan =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-06
         DopplerSpectrum: [1x1 doppler.flat]
         MaxDopplerShift: 50
              PathDelays: 0
           AvgPathGaindB: 0
      NormalizePathGains: 1
            StoreHistory: 0
          StorePathGains: 0
               PathGains: -0.6760 + 0.6319i
      ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2007a

doppler.gaussian

(To be removed) Construct Gaussian Doppler spectrum object

Syntax

```
dop = doppler.gaussian
dop = doppler.gaussian(sigmagaussian)
```

Note `doppler.gaussian` will be removed in a future release. Use `doppler('Gaussian',...)` instead.

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007a

doppler.jakes

(To be removed) Construct Jakes Doppler spectrum object

Syntax

Note `doppler.jakes` will be removed in a future release. Use `doppler('Jakes')` instead.

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2007a

doppler.rjakes

(To be removed) Construct restricted Jakes Doppler spectrum object

Syntax

```
dop = doppler.rjakes
dop = doppler.rjakes(freqminmaxrjakes)
```

Note `doppler.rjakes` will be removed in a future release. Use `doppler('Restricted Jakes', ...)` instead.

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

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007a

doppler.rounded

(To be removed) Construct rounded Doppler spectrum object

Syntax

```
dop = doppler.rounded
dop = doppler.rounded(coeffrounded)
```

Note `doppler.rounded` will be removed in a future release. Use `doppler('Rounded', ...)` instead.

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 1-397 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

comm.RayleighChannel | comm.RicianChannel | doppler | stdchan

Topics

“Fading Channels”

Introduced in R2007a

dpcmdeco

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

`dpcmenco` | `dpcmopt` | `quantiz`

Topics

“Differential Pulse Code Modulation”

Introduced before R2006a

dpcmenco

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

`dpcmdeco` | `dpcmopt` | `quantiz`

Topics

"Differential Pulse Code Modulation"

Introduced before R2006a

dpcmopt

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

`dpcmdeco` | `dpcmenco` | `lloyds` | `quantiz`

Topics

“Differential Pulse Code Modulation”

Introduced before R2006a

dpskdemod

Differential phase shift keying demodulation

Syntax

```
z = dpskdemod(y,M)
z = dpskdemod(y,M,phaserot)
z = dpskdemod(y,M,phaserot,symorder)
```

Description

`z = dpskdemod(y,M)` demodulates the complex envelope, `y`, of a DPSK-modulated signal having modulation order `M`.

`z = dpskdemod(y,M,phaserot)` specifies the phase rotation of the DPSK modulation.

`z = dpskdemod(y,M,phaserot,symorder)` also specifies the symbol order.

Examples

DPSK Demodulation

Demodulate DPSK data in a communication channel in which a phase shift is introduced.

Generate a 4-ary data vector and modulate it using DPSK.

```
M = 4;                          % Alphabet size
dataIn = randi([0 M-1],1000,1); % Random message
txSig = dpskmod(dataIn,M);      % Modulate
```

Apply the random phase shift resulting from the transmission process.

```
rxSig = txSig*exp(2i*pi*rand());
```

Demodulate the received signal.

```
dataOut = dpskdemod(rxSig,M);
```

The modulator and demodulator have the same initial condition. However, only the received signal experiences a phase shift. As a result, the first demodulated symbol is likely to be in error. Therefore, you should always discard the first symbol when using DPSK.

Find the number of symbol errors.

```
errs = symerr(dataIn,dataOut)
```

```
errs = 1
```

One symbol is in error. Repeat the error calculation after discarding the first symbol.

```
errs = symerr(dataIn(2:end),dataIn(2:end))
```

```
errs = 0
```

Input Arguments

y — DPSK-modulated input signal

vector | matrix

DPSK-modulated input signal, specified as a real or complex vector or matrix. If `y` is a matrix, the function processes the columns independently.

Data Types: `double` | `single`
Complex Number Support: Yes

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: `double` | `single`

phaserot — Phase rotation

0 (default) | scalar | [ ]

Phase rotation of the DPSK modulation, specified in radians as a real scalar. The total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

If you specify `phaserot` as empty, then `dspkdemod` uses a phase rotation of 0 degrees.

Example: `pi/4`

Data Types: `double` | `single`

`symorder` — Symbol order
`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

Output Arguments

`z` — DPSK-demodulated output signal
vector | matrix

DPSK-demodulated output signal, returned as a vector or matrix having the same number of columns as input signal `y`.

Note The differential algorithm used in this function compares two successive elements of a modulated signal. To determine the first element of vector `z`, or the first row of matrix `z`, the function uses an initial phase rotation of `0`.

Data Types: `double` | `single`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.DPSKDemodulator` | `dpskmod` | `pskdemod` | `pskmod`

Topics

“Phase Modulation”

Introduced before R2006a

dpskmod

Differential phase shift keying modulation

Syntax

```
y = dpskmod(x,M)
y = dpskmod(x,M,phaserot)
y = dpskmod(x,M,phaserot,symorder)
```

Description

`y = dpskmod(x,M)` modulates the input signal using differential phase shift keying (DPSK) with modulation order `M`.

`y = dpskmod(x,M,phaserot)` specifies the phase rotation of the DPSK modulation.

`y = dpskmod(x,M,phaserot,symorder)` also specifies the symbol order.

Examples

View Signal Trajectory of DPSK-Modulated Signal

Plot the output of the `dspkmod` function to view the possible transitions between DPSK symbols.

Set the modulation order to 4 to model DQPSK modulation.

```
M = 4;
```

Generate a sequence of 4-ary random symbols.

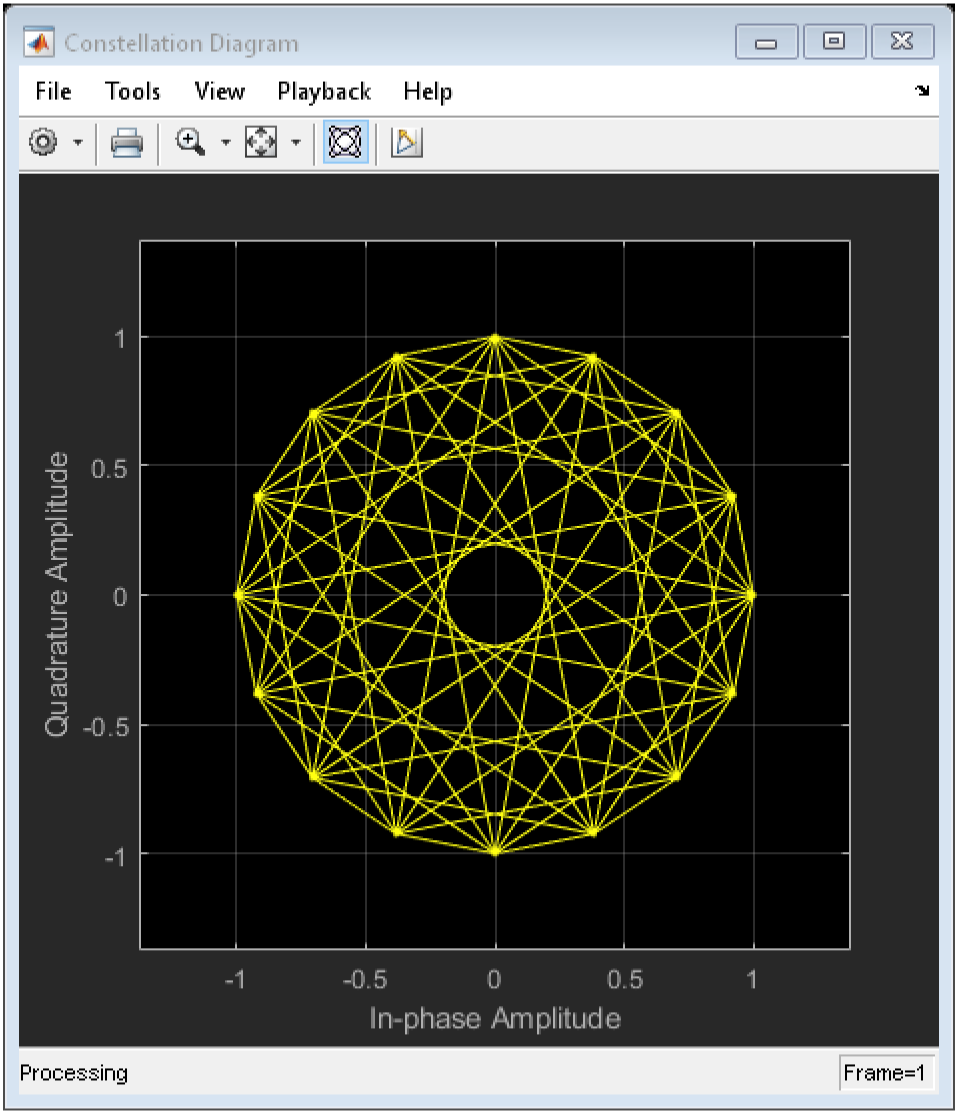
```
x = randi([0 M-1],500,1);
```

Apply DQPSK modulation to the input symbols.

```
y = dpskmod(x,M,pi/8);
```

Specify a constellation diagram object to display a signal trajectory diagram and without displaying the corresponding reference constellation. Display the trajectory.

```
cd = comm.ConstellationDiagram('ShowTrajectory',true,'ShowReferenceConstellation',false
cd(y)
```


Constellation Diagram
File
Tools
View
Playback
Help
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=1

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of x must have values in the range of [0, M – 1].

Data Types: double | single

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double | single

phaserot — Phase rotation

0 (default) | scalar | [ ]

Phase rotation of the DPSK modulation, specified in radians as a real scalar. The total phase shift per symbol is the sum of phaserot and the phase generated by the differential modulation.

If you specify phaserot as empty, then dpskmod uses a phase rotation of 0 degrees.

Example: pi/4

Data Types: double | single

symorder — Symbol order

'bin' (default) | 'gray'

Symbol order, specified as 'bin' or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

- If symorder is 'bin', the function uses a natural binary-coded ordering.
- If symorder is 'gray', the function uses a Gray-coded ordering.

Data Types: char

Output Arguments

y — DPSK-modulated output signal
vector | matrix

Complex baseband representation of a DPSK-modulated output signal, returned as vector or matrix. The columns represent independent channels.

Note An initial phase rotation of 0 is used in determining the first element of the output `y` (or the first row of `y` if it is a matrix with multiple rows), because two successive elements are required for a differential algorithm.

Data Types: `double` | `single`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.DPSKModulator` | `dpskdemod` | `pskdemod` | `pskmod`

Topics

"Phase Modulation"

Introduced before R2006a

dvbs2ldpc

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

Create LDPC Code Parity Check Matrix from DVB-S.2

Create an LDPC parity check matrix for a code rate of 3/5 from the DVB-S.2 standard.

```
p = dvbs2ldpc(3/5);
```

Create an LDPC encoder object from the parity check matrix p.

```
enc = comm.LDPCEncoder(p);
```

The parity check matrix has dimensions of (N-K)-by-N. Determine the length of the input message.

```
msgLength = size(p,2) - size(p,1)
```

```
msgLength = 38880
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

comm.LDPCDecoder | comm.LDPCEncoder

Introduced in R2007a

dvbsapskdemod

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) demodulation

Syntax

```
z = dvbsapskdemod(y,M,stdSuffix)
z = dvbsapskdemod(y,M,stdSuffix,codeIDF)
z = dvbsapskdemod(y,M,stdSuffix,codeIDF,frameLength)
z = dvbsapskdemod( ___ ,Name,Value)
```

Description

`z = dvbsapskdemod(y,M,stdSuffix)` demodulates an APSK input signal, `y`, that was modulated in accordance with the DVB standard identified by `stdSuffix` and the modulation order, `M`. For a description of DVB-compliant APSK demodulation, see “Algorithms” on page 1-424.

`z = dvbsapskdemod(y,M,stdSuffix,codeIDF)` specifies code identifier `codeIDF`, to use when selecting the demodulation parameters.

`z = dvbsapskdemod(y,M,stdSuffix,codeIDF,frameLength)` specifies `codeIDF` and `frameLength` to use when selecting the demodulation parameters.

`z = dvbsapskdemod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate DVB-S2X Specific 64-APSK Signal

Demodulate a 64-APSK signal that was modulated as specified in DVB-S2X. Compute hard decision integer output and verify that the output matches the input.

Set the modulation order and standard suffix. Generate random data.

```
M = 64;
std = 's2x';
x = randi([0 M-1],1000,1);
```

Modulate the data.

```
y = dvbsapskmod(x,M,std);
```

Demodulate the received signal. Compare the demodulated data to the original data.

```
z = dvbsapskdemod(y,M,std);
isequal(z,x)
```

```
ans = logical
   1
```

Demodulate DVB-S2 Specific 32-APSK Signal

Demodulate a 32-APSK signal that was modulated as specified in DVB-S2. Compute hard decision bit output and verify that the output matches the input.

Set the modulation order, standard suffix, and code identifier. Generate random bit data.

```
M = 32;
std = 's2';
codeIDF = '4/5';
numBitsPerSym = log2(M);
x = randi([0 1],100*numBitsPerSym,1,'uint32');
```

Modulate the data. Use a name-value pair to specify bit input data.

```
y = dvbsapskmod(x,M,std,codeIDF,'InputType','bit');
```

Demodulate the received signal. Compare the demodulated data to the original data.

```
z = dvbsapskdemod(y,M,std,'4/5','OutputType','bit', ...
    'OutputDataType','uint32');
isequal(z,x)
```

```
ans = logical
   1
```

Soft Bit Demodulate DVB-SH Specific 16-APSK Signal

Demodulate a DVB-SH compliant 16-APSK signal and calculate soft bits.

Set the modulation order and generate a random bit sequence.

```
M = 16;
std = 'sh';
numSym = 20000;
numBitsPerSym = log2(M);
x = randi([0 1],numSym*numBitsPerSym,1);
```

Modulate the data. Use a name-value pair to specify bit input data.

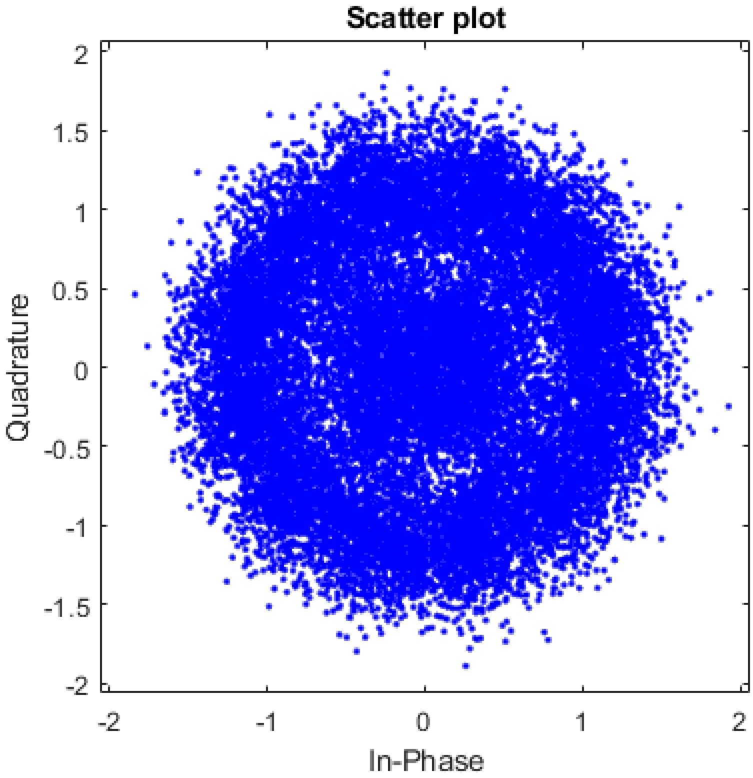
```
txSig = dvbsapskmod(x,M,std,'InputType','bit');
```

Pass the modulated signal through a noisy channel.

```
rxSig = awgn(txSig,10,'measured');
```

View the constellation of the received signal using a scatter plot.

```
scatterplot(rxSig)
```



DVB-SH compliant constellations have unit average power. Demodulate the signal, computing soft bits using the approximate LLR algorithm.

```
z = dvbsapskdemod(rxSig,M,std,'OutputType','approxllr', ...
     'NoiseVariance',0.1);
```

Input Arguments

y — APSK modulated signal
scalar | vector | matrix

APSK modulated signal, specified as a complex scalar, vector, or matrix. When `y` is a matrix, each column is treated as an independent channel.

`y` must be modulated in accordance with Digital Video Broadcasting (DVB) - Satellite Communications standard DVB-S2, DVB-S2X or DVB-SH. For more information, see [1], [2], and [3].

Data Types: `single` | `double`
Complex Number Support: Yes

`M` — Modulation order

integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Data Types: `double`

`stdSuffix` — Standard suffix

`'s2'` | `'s2x'` | `'s2h'`

Standard suffix for DVBS modulation variant, specified as `'s2'`, `'s2x'`, or `'s2h'`.

Data Types: `char` | `string`

`codeIDF` — Code identifier

char | string

Code identifier, specified as a character vector or string. This table lists the accptable `codeIDF` values.

Constellation Order (`M`)	Applicable Standard (`stdSuffix`)	Acceptable Code Identifier (`CodeIDF`) Values
16	`'s2'` or `'s2x'`	`'2/3'`, `'3/4'`, `'4/5'`, `'5/6'`, `'8/9'`, `'9/10'`

Constellation Order (M)	Applicable Standard (stdSuffix)	Acceptable Code Identifier (CodeIDF) Values
16	's2x'	'26/45', '3/5', '28/45', '23/36', '25/36', '13/18', '7/9', '77/90', '100/180', '96/180', '90/180', '18/30', '20/30'
32	's2' or 's2x'	'3/4', '4/5', '5/6', '8/9', '9/10'
32	's2x'	'32/45', '11/15', '7/9', '2/3'
64	's2x'	'11/15', '7/9', '4/5', '5/6', '128/180'
128	's2x'	'3/4', '7/9'
256	's2x'	'32/45', '3/4', '116/180', '20/30', '124/180', '22/30'

For more information, refer to Tables 9 and 10 in the DVB-S2 standard [1] and Table 17a in the DVB-S2X standard [2].

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

frameLength — Frame length

'normal' (default) | 'short'

Frame length, specified as 'normal' or 'short'. The function usesframeLength and codeIDF to select the modulation parameters.

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = dvbsapskdemod(x,M,stdSuffix,'InputType','bit','OutputDataType','single');`

OutputType — Output type

`'integer'` (default) | `'bit'` | `'llr'` | `'approxllr'`

Output type, specified as the comma-separated pair consisting of `'OutputType'` and `'integer'`, `'bit'`, `'llr'`, or `'approxllr'`. For a description of returned output, see `z`.

Data Types: `char` | `string`

OutputDataType — Output data type

`'double'` (default) | `...`

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and one of the indicated data types. Acceptable values for `'OutputDataType'` depend on the `'OutputType'` value.

`'OutputType'` Value	Acceptable `'OutputDataType'` Values
`'integer'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, or `'uint32'`
`'bit'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'logical'`

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'integer'` or `'bit'`.

Data Types: `char` | `string`

UnitAveragePower — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the functon scales the constellation based on specifications in the relevant standard, as described in [1] and [2].

Note When `stdSuffix` is set to `'sh'`, the constellation always has unit average power.

Dependencies

This name-value pair argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `logical`

`NoiseVariance` — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of `'NoiseVariance'` and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

Tip When the output type is set to `'llr'`, an exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are a very small values

When the output returns any of these values, try setting output type to `'approxllr'` instead. The approximate LLR algorithm does not compute exponentials.

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'llr'` or `'approxllr'`.

Data Types: `double`

`PlotConstellation` — Option to plot constellation
`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

`z` — Demodulated signal
scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of the output vary depending on the specified `'OutputType'` value.

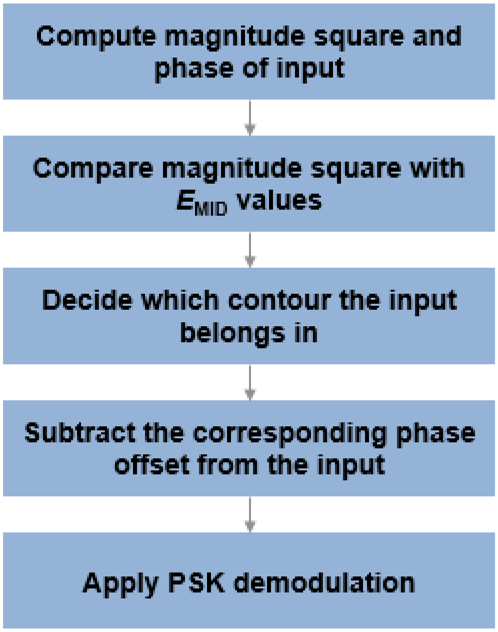
`'OutputType'` Value	Return Value of `dvbsapskdemod`	Dimensions of `z`
`'integer'`	Demodulated integer values from 0 to (`M` – 1)	`z` has the same dimensions as input `y`.
`'bit'`	Demodulated bits	The number of rows in `z` is $\log_2$(sum(`M`)) times the number of rows in `y`. Each demodulated symbol is mapped to a group of $\log_2$(sum(`M`)) elements in a column, where the first element represents the MSB and the last element represents the LSB.
`'llr'`	Log-likelihood ratio value for each bit	
`'approxllr'`	Approximate log-likelihood ratio value for each bit	

Algorithms

DVB Compliant APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding as described in [4].

Calculate the radial partitions of the constellation by taking the mean of consecutive elements of radii. Square the radial partition values and store them in a variable E_{MID}.


Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide which contour the input belongs in
Subtract the corresponding phase offset from the input
Apply PSK demodulation

DVB Compliant APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio algorithms are available: exact LLR and approximate LLR. Exact LLR provides the greatest accuracy but is slower, while approximate LLR is less accurate but faster.

For a description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

[4] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskmod` | `genqamdemod` | `mil188qamdemod` | `pskdemod` | `qamdemod`

System Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator` | `comm.RectangularQAMDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

dvbsapskmod

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) modulation

Syntax

```
y = dvbsapskmod(x,M,stdSuffix)
y = dvbsapskmod(x,M,stdSuffix,codeIDF)
y = dvbsapskmod(x,M,stdSuffix,codeIDF,frameLength)
y = dvbsapskmod( ___ ,Name,Value)
```

Description

`y = dvbsapskmod(x,M,stdSuffix)` performs APSK modulation on the input signal, `x`, in accordance with the DVB standard identified by `stdSuffix` and the modulation order, `M`.

`y = dvbsapskmod(x,M,stdSuffix,codeIDF)` specifies the code identifier, `codeIDF`, to use when selecting the modulation parameters.

`y = dvbsapskmod(x,M,stdSuffix,codeIDF,frameLength)` specifies `codeIDF` and `frameLength` to use when selecting the modulation parameters.

`y = dvbsapskmod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply DVB-S2X 32-APSK Modulation to Data

Modulate data using the DVB-S2X standard specified 32-APSK modulation scheme. Display the result in a scatter plot.

Set the modulation order and the suffix identifying the DVB-S2X standard. Create a data vector with all possible symbols.

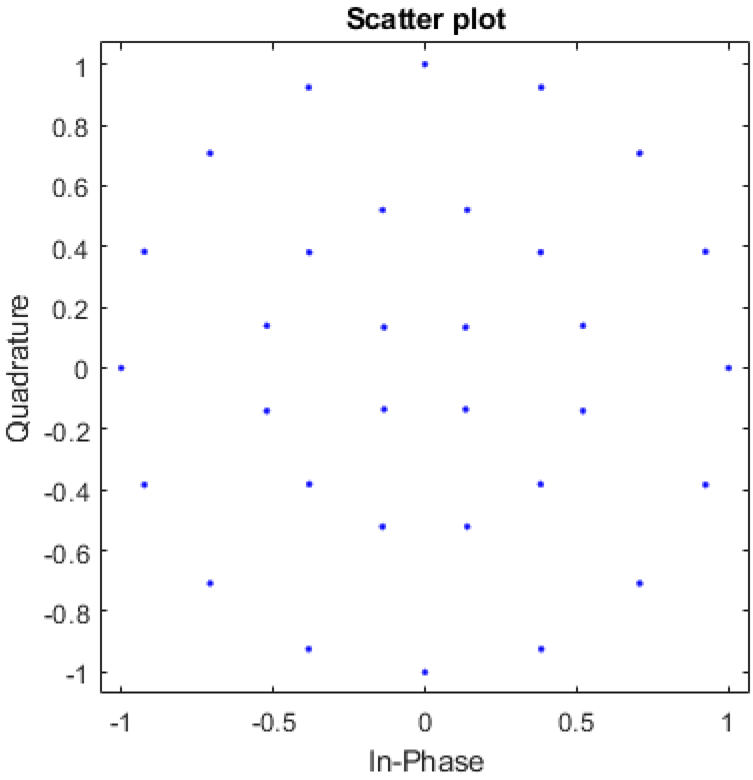
```
M = 32;
stdSuffix = 's2x';
x = (0:M-1);
```

Modulate the data.

```
y = dvbsapskmod(x,M,stdSuffix);
```

Display the constellation using a scatter plot.

```
scatterplot(y)
```



Apply DVB-S2X 64-APSK Modulation Specifying Code Identifier

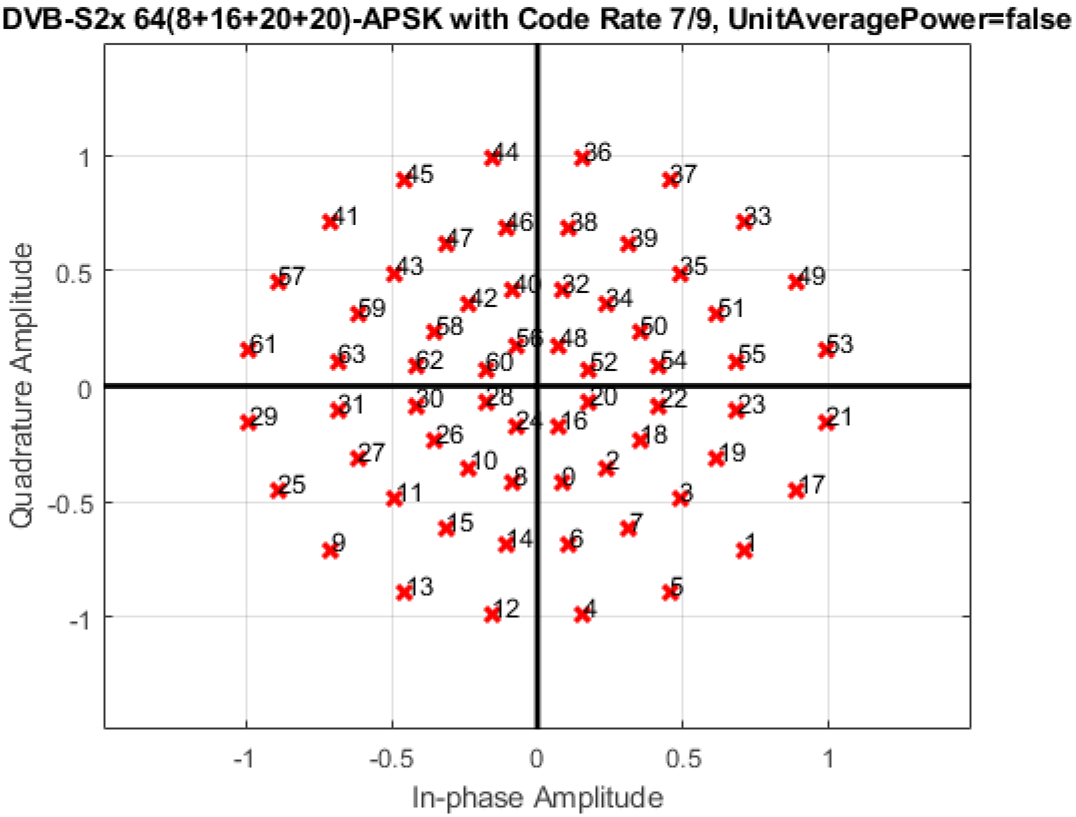
Modulate data using 64-APSK as specified in DVB-S2X standard. Normalize modulated signal power to 1.

Set the modulation order and standard suffix. Generate 1000 symbols of random data in one channel.

```
M = 64;
std = 's2x';
x = randi([0 M-1],1000,1);
```

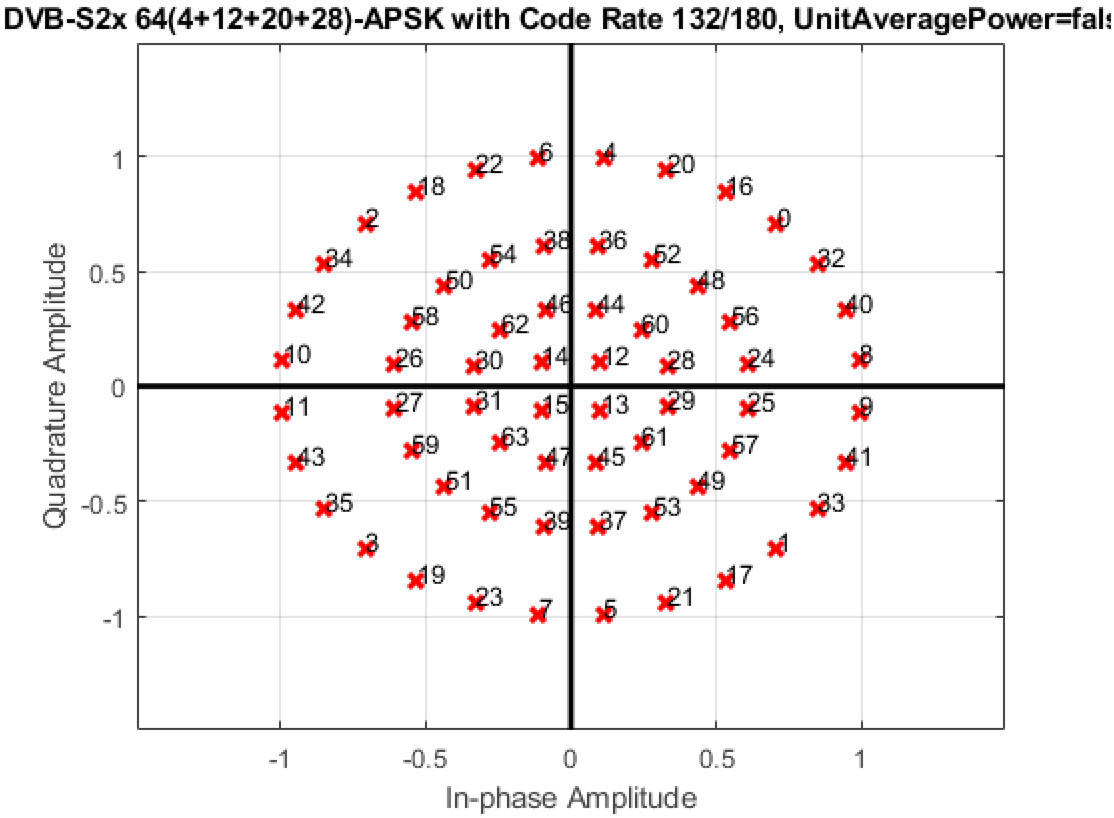
Modulate the data according to the 64-APSK constellation for the code identifier 7/9 and plot the reference constellation.

```
y1 = dvbsapskmod(x,M,std,'7/9','PlotConstellation',true);
```



Modulate setting the code identifier to 132/180 and observe the constellation structure differences.

```
y2 = dvbsapskmod(x,M,std,'132/180','PlotConstellation',true);
```



Apply DVB-S2 16-APSK Modulation Change Frame Length

Modulate data using 16-APSK as specified in DVB-S2 standard for normal and short frame lengths. Compute the output signal power.

Set the modulation order and the standard suffix. Generate random bit data for 1000 symbols in one channel.

```
M = 16;
std = 's2';
x = randi([0 1],1000*log2(M),1);
```

Set the input type to bit and modulate the data according to the 16-APSK constellation for code identifier 2/3. Use the default normal frame length.

```
y1 = dvbsapskmod(x,M,std,'2/3','InputType','bit');
```

Modulate the data using different settings, set the code-identifier to 8/9 and use a short frame length.

```
y2 = dvbsapskmod(x,M,std,'8/9','short','InputType','bit');
```

The average power of the modulated signal changes based on the code identifier. Compute the average power of the modulated signals.

```
y1avgPow = mean(abs(y1).^2)
```

```
y1avgPow = 0.7590
```

```
y2avgPow = mean(abs(y2).^2)
```

```
y2avgPow = 0.7716
```

Normalize 16-APSK Modulated DVB Signals by Average Power

Modulate data applying 16-APSK as specified in the DVB-SH and DVB-S2 standards. Normalize the modulator output so that it has an average signal power of 1 W.

Set the modulation order and generate all possible symbols.

```
M = 16;
x =  0:M-1;
```

Modulate the data applying 16-APSK as specified in DVB-SH. Use a name-value pair to specify single data type output.

```
y1 = dvbsapskmod(x,M,'sh','OutputDataType','single');
```

Modulate the data applying 16-APSK as specified in DVB-S2. Use a name-value pair to specify single data type output.

```
y2 = dvbsapskmod(x,M,'s2','OutputDataType','single');
```

Modulate the data applying 16-APSK as specified in DVB-S2. Use name-value pairs to set unit average power to true and to specify single data type output.

```
y3 = dvbsapskmod(x,M,'s2','UnitAveragePower',true,'OutputDataType','single');
```

Check which signals have unit average power.

```
y1avgPow = mean(abs(y1).^2)
```

```
y1avgPow = single
    1.0000
```

```
y2avgPow = mean(abs(y2).^2)
```

```
y2avgPow = single
    0.7752
```

```
y3avgPow = mean(abs(y3).^2)
```

```
y3avgPow = single
    1.0000
```

Input Arguments

x — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers that range from 0 to (M – 1), where M is the modulation order.

Note To process the input signal as binary elements, set 'InputType' value to 'bit'. For binary inputs, the number of rows must be an integer multiple of $\log_2$(M). A group of $\log_2$(M) bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

M — Modulation order

integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Data Types: `double`

`stdSuffix` — Standard suffix

`'s2'` | `'s2x'` | `'s2h'`

Standard suffix for DVBS modulation variant, specified as `'s2'`, `'s2x'`, or `'s2h'`.

Data Types: `char` | `string`

`codeIDF` — Code identifier

char | string

Code identifier, specified as a character vector or string. This table lists the acceptable `codeIDF` values.

Constellation Order (`M`)	Applicable Standard (`stdSuffix`)	Acceptable Code Identifier (`CodeIDF`) Values
16	`'s2'` or `'s2x'`	`'2/3'`, `'3/4'`, `'4/5'`, `'5/6'`, `'8/9'`, `'9/10'`
16	`'s2x'`	`'26/45'`, `'3/5'`, `'28/45'`, `'23/36'`, `'25/36'`, `'13/18'`, `'7/9'`, `'77/90'`, `'100/180'`, `'96/180'`, `'90/180'`, `'18/30'`, `'20/30'`
32	`'s2'` or `'s2x'`	`'3/4'`, `'4/5'`, `'5/6'`, `'8/9'`, `'9/10'`
32	`'s2x'`	`'32/45'`, `'11/15'`, `'7/9'`, `'2/3'`
64	`'s2x'`	`'11/15'`, `'7/9'`, `'4/5'`, `'5/6'`, `'128/180'`
128	`'s2x'`	`'3/4'`, `'7/9'`
256	`'s2x'`	`'32/45'`, `'3/4'`, `'116/180'`, `'20/30'`, `'124/180'`, `'22/30'`

For more information, refer to Tables 9 and 10 in the DVB-S2 standard, [1], and Table 17a in the DVB-S2X standard, [2].

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

frameLength — Frame length

'normal' (default) | 'short'

Frame length, specified as 'normal' or 'short'. frameLength and codeIDF are used to determine the modulation parameters.

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: y = dvbsapskmod(x,M,std,'InputType','bit','OutputDataType','single');

InputType — Input type

'integer' (default) | 'bit'

Input type, specified as the comma-separated pair consisting of 'InputType' and either 'integer' or 'bit'. To use 'integer', the input signal must consist of integer values from 0 to (M – 1). To use 'bit', the input signal must contain binary values and the number of rows must be an integer multiple of $\log_2(M)$.

Data Types: char | string

UnitAveragePower — Unit average power flag

false (default) | true

Unit average power flag, specified as the comma-separated pair consisting of 'UnitAveragePower' and a logical scalar. When this flag is true, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the functon scales the constellation based on specifications in the relevant standard, as described in [1] and [2].

Note When `stdSuffix` is set to `'sh'`, the constellation always has unit average power.

Dependencies

This name-value pair argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `logical`

`OutputDataType` — Output data type

`'double'` (default) | `'single'`

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and either `'double'` or `'single'`.

Data Types: `char` | `string`

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

y — Modulated signal

scalar | vector | matrix

Modulated signal, returned as a complex scalar, vector, or matrix. The dimensions of `y` depend on the specified `'InputType'` value.

`'InputType'` Value	Dimensions of `y`
`'integer'`	`y` has the same dimensions as input `x`.

'InputType' Value	Dimensions of y
'bit'	The number of rows in y equals the number of rows in x divided by $\log_2$(M) times.

Data Types: double | single

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

apskmod | dvbsapskdemod | genqammod | mil188qammod | pskmod | qammod

System Objects
`comm.GeneralQAMModulator` | `comm.PSKModulator` | `comm.RectangularQAMModulator`

Introduced in R2018a

encode

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 **Information Formats** table 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.

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].'		

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.

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 polynomial character vector or 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 polynomial character vector or 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

Encoding and Decoding with Linear Block Codes

Encode and decode corrupted data using three types of linear block codes.

Hamming Code

Set the code parameters.

```
n = 15;                % Code length
k = 11;                % Message length
```

Create a binary message having length `k`.

```
data = randi([0 1],k,1);
```

Encode the message.

```
encData = encode(data,n,k,'hamming/binary');
```

Introduce an error in the 4th bit of the encoded sequence.

```
encData(4) = ~encData(4);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'hamming/binary');
```

```
numerr = biterr(data,decData)
```

```
numerr = 0
```

Linear Block Code

Set the code parameters.

```
n = 7;                  % Code length
k = 3;                  % Message length
```

Create a binary message having length `k`.

```
data = randi([0 1],k,1);
```

Create a cyclic generator polynomial. Then, create a parity-check matrix and convert it into a generator matrix.

```
pol = cyclpoly(n,k);
parmat = cyclgen(n,pol);
genmat = gen2par(parmat);
```

Encode the message sequence by using the generator matrix.

```
encData = encode(data,n,k,'linear/binary',genmat);
```

Introduce an error in the 3rd bit of the encoded sequence.

```
encData(3) = ~encData(3);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'linear/binary',genmat);
```

```
Single-error patterns loaded in decoding table.  8 rows remaining.
2-error patterns loaded.  1 rows remaining.
3-error patterns loaded.  0 rows remaining.
```

```
numerr = biterr(data,decData)
numerr = 0
```

Cyclic Code

Set the code parameters.

```
n = 15;                % Code length
k = 5;                 % Message length
```

Create a binary message having length k.

```
data = randi([0 1],k,1);
```

Create a generator polynomial for a cyclic code. Create a parity-check matrix by using the generator polynomial.

```
gpol = cyclpoly(n,k);
parmat = cyclgen(n,gpol);
```

Create a syndrome decoding tabled by using the parity-check matrix.

```
trt = syndtable(parmat);
```

```
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.
```

Encode the data by using the generator polynomial.

```
encData = encode(data,n,k,'cyclic/binary',gpol);
```

Introduce errors in the 4th and 7th bits of the encoded sequence.

```
encData(4) = ~encData(4);
encData(7) = ~encData(7);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'cyclic/binary',gpol,trt);
```

```
numerr = biterr(data,decData)
numerr = 0
```

Algorithms

Depending on the encoding method, `encode` relies on such lower-level functions as `hammgen` and `cyclgen`.

See Also

`cyclgen` | `cyclpoly` | `decode` | `hammgen`

Topics

"Block Codes"

Introduced before R2006a

equalize

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

cma | dfe | lineareq | lms | normlms | rls | signlms | varlms

Topics

“Equalization”

Introduced before R2006a

evdoForwardReferenceChannels

Define 1xEV-DO forward reference channel

Syntax

```
cfg = evdoForwardReferenceChannels(wv)
cfg = evdoForwardReferenceChannels(wv,numpackets)
```

Description

`cfg = evdoForwardReferenceChannels(wv)` returns a structure, `cfg`, that defines 1xEV-DO forward link parameters given the input waveform identifier, `wv`. Pass this structure to the `evdoForwardWaveformGenerator` function to generate a forward link reference channel waveform.

For all syntaxes, `evdoForwardReferenceChannels` creates a configuration structure that is compliant with the cdma2000 high data rate packet specification, [1].

`cfg = evdoForwardReferenceChannels(wv,numpackets)` specifies the number of packets to be generated.

Examples

Generate 1xEV-DO Release 0 Forward Link Waveform

Create a configuration structure for a Release 0 channel having a 921.6 kbps data rate and transmitted over two slots.

```
config = evdoForwardReferenceChannels('Rel0-921600-2');
```

Display the number of slots and the data rate.

```
config.PacketSequence
```

```
ans = struct with fields:
    MACIndex: 0
    DataRate: 921600
    NumSlots: 2
```

Generate the complex waveform using the associated waveform generator function, `evdoForwardWaveformGenerator`.

```
wv = evdoForwardWaveformGenerator(config);
```

Generate 1xEV-DO Revision A Forward Link Waveform

Create a structure to transmit a Revision A 1xEV-DO channel consisting of three 1024-bit packets transmitted over 2 slots with a 64-bit preamble length.

```
config = evdoForwardReferenceChannels('RevA-1024-2-64',3);
```

Verify that the function created a 1-by-3 structure array. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans = 1x3 struct array with fields:
    MACIndex
    PacketSize
    NumSlots
    PreambleLength
```

Examine the first structure element to verify the packet size, number of slots, and preamble length match what you specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
          MACIndex: 0
        PacketSize: 1024
          NumSlots: 2
    PreambleLength: 64
```

Generate the waveform.

```
wv = evdoForwardWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector.

Parameter Field	Values	Description
`wv`	`'Rel0-38400-16'` | `'Rel0-76800-8'` | `'Rel0-153600-4'` | `'Rel0-307200-2'` | `'Rel0-307200-4'` | `'Rel0-614400-1'` | `'Rel0-614400-2'` | `'Rel0-921600-2'` | `'Rel0-1228800-1'` | `'Rel0-1228800-2'` | `'Rel0-1843200-1'` | `'Rel0-2457600-1'`	Character vector representing the 1xEV-DO Release 0 reference channel with data rate in bps and number of slots. For example, you can specify `'Rel0-153600-4'` to create a structure that represents a reference channel with a 153,600 bps data rate and uses four slots.

Parameter Field	Values	Description
	'RevA-128-1-64' | 'RevA-128-2-128' | 'RevA-128-4-256' | 'RevA-128-4-1024' | 'RevA-128-8-512' | 'RevA-256-1-64' | 'RevA-256-2-128' | 'RevA-256-4-256' | 'RevA-256-4-1024' | 'RevA-256-8-512' | 'RevA-256-16-1024' | 'RevA-512-1-64' | 'RevA-512-2-64' | 'RevA-512-2-128' | 'RevA-512-4-128' | 'RevA-512-4-256' | 'RevA-512-4-1024' | 'RevA-512-8-512' | 'RevA-512-16-1024' | 'RevA-1024-1-64' | 'RevA-1024-2-64' | 'RevA-1024-2-128' | 'RevA-1024-4-128' | 'RevA-1024-4-256' | 'RevA-1024-8-512' | 'RevA-1024-16-1024' | 'RevA-2048-1-64' | 'RevA-2048-2-64' | 'RevA-2048-4-128' | 'RevA-3072-1-64' | 'RevA-3072-2-64' 'RevA-4096-1-64' | 'RevA-4096-2-64' | 'RevA-5120-1-64' | 'RevA-5120-2-64'	Character vector representing the 1xEV-DO Revision A reference channel with the packet size in bits, the number of slots, and the preamble length in chips. For example, you can specify 'RevA-256-1-64' to create a reference channel having a 256-bit packet, transmitted in one slot, with a 64-bit preamble length.

Example: 'Rel0-614400-2'

Example: 'RevA-4096-2-64'

Data Types: char

numpackets — Number of packets
1 (default) | positive integer scalar

Number of packets, specified as a positive integer.

Example: 4

Data Types: double

Output Arguments

cfg — Configuration of the parameters and channels used by the waveform generator
structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO
PNOffset	Nonnegative scalar integer [0, 511]	PN offset of the base station
IdleSlotsWithControl	'Off' | 'On'	Include idle slots with control channels
EnableControl	'Off' | 'On'	Enable control signaling
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer [1, 8]	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000short' | 'Custom' | 'Off'	Select filter type or disable filtering

Parameter Field	Values	Description
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when the `FilterType` field is set to `'Custom'`)
InvertQ	`'Off'` | `'On'`	Negate the quadrature output
EnableModulation	`'Off'` | `'On'`	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when `EnableModulation` is `'On'`)
PacketSequence	Structure	See **PacketSequence substructure**.
PacketDataSources	Structure	See **PacketDataSources substructure**.

PacketSequence Substructure

Include the `PacketSequence` substructure in the `cfg` structure to define a sequence of data packets for consecutive transmission. The `PacketSequence` substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet
	Release 0	
DataRate	38400 | 76800 | 153600 | 307200 | 614400 | 921600 | 1228800 | 1843200 | 2457600	Data rate (bps)
NumSlots	Positive scalar integer	Number of slots
	Revision A	
PacketSize	128 | 256 | 512 | 1024 | 2048 | 3072 | 4096 | 5120	Packet size (bits)
NumSlots	1 | 2 | 4 | 8 | 16	Number of slots
PreambleLength	64 | 128 | 256 | 512 | 1024	Preamble length (chips)

PacketDataSources Substructure

Include a `PacketDataSources` substructure in the `cfg` structure to define a set of matching data sources for each MAC index. The `PacketDataSources` substructure contains these fields.

Parameter Field	Values	Description
`MACIndex`	Positive scalar integer	MAC index associated with the packet
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
`EnableCoding`	`'Off'` | `'On'`	Enable error correction coding

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`evdoForwardWaveformGenerator` | `evdoReverseReferenceChannels`

Introduced in R2015b

evdoForwardWaveformGenerator

Generate 1xEV-DO forward link waveform

Syntax

```
waveform = evdoForwardWaveformGenerator(cfg)
```

Description

`waveform = evdoForwardWaveformGenerator(cfg)` returns the 1xEV-DO forward link `waveform` as defined by the parameter configuration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties the function uses to generate a 1xEV-DO waveform. You can generate `cfg` by using the `evdoForwardReferenceChannels` function.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all parameter combinations are supported. To ensure that the input argument is valid, use the `evdoForwardReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate 1xEV-DO Revision A Forward Link Waveform

Create a structure to transmit a Revision A 1xEV-DO channel consisting of three 1024-bit packets transmitted over 2 slots with a 64-bit preamble length.

```
config = evdoForwardReferenceChannels('RevA-1024-2-64',3);
```

Verify that the function created a 1-by-3 structure array. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans = 1x3 struct array with fields:
    MACIndex
    PacketSize
    NumSlots
    PreambleLength
```

Examine the first structure element to verify the packet size, number of slots, and preamble length match what you specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
          MACIndex: 0
        PacketSize: 1024
          NumSlots: 2
    PreambleLength: 64
```

Generate the waveform.

```
wv = evdoForwardWaveformGenerator(config);
```

Generate 1xEV-DO Forward Link Waveform with Custom Filter

Create a structure to generate two packets of a 1.8 Mbps Release 0 channel.

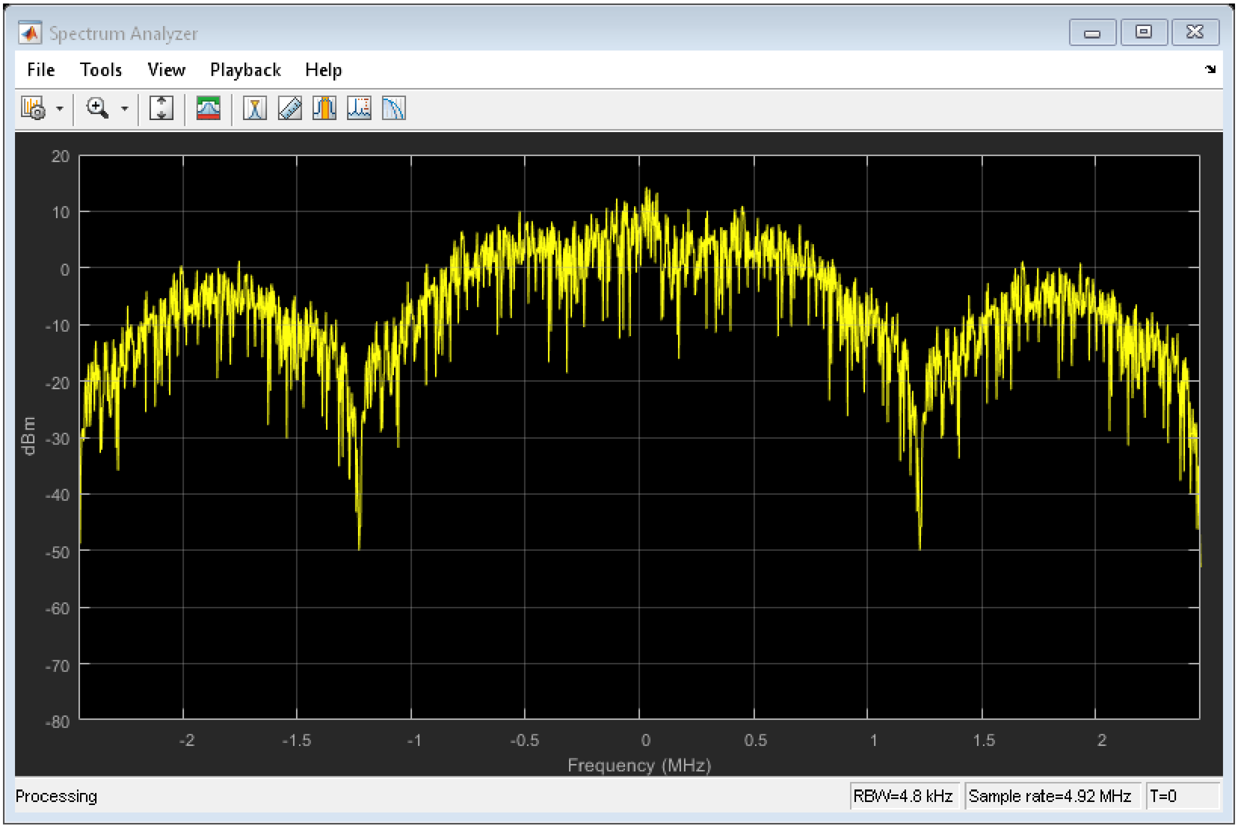
```
config = evdoForwardReferenceChannels('Rel0-1843200-1',2);
```

Calculate the sample rate of the waveform.

```
fs = 1.2288e6 * config.OversamplingRatio;
```

Disable the internal filter of the `evdoForwardWaveformGenerator` function. Generate the 1xEV-DO waveform. Plot the spectrum of the waveform.

```
config.FilterType = 'off';
wv = evdoForwardWaveformGenerator(config);

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Create a lowpass FIR filter with a 500 kHz passband, a 750 kHz stopband, and a stopband attenuation of 60 dB.

```
d = designfilt('lowpassfir', ...
    'PassbandFrequency',500e3, ...
    'StopbandFrequency',750e3, ...
    'StopbandAttenuation',60, ...
    'SampleRate',fs);
```

Change the filter type to `'Custom'` and specify the coefficients from the digital filter, `d`.

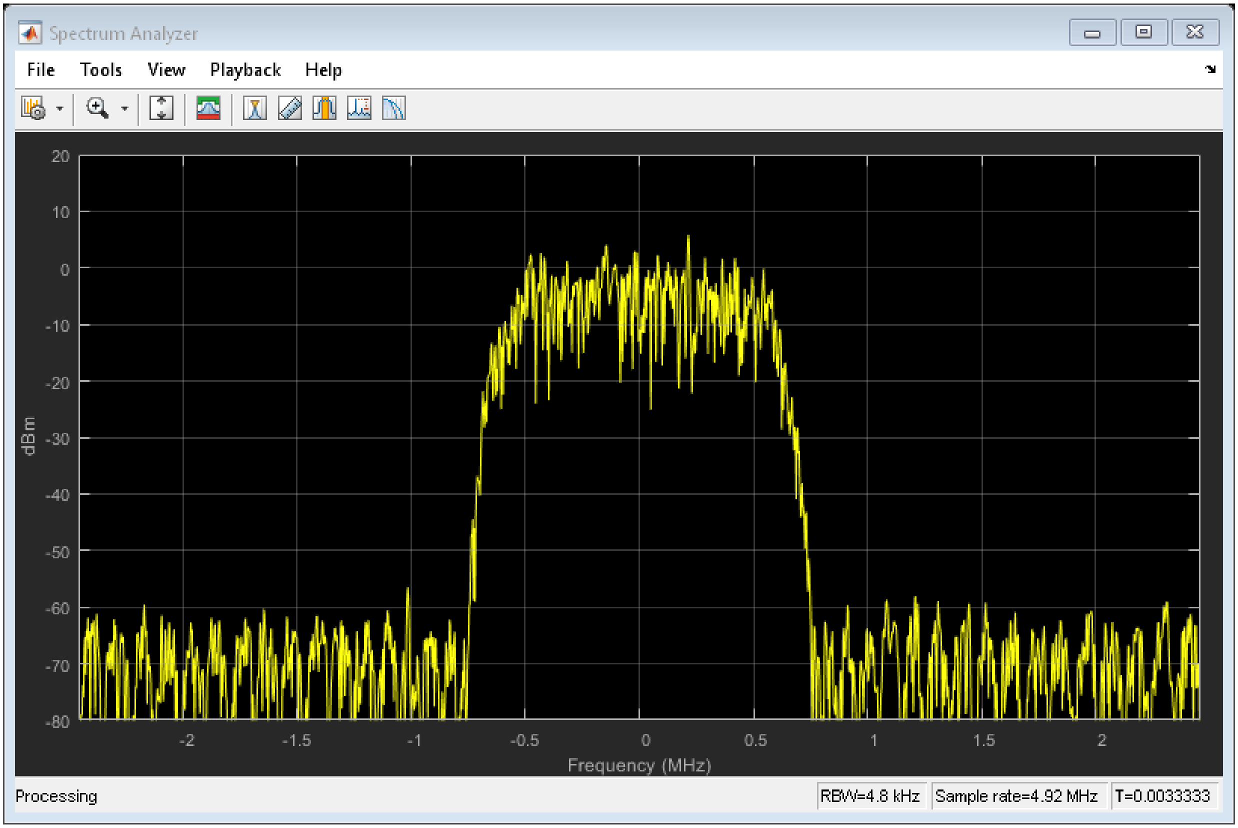
```
config.FilterType = 'Custom';
config.CustomFilterCoefficients = d.Coefficients;
```

Generate the waveform using the custom filter coefficients.

```
wv = evdoForwardWaveformGenerator(config);
```

Plot the spectrum of the filtered 1xEV-DO waveform.

```
step(sa,wv)
```



The filter attenuates the waveform by 60 dB for frequencies outside of ± 750 kHz.

Input Arguments

cfg — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO
PNOffset	Nonnegative scalar integer [0, 511]	PN offset of the base station
IdleSlotsWithControl	'Off' | 'On'	Include idle slots with control channels
EnableControl	'Off' | 'On'	Enable control signaling
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer [1, 8]	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000short' | 'Custom' | 'Off'	Select filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when the FilterType field is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
PacketSequence	Structure	See **PacketSequence substructure**.
PacketDataSources	Structure	See **PacketDataSources substructure**.

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet

Parameter Field	Values	Description
Release 0		
DataRate	38400 | 76800 | 153600 | 307200 | 614400 | 921600 | 1228800 | 1843200 | 2457600	Data rate (bps)
NumSlots	Positive scalar integer	Number of slots
Revision A		
PacketSize	128 | 256 | 512 | 1024 | 2048 | 3072 | 4096 | 5120	Packet size (bits)
NumSlots	1 | 2 | 4 | 8 | 16	Number of slots
PreambleLength	64 | 128 | 256 | 512 | 1024	Preamble length (chips)

PacketDataSources Substructure

Include a `PacketDataSources` substructure in the `cfg` structure to define a set of matching data sources for each MAC index. The `PacketDataSources` substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet
DataSource	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	`'Off'` | `'On'`	Enable error correction coding

Output Arguments

`waveform` — Modulated baseband waveform comprising the primary physical channels

complex vector array

Modulated baseband waveform comprising the primary cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`cdma2000ForwardReferenceChannels` | `cdma2000ReverseWaveformGenerator`

Introduced in R2015b

evdoReverseReferenceChannels

Define 1xEV-DO reverse reference channel

Syntax

```
cfg = evdoReverseReferenceChannels(wv)
cfg = evdoReverseReferenceChannels(wv,numpackets)
```

Description

`cfg = evdoReverseReferenceChannels(wv)` returns a structure, `cfg`, that defines 1xEV-DO reverse link parameters given the input waveform identifier, `wv`. Pass this structure to the `evdoReverseWaveformGenerator` function to generate a reverse link reference channel waveform.

For all syntaxes, `evdoReverseReferenceChannels` creates a structure that is compliant with the cdma2000 high data rate packet specification,[1].

`cfg = evdoReverseReferenceChannels(wv,numpackets)` specifies the number of packets to be generated.

Examples

Generate 1xEV-DO Reverse Channel Waveform

Create a structure to generate a Release 0, 1xEV-DO waveform having a 19.2 kbps data rate.

```
config = evdoReverseReferenceChannels('Rel0-19200');
```

Verify that the packet has a data rate of 19.2 kbps.

```
config.PacketSequence.DataRate
```

```
ans = 19200
```

Generate the complex waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Generate 1xEV-DO Revision A Reverse Link Waveform

Create a structure for a Revision A 1xEV-DO channel having 2048-bit packets, transmitted in 12 slots. Specify that five packets are transmitted.

```
config = evdoReverseReferenceChannels('RevA-2048-12',5);
```

Verify that a 1-by-5 structure array is created. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans = 1x5 struct array with fields:
    Power
    DataSource
    EnableCoding
    PayloadSize
    NumSlots
    DataRate
```

Examine the first structure element to verify the packet size and the number of slots are as specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
           Power: 0
      DataSource: {'PN9'  [1]}
    EnableCoding: 'On'
     PayloadSize: 2048
        NumSlots: 12
        DataRate: 102400
```

Generate the waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector.

Parameter Field	Values	Description
wv	'Rel0-9600' | 'Rel0-19200' | 'Rel0-38400' | 'Rel0-76800' | 'Rel0-153600'	Character vector representing the 1xEV-DO Release 0 data rate in bps. For example, you can specify 'Rel0-153600' to create a structure corresponding to a Release 0 reference channel having a 153,600 bps data rate.

Parameter Field	Values	Description
	`'RevA-128-4'` | `'RevA-128-8'` | `'RevA-128-12'` | `'RevA-128-16'` | `'RevA-256-4'` | `'RevA-256-8'` | `'RevA-256-12'` | `'RevA-256-16'` | `'RevA-512-4'` | `'RevA-512-8'` | `'RevA-512-12'` | `'RevA-512-16'` | `'RevA-768-4'` | `'RevA-768-8'` | `'RevA-768-12'` | `'RevA-768-16'` | `'RevA-1024-4'` | `'RevA-1024-8'` | `'RevA-1024-12'` | `'RevA-1024-16'` | `'RevA-1536-4'` | `'RevA-1536-8'` | `'RevA-1536-12'` | `'RevA-1536-16'` | `'RevA-2048-4'` | `'RevA-2048-8'` | `'RevA-2048-12'` | `'RevA-2048-16'` | `'RevA-3072-4'` | `'RevA-3072-8'` | `'RevA-3072-12'` | `'RevA-3072-16'` | `'RevA-4096-4'` | `'RevA-4096-8'` | `'RevA-4096-12'` | `'RevA-4096-16'` | `'RevA-6144-4'` |	Character vector representing the 1xEV-DO Revision A packet size in bits and the number of slots. For example, you can specify `'RevA-256-4'` to create a structure corresponding to a Revision A reference channel having 256-bit packets and transmitted in four slots.

Parameter Field	Values	Description
	'RevA-6144-8' | 'RevA-6144-12' | 'RevA-6144-16' | 'RevA-8192-4' | 'RevA-8192-8' | 'RevA-8192-12' | 'RevA-8192-16' | 'RevA-12288-4' | 'RevA-12288-8' | 'RevA-12288-12' | 'RevA-12288-16'	

Example: 'Rel0-38400'

Example: 'RevA-3072-12'

Data Types: char

numpackets — Number of packets

1 (default) | positive integer scalar

Number of packets, specified as a positive integer.

Example: 2

Data Types: double

Output Arguments

cfg — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO applicable standard
LongCodeMaskI	42-bit binary number	Long code identifier for in-phase channel
LongCodeMaskQ	42-bit binary number	Long code identifier for quadrature channel
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Custom' | 'Off'	Specify the filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when FilterType is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
ACKChannel	Structure	See **ACKChannel substructure**.
PilotChannel	Structure	See **PilotChannel substructure**.
AuxPilotChannel	Not present or structure	See **AuxPilotChannel substructure**.
PacketSequence	Structure	See **PacketSequence substructure**.

ACKChannel Substructure

Include the ACKChannel substructure in the cfg structure to specify the acknowledgment channel. The ACKChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)

Parameter Fields	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

PilotChannel Substructure

Include the PilotChannel substructure in the cfg structure to specify the pilot channel. The PilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

AuxPilotChannel Substructure

Include the AuxPilotChannel substructure in the cfg structure to specify the auxiliary pilot channel, which is available only for Revision A. The AuxPilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)

Parameter Fields	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
Power	Real scalar	MAC index associated with the packet
EnableCoding	'Off' | 'On'	Enable error correction coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
Release 0		
DataRate	9600 | 19200 | 38400 | 76800 | 153600	Data rate (bps)
Revision A		
PacketSize	128 | 256 | 512 | 768 | 1024 | 1536 | 2048 | 3072 | 4096 | 6144 | 8192 | 12288	Packet size (bits)
NumSlots	4 | 8 | 12 | 16	Number of slots

Data Types: `struct`

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`evdoForwardReferenceChannels` | `evdoReverseWaveformGenerator`

Introduced in R2015b

evdoReverseWaveformGenerator

Generate 1xEV-DO reverse link waveform

Syntax

```
waveform = evdoReverseWaveformGenerator(cfg)
```

Description

`waveform = evdoReverseWaveformGenerator(cfg)` returns the 1xEV-DO reverse link `waveform` as defined by the parameter configration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties used by the function to generate a 1xEV-DO waveform. You can generate `cfg` by using the `evdoReverseReferenceChannels` function.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all parameter combinations are supported. To ensure that the input argument is valid, use the `evdoReverseReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate 1xEV-DO Reverse Channel Waveform

Create a structure to generate a Release 0, 1xEV-DO waveform having a 19.2 kbps data rate.

```
config = evdoReverseReferenceChannels('Rel0-19200');
```

Verify that the packet has a data rate of 19.2 kbps.

```
config.PacketSequence.DataRate
```

```
ans = 19200
```

Generate the complex waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Generate 1xEV-DO Reverse Link Waveform with Custom Filter

Create a structure to generate four packets of a Revision A channel having 768-bit packets transmitted over eight slots.

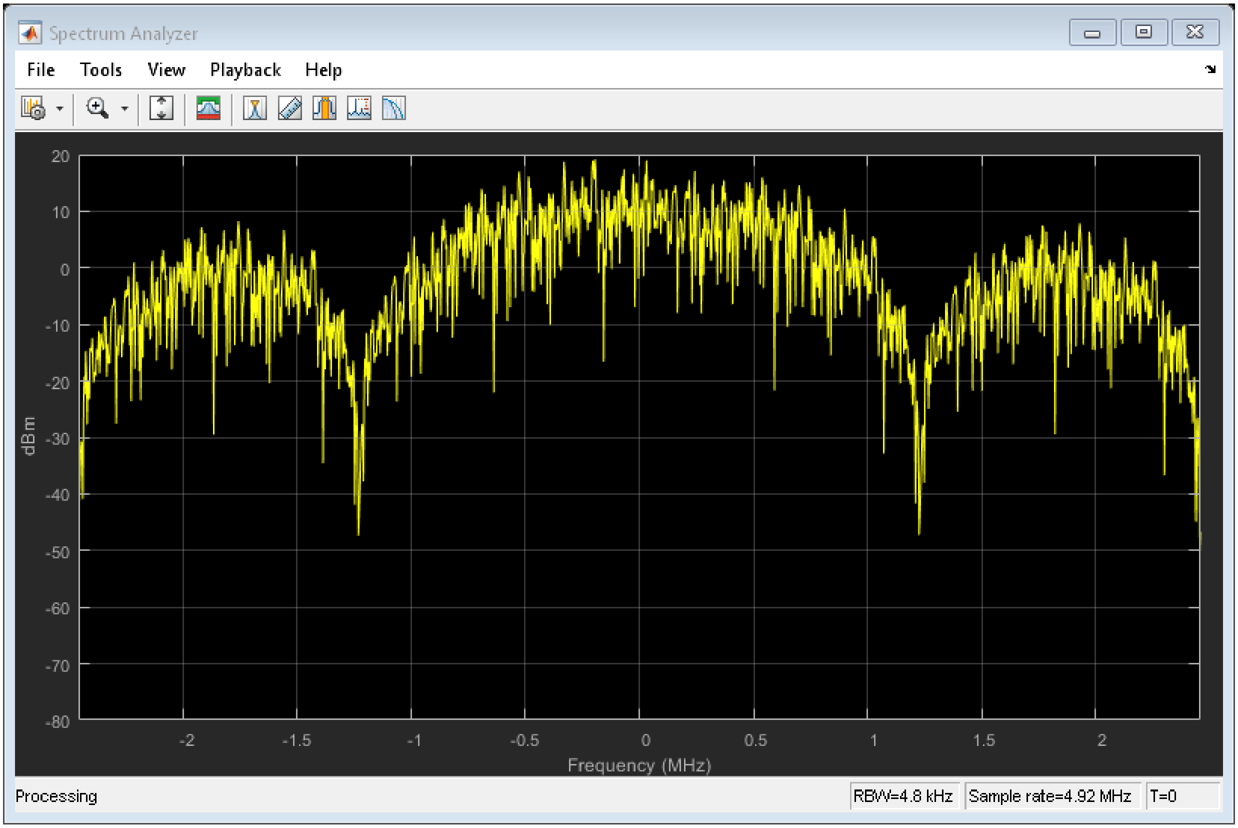
```
config = evdoReverseReferenceChannels('RevA-768-8',4);
```

Calculate the sample rate of the waveform.

```
fs = 1.2288e6 * config.OversamplingRatio;
```

Disable the internal filter of the `evdoReverseWaveformGenerator`. Generate the 1xEV-DO waveform. Plot the spectrum of the waveform.

```
config.FilterType = 'off';
wv = evdoReverseWaveformGenerator(config);

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Create a lowpass FIR filter with a 500 kHz passband, a 750 kHz stopband, and a stopband attenuation of 60 dB.

```
d = designfilt('lowpassfir', ...
    'PassbandFrequency',500e3, ...
    'StopbandFrequency',750e3, ...
    'StopbandAttenuation',60, ...
    'SampleRate',fs);
```

Change the filter type to `'Custom'` and specify the coefficients from the digital filter, `d`.

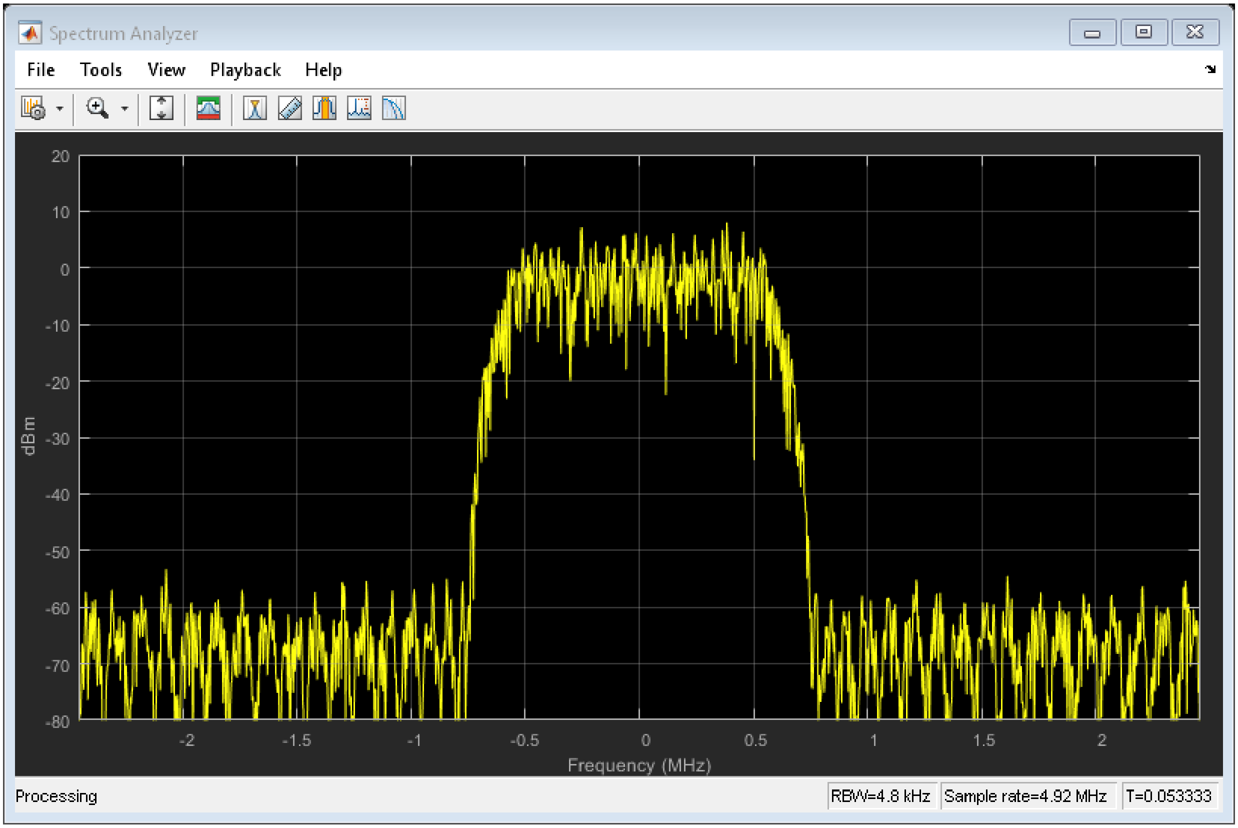
```
config.FilterType = 'Custom';
config.CustomFilterCoefficients = d.Coefficients;
```

Generate the waveform using the custom filter coefficients.

```
wv = evdoReverseWaveformGenerator(config);
```

Plot the spectrum of the filtered 1xEV-DO waveform.

```
step(sa,wv)
```



The filter attenuates the waveform by 60 dB for frequencies outside of ± 750 kHz.

Input Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	`'Release0'` | `'RevisionA'`	1xEV-DO applicable standard
LongCodeMaskI	42-bit binary number	Long code identifier for in-phase channel
LongCodeMaskQ	42-bit binary number	Long code identifier for quadrature channel
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	`'cdma2000Long'` | `'cdma2000Short'` | `'Custom'` | `'Off'`	Specify the filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when `FilterType` is set to `'Custom'`)
InvertQ	`'Off'` | `'On'`	Negate the quadrature output
EnableModulation	`'Off'` | `'On'`	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when `EnableModulation` is `'On'`)
ACKChannel	Structure	See **`ACKChannel` substructure**.
PilotChannel	Structure	See **`PilotChannel` substructure**.
AuxPilotChannel	Not present or structure	See **`AuxPilotChannel` substructure**.
PacketSequence	Structure	See **`PacketSequence` substructure**.

ACKChannel Substructure

Include the `ACKChannel` substructure in the `cfg` structure to specify the acknowledgment channel. The `ACKChannel` substructure contains these fields.

Parameter Fields	Values	Description
Enable	`'On'` | `'Off'`	Character vector to enable or disable the channel

Parameter Fields	Values	Description
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

PilotChannel Substructure

Include the PilotChannel substructure in the cfg structure to specify the pilot channel. The PilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

AuxPilotChannel Substructure

Include the AuxPilotChannel substructure in the cfg structure to specify the auxiliary pilot channel, which is available only for Revision A. The AuxPilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel

Parameter Fields	Values	Description
`Power`	Real scalar	Channel power (dBW)
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
`EnableCoding`	`'On'` | `'Off'`	Enable channel coding

PacketSequence Substructure

Include the `PacketSequence` substructure in the `cfg` structure to define a sequence of data packets for consecutive transmission. The `PacketSequence` substructure contains these fields.

Parameter Field	Values	Description
`Power`	Real scalar	MAC index associated with the packet
`EnableCoding`	`'Off'` | `'On'`	Enable error correction coding
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
Release 0		
`DataRate`	`9600` | `19200` | `38400` | `76800` | `153600`	Data rate (bps)
Revision A		
`PacketSize`	`128` | `256` | `512` | `768` | `1024` | `1536` | `2048` | `3072` | `4096` | `6144` | `8192` | `12288`	Packet size (bits)

Parameter Field	Values	Description
`NumSlots`	4 | 8 | 12 | 16	Number of slots

Output Arguments

`waveform` — Modulated baseband waveform comprising the physical channels
complex vector array

Modulated baseband waveform comprising the 1xEV-DO physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*. URL: 3gpp2.org.

See Also

`evdoForwardWaveformGenerator` | `evdoReverseReferenceChannels`

Introduced in R2015b

eyediagram

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 character vector whose format and meaning are the same as in the `plot` function. The default value 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

`comm.EyeDiagram` | `plot` | `scatterplot`

Introduced before R2006a

EyeScope

(To be removed) Launch eye diagram scope for eye diagram object H

Note `eyescope` will be removed in a future release. Use `comm.EyeDiagram` instead.

Syntax

```
eyescope
eyescope(h)
```

Description

Eye Diagram Scope is a graphical user interface (GUI) that enables you to visualize and measure the effects that various impairments, such as noise, jitter, and filtering, have on a modulated signal. The scope performs a probability density function (pdf) analysis on the signal to illustrate its trajectory in time, and to calculate such quantities as eye SNR, RMS jitter, rise time, and fall time. The scope also enables you to import and compare measurement results for eye diagrams of multiple signals.

There are two ways to call EyeScope:

- `eyescope` calls an empty scope
- `eyescope(h)` calls the scope and displays 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 cannot determine the work space name, type `help inputname` at the MATLAB command line.

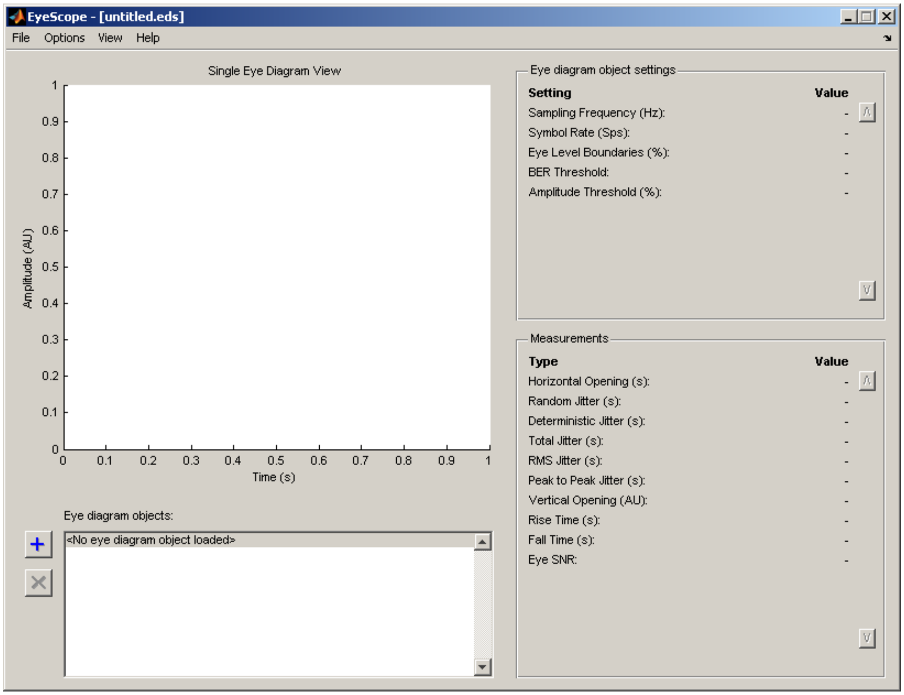
For more information, see "Eye Diagram Analysis".

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.



Eye diagram objects:

<No eye diagram object loaded>

Eye diagram object settings

Setting	Value
Sampling Frequency (Hz):	-
Symbol Rate (Sps):	-
Eye Level Boundaries (%):	-
BER Threshold:	-
Amplitude Threshold (%):	-

Measurements

Type	Value
Horizontal Opening (s):	-
Random Jitter (s):	-
Deterministic Jitter (s):	-
Total Jitter (s):	-
RMS Jitter (s):	-
Peak to Peak Jitter (s):	-
Vertical Opening (AU):	-
Rise Time (s):	-
Fall Time (s):	-
Eye SNR:	-

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.

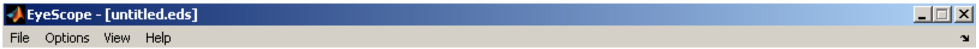
The EyeScope Environment

- "EyeScope Menu Bar" on page 1-483
- "Eye Diagram Object Plot and Plot Controls" on page 1-483
- "Eye Diagram Object Settings Panel" on page 1-485
- "Measurements" on page 1-486

EyeScope Menu Bar

EyeScope Menu Bar

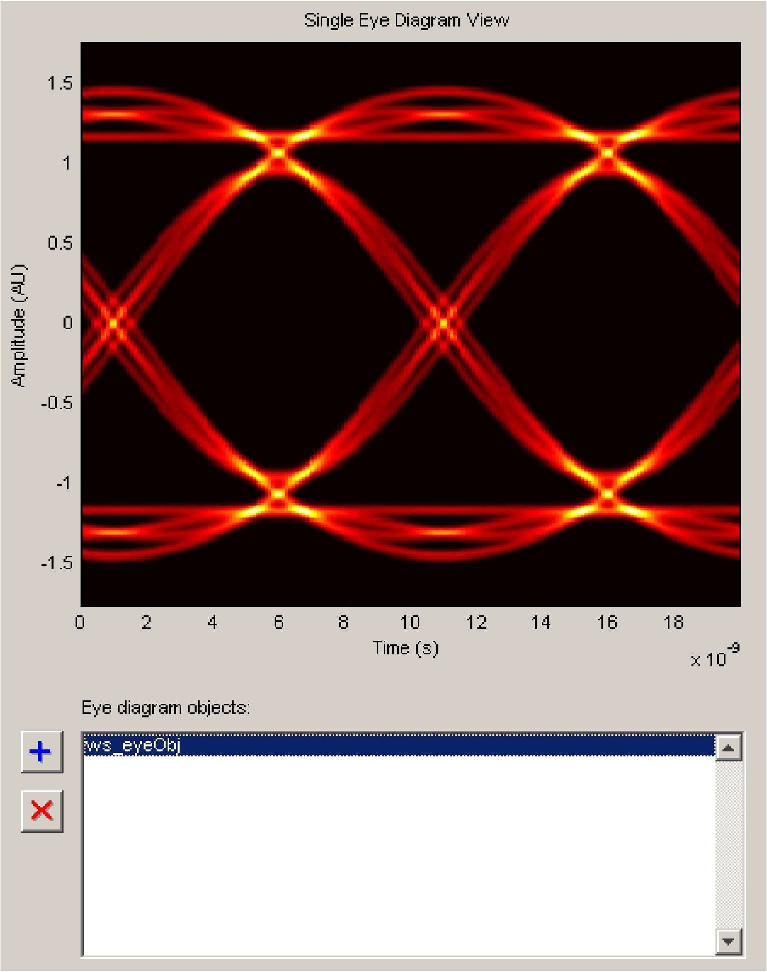
The EyeScope menu bar is comprised of four menus: File, Options, View, and Help.



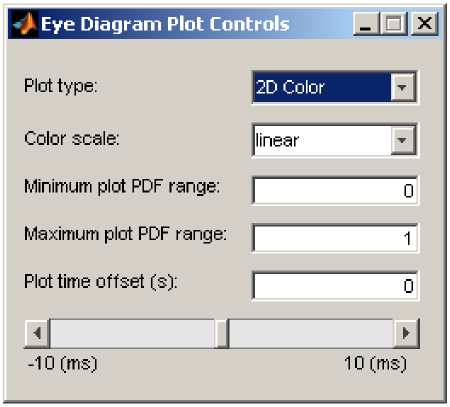
- Use the **File** menu to control the session management functions, import an eye diagram object into EyeScope, and export an eye diagram plot.
- Use the **Options** menu to setup the eye diagram scope by selecting which eye diagram settings and measurements EyeScope displays.
- Use the **View** menu 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**



Eye Diagram Plot Controls

Plot type: 2D Color

Color scale: linear

Minimum plot PDF range: 0

Maximum plot PDF range: 1

Plot time offset (s): 0

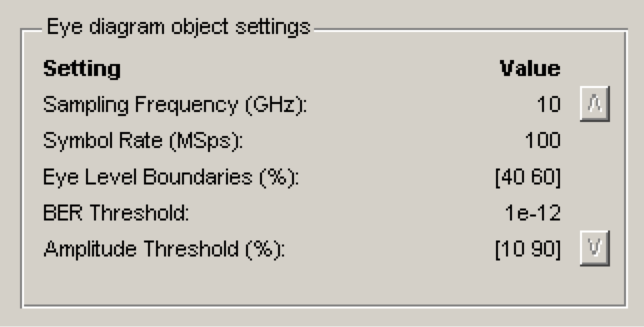
Note The value for the **Plot time offset** parameter can either be entered directly into the text box or set using the slide bar control.

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



Eye diagram object settings

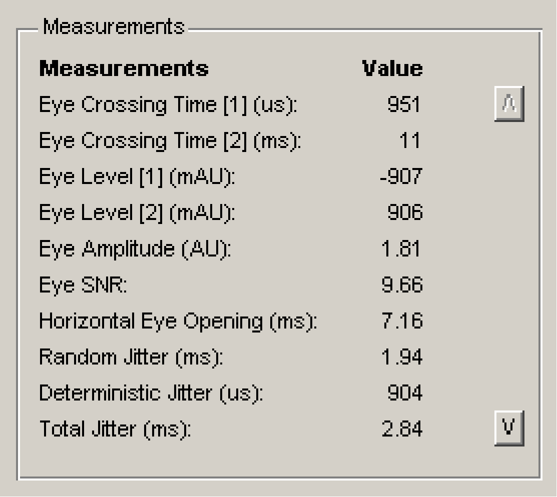
Setting	Value
Sampling Frequency (GHz):	10
Symbol Rate (MSps):	100
Eye Level Boundaries (%):	[40 60]
BER Threshold:	1e-12
Amplitude Threshold (%):	[10 90]

To specify which eye diagram object settings display in EyeScope, refer to "Selecting Which Eye Diagram Object Settings To Display" on page 1-492. 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



Measurements	Value
Eye Crossing Time [1] (us):	951
Eye Crossing Time [2] (ms):	11
Eye Level [1] (mAU):	-907
Eye Level [2] (mAU):	906
Eye Amplitude (AU):	1.81
Eye SNR:	9.66
Horizontal Eye Opening (ms):	7.16
Random Jitter (ms):	1.94
Deterministic Jitter (us):	904
Total Jitter (ms):	2.84

To select which eye diagram measurements EyeScope displays, refer to “Selecting Which Eye Diagram Measurements To Display” on page 1-493. 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 1-487
- “Starting a new Session” on page 1-488
- “Opening a Session” on page 1-488
- “Saving a Session” on page 1-489
- “Importing an Eye Diagram Object” on page 1-490
- “Printing to a Figure” on page 1-491
- “Selecting Which Eye Diagram Object Settings To Display” on page 1-492
- “Selecting Which Eye Diagram Measurements To Display” on page 1-493

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.

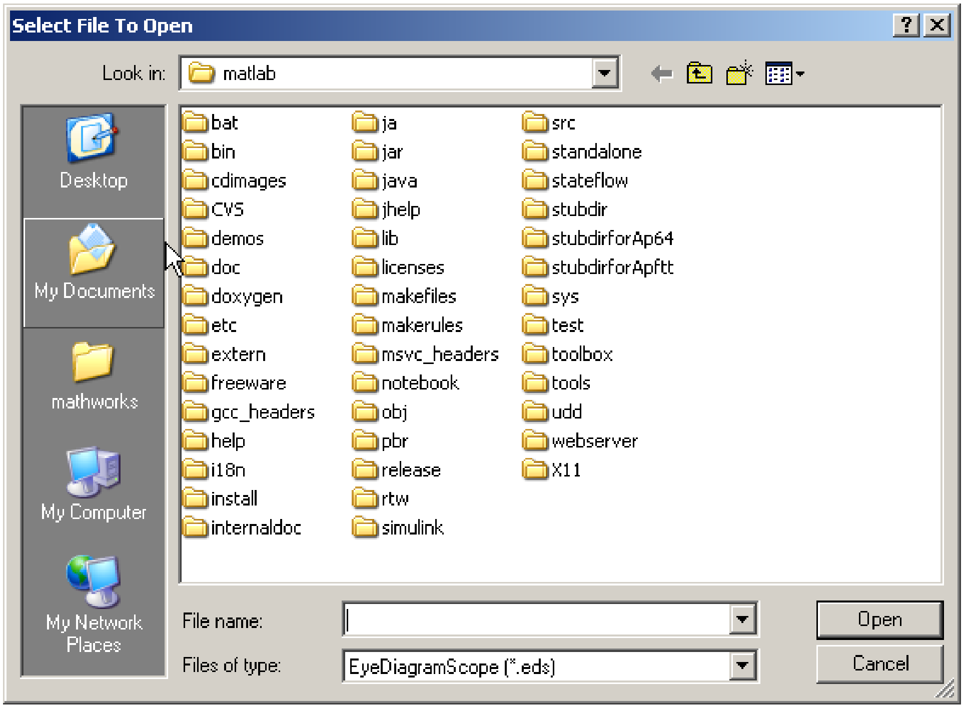
Opening a Session

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.



Select File To Open

Look in: matlab

File name:

Files of type: EyeDiagramScope (*.eds)

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:

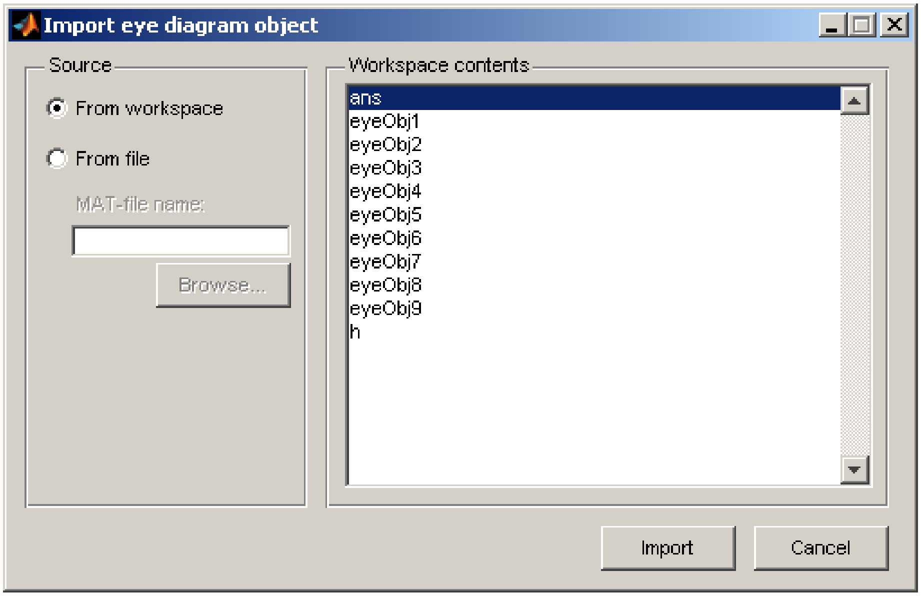
- If an object is imported from the workspace, the variable name will be *ws_object name*, where *object name* is the name of the original variable.
- If the object is imported from a MATLAB file, then the file name (without the path) precedes the object name.

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 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**.

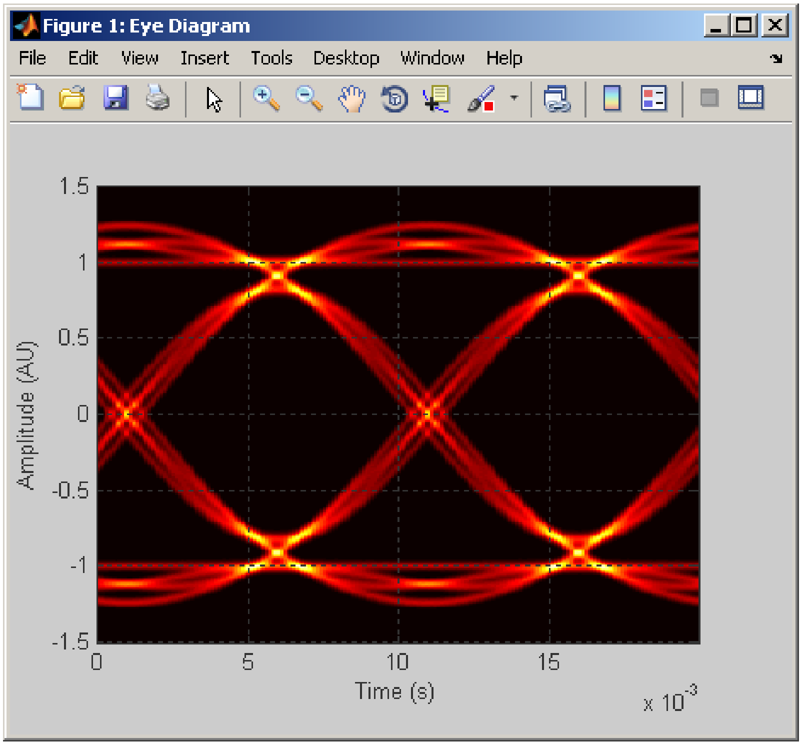
Printing to a Figure

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:

- Click **File** > **Print to Figure**

  The MATLAB figure window, containing the exported image, appears.



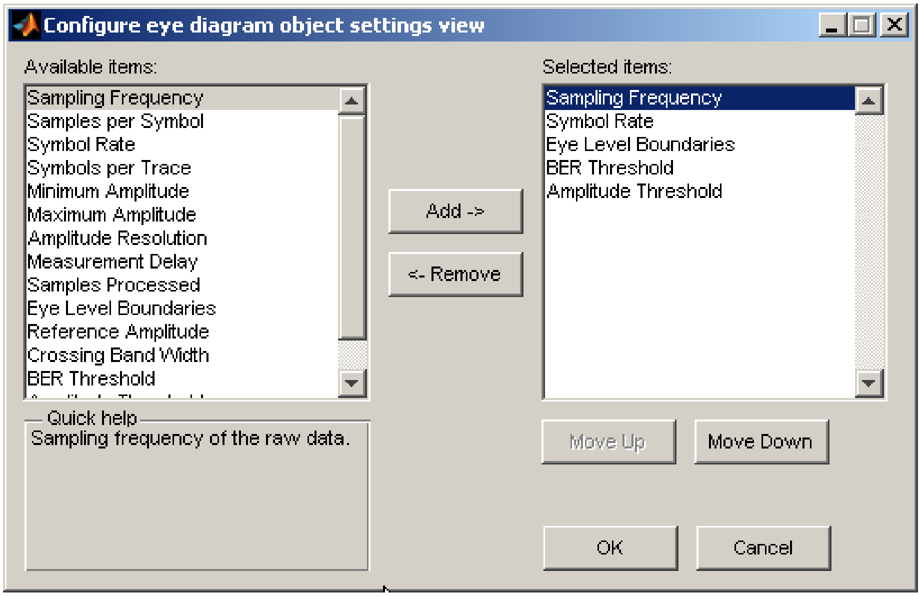
Selecting Which Eye Diagram Object Settings To Display

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.



Configure eye diagram object settings view

Available items:

Sampling Frequency
Samples per Symbol
Symbol Rate
Symbols per Trace
Minimum Amplitude
Maximum Amplitude
Amplitude Resolution
Measurement Delay
Samples Processed
Eye Level Boundaries
Reference Amplitude
Crossing Band Width
BER Threshold

Selected items:

Sampling Frequency
Symbol Rate
Eye Level Boundaries
BER Threshold
Amplitude Threshold

Quick help

Sampling frequency of the raw data.

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** .

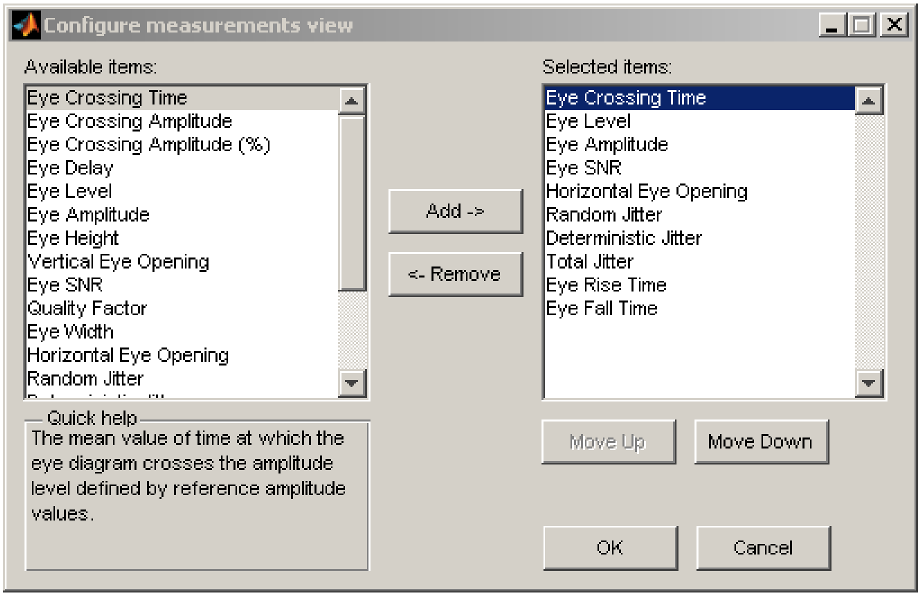
Selecting Which Eye Diagram Measurements To Display

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.



The mean value of time at which the eye diagram crosses the amplitude level defined by reference amplitude values.

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** .

See Also

`comm.EyeDiagram`

Introduced in R2008b

fft

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

Discrete Fourier Transform of Galois Vector

Set the order of the Galois field. Because `x` is in the Galois field (2^4), the length of `x` must be $2^m - 1$.

```
m = 4;
n = 2^m-1;
```

Generate a random GF vector.

```
x = gf(randi([0 2^m-1],n,1),m);
```

Perform the Fourier transform.

```
y = fft(x);
```

Invert the transform.

```
z = ifft(y);
```

Confirm that the inverse transform `z` = `x`.

```
isequal(z,x)

ans = logical
   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

dftmtx | gf | ifft

Topics

“Signal Processing Operations in Galois Fields”

Introduced before R2006a

filter (channel)

(To be removed) Filter signal with channel object

Syntax

```
y = filter(chan,x)
```

Note This function will be removed in a future release. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

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 `ricianchan`. 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`.

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

`comm.RayleighChannel` | `comm.RicianChannel`

Topics

"Fading Channels"

Introduced in R2007a

filter (Galois field)

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 shown here:

```
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

Filter a Galois Field

```
When using the Galois 1-D digital filter function, the data is
normalized by the first element of the denominator coefficient
vector.
```

```
a = gf([2 3 5 7], 3);
b = gf([1 3], 3);
x = gf(randi([0, 7], 10, 1), 3);
filt_x = filter(b, a, x)
filt_x = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

   6
   6
   3
   4
   7
   4
   2
   2
   0
   5
```

The first coefficient of the denominator coefficient vector, a(1) = 2. To confirm the function normalizes the data, manually normalize the filtered data. Using `isequal` to compare the outputs, we see they are equal.

```
filt_x2 = a(1) * filter(b/a(1), a, x)
```

```
filt_x2 = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

   6
   6
   3
   4
   7
   4
   2
   2
   0
   5
```

```
isequal(filt_x,filt_x2)
```

```
ans = logical
   1
```

See Also

`gf`

Introduced before R2006a

fmdemod

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

FM Modulate and Demodulate Sinusoidal Signal

Set the sample rate and carrier frequency. Generate a time vector having a duration of 0.2 s.

```
fs = 1000;
fc = 200;
t = (0:1/fs:0.2)';
```

Create a sinusoidal signal.

```
x = sin(2*pi*30*t)+2*sin(2*pi*60*t);
```

Set the frequency deviation to 50 Hz.

```
fDev = 50;
```

Frequency modulate x.

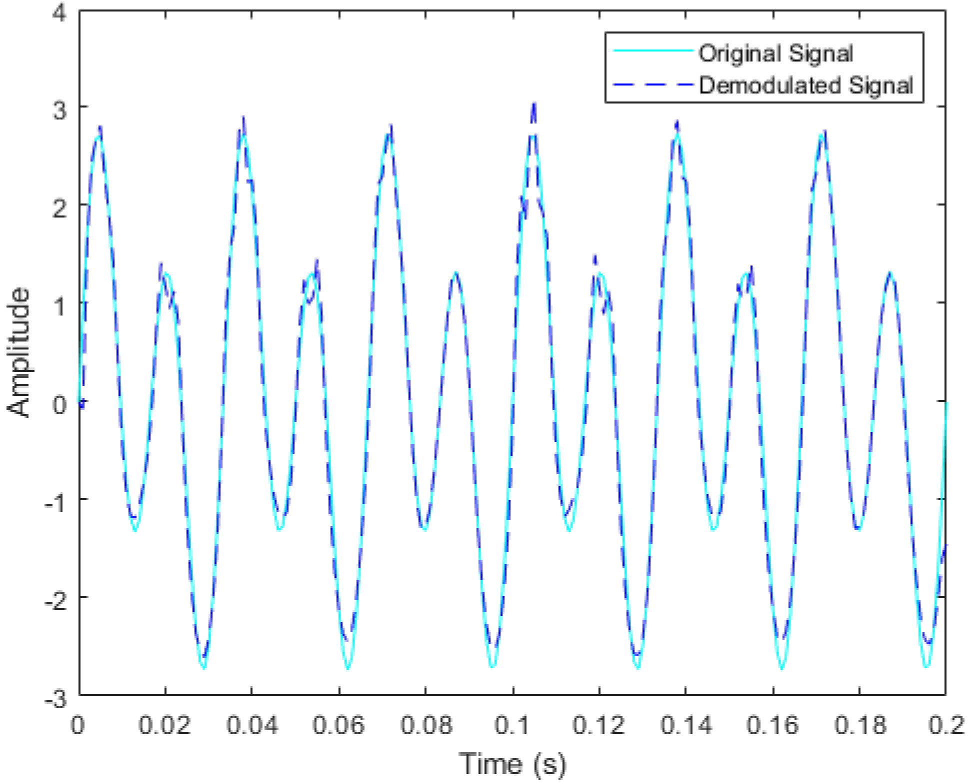
```
y = fmmod(x,fc,fs,fDev);
```

Demodulate z.

```
z = fmdemod(y,fc,fs,fDev); % Demodulate both channels.
```

Plot the original and demodulated signals.

```
plot(t,x,'c',t,z,'b--');
xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Demodulated Signal')
```



The demodulated signal is well aligned with the original.

See Also

`fmmod` | `pmdemod` | `pmmod`

Topics

“Digital Modulation”

Introduced before R2006a

fmmod

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

FM Modulate and Demodulate Sinusoidal Signal

Set the sample rate and carrier frequency. Generate a time vector having a duration of 0.2 s.

```
fs = 1000;
fc = 200;
t = (0:1/fs:0.2)';
```

Create a sinusoidal signal.

```
x = sin(2*pi*30*t)+2*sin(2*pi*60*t);
```

Set the frequency deviation to 50 Hz.

```
fDev = 50;
```

Frequency modulate `x`.

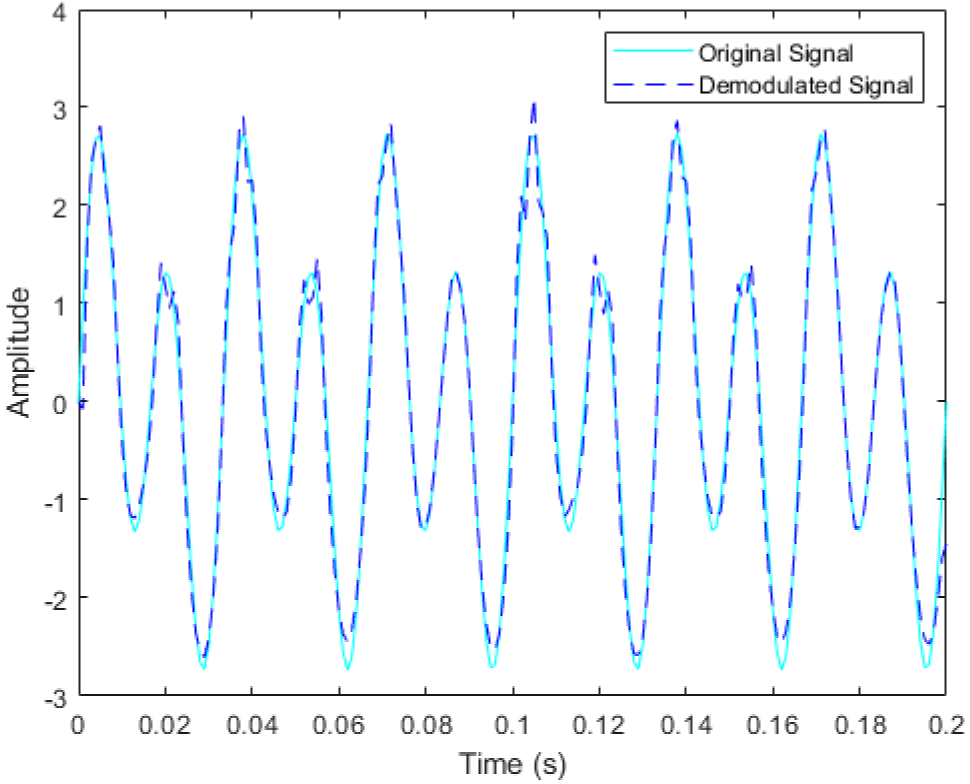
```
y = fmmod(x,fc,fs,fDev);
```

Demodulate `z`.

```
z = fmdemod(y,fc,fs,fDev); % Demodulate both channels.
```

Plot the original and demodulated signals.

```
plot(t,x,'c',t,z,'b--');
xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Demodulated Signal')
```



The demodulated signal is well aligned with the original.

See Also

ammod | fmdemod | pmmod

Topics

"Digital Modulation"

Introduced before R2006a

fogpl

RF signal attenuation due to fog and clouds

Syntax

```
L = fogpl(R,freq,T,den)
```

Description

`L = fogpl(R,freq,T,den)` returns attenuation, `L`, when signals propagate in fog or clouds. `R` represents the signal path length. `freq` represents the signal carrier frequency, `T` is the ambient temperature, and `den` specifies the liquid water density in the fog or cloud.

The `fogpl` function applies the International Telecommunication Union (ITU) cloud and fog attenuation model to calculate path loss of signals propagating through clouds and fog [1]. Fog and clouds are the same atmospheric phenomenon, differing only by height above ground. Both environments are parameterized by their liquid water density. Other model parameters include signal frequency and temperature. This function applies when the signal path is contained entirely in a uniform fog or cloud environment. The liquid water density does not vary along the signal path. The attenuation model applies only for frequencies at 10–1000 GHz.

Examples

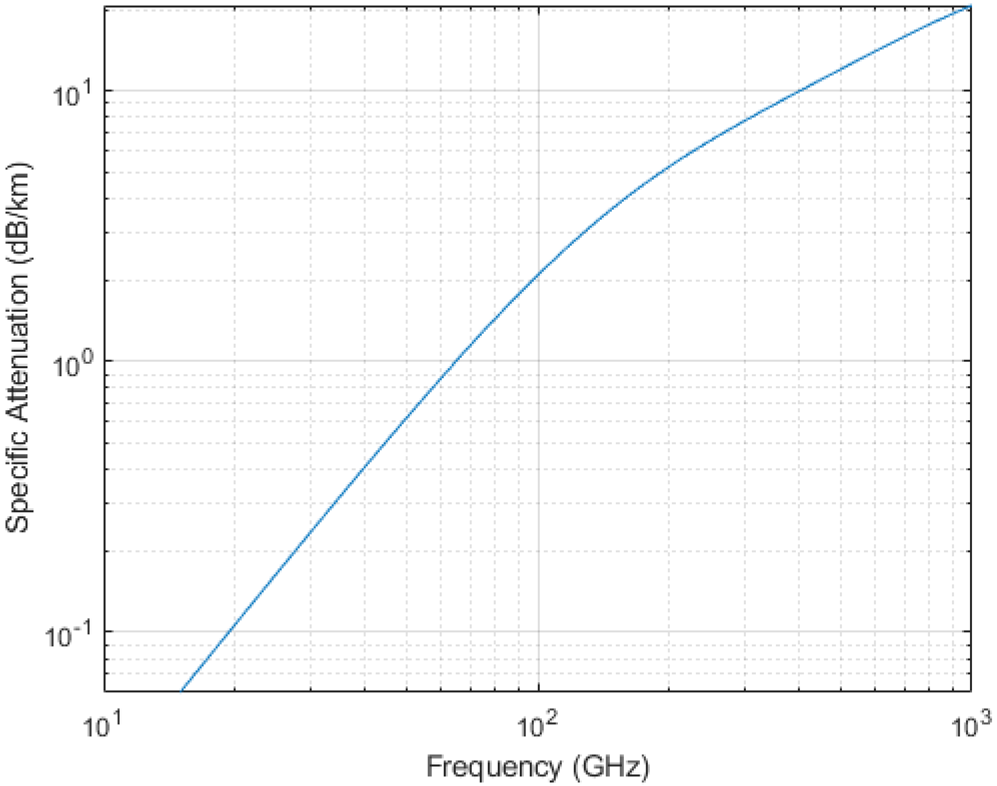
Attenuation in Cumulus Clouds

Compute the attenuation of signals propagating through a cloud that is 1 km long at 1000 meters altitude. Compute the attenuation for frequencies from 15 to 1000 GHz. A typical value for the cloud liquid water density is 0.5 g/m^3. Assume the atmospheric temperature at 1000 meters is 20°C.

```
R = 1000.0;
freq = [15:5:1000]*1e9;
T = 20.0;
lwd = 0.5;
L = fogpl(R,freq,T,lwd);
```

Plot the specific attenuation as a function of frequency. Specific attenuation is the attenuation or loss per kilometer.

```
loglog(freq/1e9,L)
grid
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB/km)')
```



Input Arguments

R — Signal path length

positive real-valued scalar | *M*-by-1 nonnegative real-valued vector | 1-by-*M* nonnegative real-valued vector

Signal path length, specified as a scalar or as an *M*-by-1 or 1-by-*M* vector of nonnegative real-values. Total attenuation is the specific attenuation multiplied by the path length. Units are meters.

Example: `[1300.0,1400.0]`

`freq` — Signal frequency

positive real-valued scalar | *N*-by-1 nonnegative real-valued column vector | 1-by-*N* nonnegative real-valued row vector

Signal frequency, specified as a positive real-valued scalar or as an *N*-by-1 nonnegative real-valued vector or 1-by-*N* nonnegative real-valued vector. Frequencies must lie in the range 10–1000 GHz.

Example: `[14.0e9,15.0e9]`

T — Ambient temperature

real-valued scalar

Ambient temperature in fog or cloud, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `-10.0`

den — Liquid water density

nonnegative real-valued scalar

Liquid water density, specified as a nonnegative real-valued scalar. Units are g/m^3. Typical values for liquid water density in fog range from approximately 0.05 g/m^3 for medium fog to approximately 0.5 g/m^3 for thick fog. For medium fog, visibility is about 50 meters. For heavy fog, visibility is about 300 meters. Cumulus cloud liquid water density is typically 0.5 g/m^3.

Example: `0.01`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

Definitions

Fog and Cloud Attenuation Model

This model calculates the attenuation of signals that propagate through fog or clouds.

Fog and cloud attenuation are the same atmospheric phenomenon. The ITU model, *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog* is used. The model computes the specific attenuation (attenuation per kilometer), of a signal as a function of liquid water density, signal frequency, and temperature. The model applies to polarized and nonpolarized fields. The formula for specific attenuation at each frequency is

$$\gamma_c = K_l(f)M,$$

where M is the liquid water density in gm/m^3. The quantity $K_l(f)$ is the specific attenuation coefficient and depends on frequency. The cloud and fog attenuation model is valid for frequencies 10–1000 GHz. Units for the specific attenuation coefficient are (dB/km)/(g/m^3).

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length R. Total attenuation is $L_c = R\gamma_c$.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply narrowband attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog*. 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fspl` | `gaspl` | `rainpl`

Introduced in R2016a

fspl

Free space path loss

Syntax

```
L = fspl(R,lambda)
```

Description

`L = fspl(R,lambda)` returns the free space path loss in decibels for a waveform with wavelength `lambda` propagated over a distance of `R` meters. The minimum value of `L` is zero, indicating no path loss.

Input Arguments

R

real-valued 1-by-*M* or *M*-by-1 vector

Propagation distance of signal. Units are in meters.

lambda

real-valued 1-by-*N* or *N*-by-1 vector

The wavelength is the speed of propagation divided by the signal frequency. Wavelength units are meters.

Output Arguments

L

Path loss in decibels. *M*-by-*N* nonnegative matrix. A value of zero signifies no path loss. When `lambda` is a scalar, `L` has the same dimensions as `R`.

Examples

Calculate Free-Space Path Loss

Calculate the free-space path loss (in dB) of a 10 GHz radar signal over a distance of 10 km.

```
fc = 10.0e9;
lambda = physconst('LightSpeed')/fc;
R = 10e3;
L = fspl(R,lambda)
```

```
L = 132.4478
```

Definitions

Free Space Path Loss

The free-space path loss, L, in decibels is:

$$L = 20\log_{10}(\frac{4\pi R}{\lambda})$$

This formula assumes that the target is in the far-field of the transmitting element or array. In the near-field, the free-space path loss formula is not valid and can result in a loss smaller than 0 dB, equivalent to a signal gain. For this reason, the loss is set to 0 dB for range values $R \leq \lambda/4\pi$.

References

[1] Proakis, J. *Digital Communications*. New York: McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fogpl` | `gaspl` | `rainpl`

Introduced in R2011a

gaspl

RF signal attenuation due to atmospheric gases

Syntax

```
L = gaspl(range,freq,T,P,den)
```

Description

`L = gaspl(range,freq,T,P,den)` returns the attenuation, `L`, when signals propagate through the atmosphere. `range` represents the signal path length, and `freq` represents the signal carrier frequency. `T` represents the ambient temperature, `P` represents the atmospheric pressure, and `den` represents the atmospheric water vapor density.

The `gaspl` function applies the International Telecommunication Union (ITU) atmospheric gas attenuation model [1] to calculate path loss for signals primarily due to oxygen and water vapor. The model computes attenuation as a function of ambient temperature, pressure, water vapor density, and signal frequency. The function requires that the signal path is contained entirely in a uniform environment. Atmospheric parameters do not vary along the signal path. The attenuation model applies only for frequencies at 1–1000 GHz.

Examples

Atmospheric Gas Attenuation Spectrum

Compute the attenuation spectrum from 1 to 1000 GHz for an atmospheric pressure of 101.300 kPa and a temperature of 15°C. Plot the spectrum for a water vapor density of 7.5 g/m^3 and then plot the spectrum for dry air (zero water vapor density).

Set the attenuation frequencies.

```
freq = [1:1000]*1e9;
```

Assume a 1 km path distance.

```
R = 1000.0;
```

Compute the attenuation for air containing water vapor.

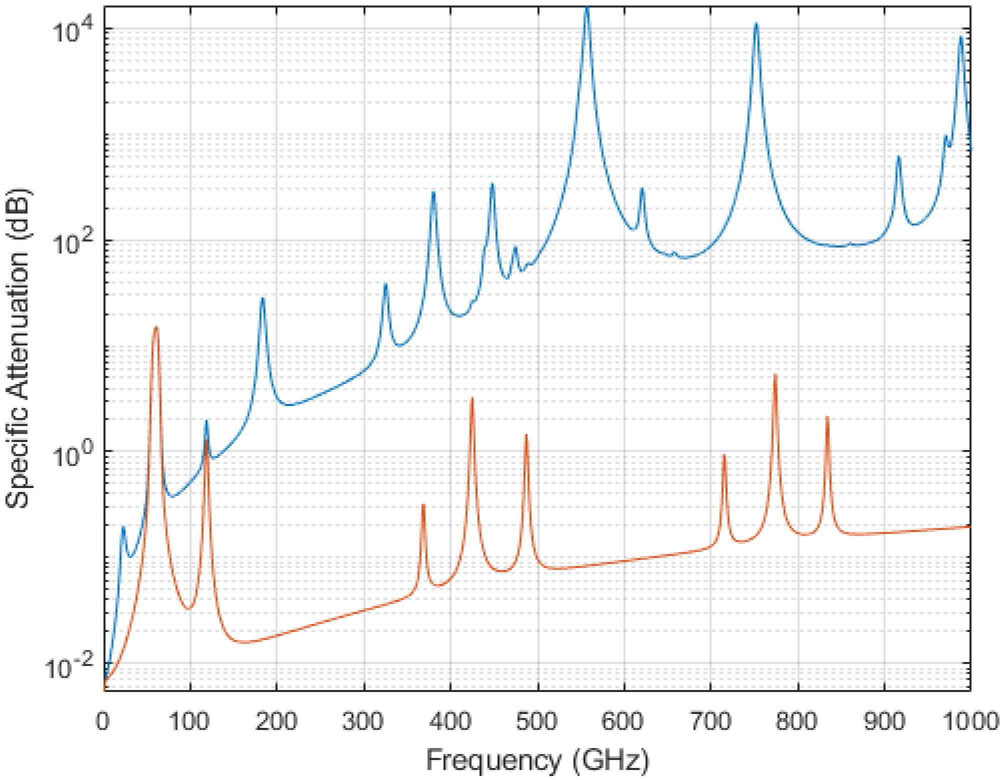
```
T = 15;
P = 101300.0;
W = 7.5;
L = gaspl(R,freq,T,P,W);
```

Compute the attenuation for dry air.

```
L0 = gaspl(R,freq,T,P,0.0);
```

Plot the attenuations.

```
semilogy(freq/1e9,L)
hold on
semilogy(freq/1e9,L0)
grid
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB)')
hold off
```



Plot Attenuation Due to Atmospheric Gases and Free Space

First, plot the specific attenuation of atmospheric gases for frequencies from 1 GHz to 1000 GHz. Assume a sea-level dry air pressure of 101.325e5 kPa and a water vapor density of 7.5 g/m^3. The air temperature is 20°C. Specific attenuation is defined as dB loss per kilometer. Then, plot the actual attenuation at 10 GHz for a span of ranges.

Plot Specific Atmospheric Gas Attenuation

Set the atmosphere temperature, pressure, water vapor density.

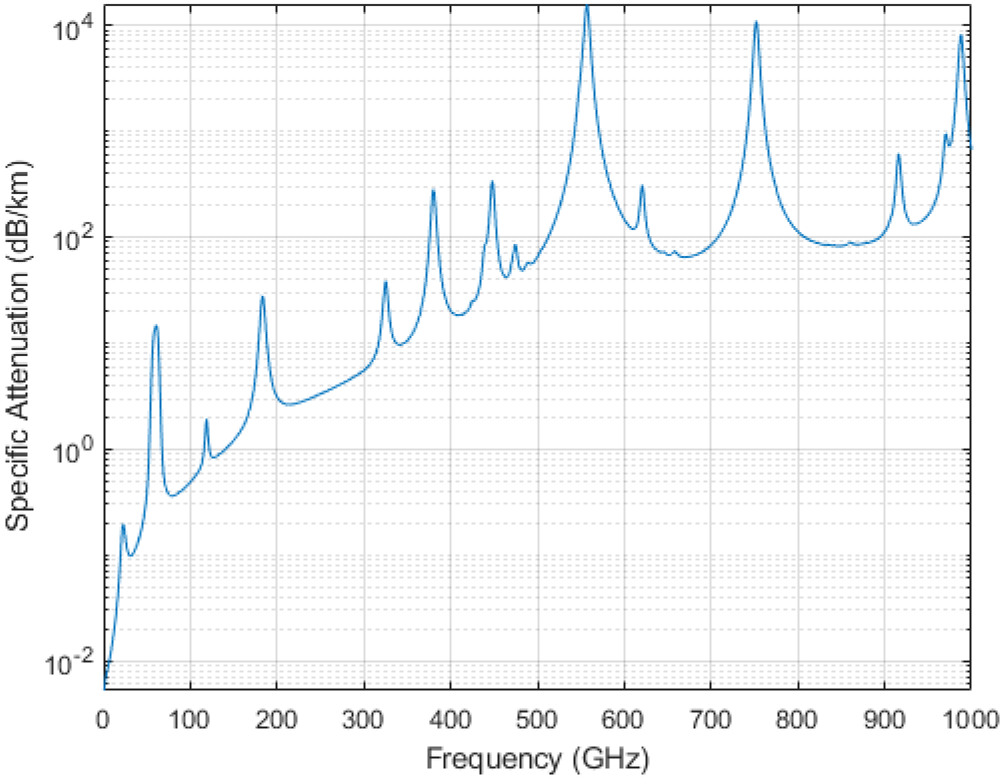
```
T = 20.0;
Patm = 101.325e3;
rho_wv = 7.5;
```

Set the propagation distance, speed of light, and frequencies.

```
km = 1000.0;
c = physconst('LightSpeed');
freqs = [1:1000]*1e9;
```

Compute and plot the atmospheric gas loss.

```
loss = gaspl(km,freqs,T,Patm,rho_wv);
semilogy(freqs/1e9,loss)
grid on
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB/km)')
```

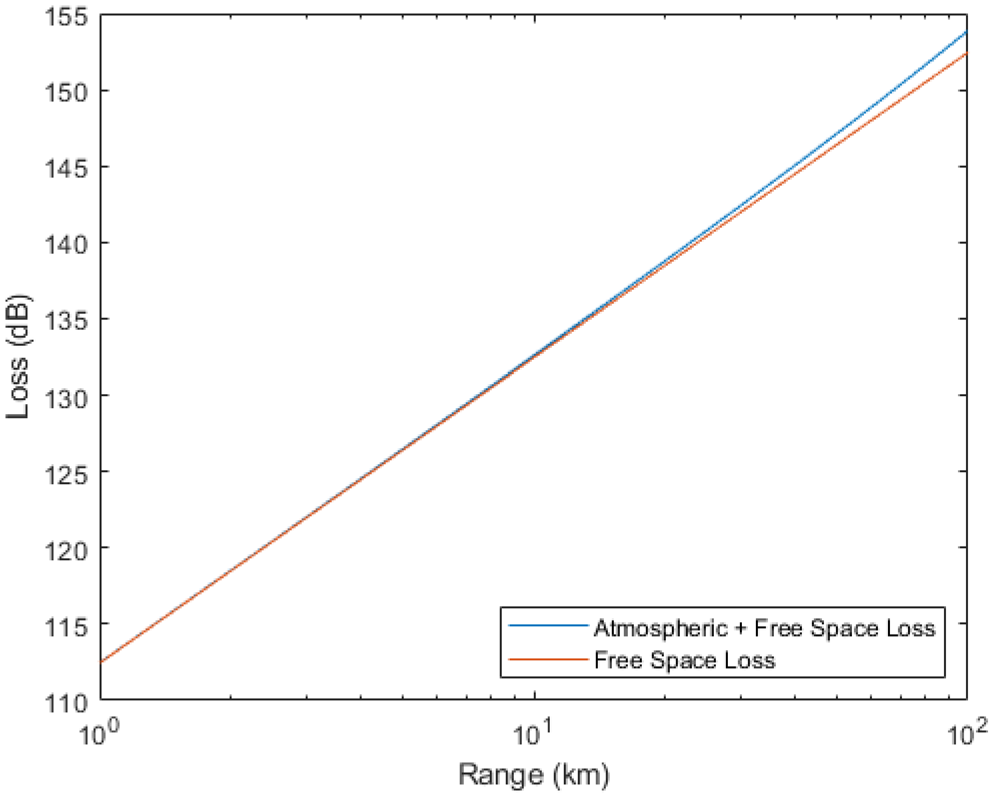


Plot Actual Atmospheric and Free Space Attenuation

Compute both free space loss and atmospheric gas loss at 10 GHz for ranges from 1 to 100 km. The frequency corresponds to an *X*-band radar. Then, plot the free space loss and the total (atmospheric + free space) loss.

```
ranges = [1:100]*1000;
freq_xband = 10e9;
loss_gas = gaspl(ranges,freq_xband,T,Patm,rho_wv);
lambda = c/freq_xband;
loss_fsp = fspl(ranges,lambda);
semilogx(ranges/1000,loss_gas + loss_fsp.',ranges/1000,loss_fsp)
legend('Atmospheric + Free Space Loss','Free Space Loss','Location','SouthEast')
```

```
xlabel('Range (km)')
ylabel('Loss (dB)')
```



Input Arguments

range — Signal path length

nonnegative real-valued scalar | *M*-by-1 nonnegative real-valued column vector | 1-by-*M* nonnegative real-valued row vector

Signal path length used to compute attenuation, specified as a nonnegative real-valued scalar or vector. You can specify multiple path lengths simultaneously. Units are in meters.

Example: `[13000.0,14000.0]`

`freq` — Signal frequency

positive real-valued scalar | *N*-by-1 nonnegative real-valued column vector | 1-by-*N* nonnegative real-valued row vector

Signal frequency, specified as a positive real-valued scalar, or as an *N*-by-1 nonnegative real-valued vector or 1-by-*N* nonnegative real-valued vector. You can specify multiple frequencies simultaneously. Frequencies must lie in the range 1–1000 GHz. Units are in hertz.

Example: `[1.4e9,2.0e9]`

`T` — Ambient temperature

real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `-10.0`

`P` — Ambient pressure

positive real-valued scalar

Ambient pressure, specified as a positive real-valued scalar. Units are in Pascals. One standard atmosphere at sea level is 101.325 kPa.

Example: `101300.0`

`den` — Water vapor density

nonnegative real-valued scalar

Water vapor density or absolute humidity, specified as a nonnegative real-valued scalar. Units are g/m^3. The maximum water vapor density of air at 30° C is approximately 30.0 g/m^3. The maximum water vapor density of air at 0°C is approximately 5.0 g/m^3.

Example: `4.0`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

Definitions

Atmospheric Gas Attenuation Model

This model calculates the attenuation of signals that propagate through atmospheric gases.

Electromagnetic signals attenuate when they propagate through the atmosphere. This effect is due primarily to the absorption resonance lines of oxygen and water vapor, with smaller contributions coming from nitrogen gas. The model also includes a continuous absorption spectrum below 10 GHz. The ITU model *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* is used. The model computes the specific attenuation (attenuation per kilometer) as a function of temperature, pressure, water vapor density, and signal frequency. The atmospheric gas model is valid for frequencies from 1–1000 GHz and applies to polarized and nonpolarized fields.

The formula for specific attenuation at each frequency is

$$\gamma = \gamma_o(f) + \gamma_w(f) = 0.1820 f N''(f).$$

The quantity *N*"*()* is the imaginary part of the complex atmospheric refractivity. It consists of a spectral line component and a continuous component:

$$N''(f) = \sum_i S_i F_i + N_D''(f)$$

The spectral component consists of a sum of discrete spectrum terms composed of a localized frequency bandwidth function, $F(f)_i$, multiplied by a spectral line strength, S_i. For atmospheric oxygen, each spectral line strength is

$$S_i = a_1 \times 10^{-7} \left(\frac{300}{T}\right)^3 \exp\left[a_2 (1 - \left(\frac{300}{T}\right)\right] P.$$

For atmospheric water vapor, each spectral line strength is

$$S_i = b_1 \times 10^{-1} \left(\frac{300}{T}\right)^{3.5} \exp\left[b_2 (1 - \left(\frac{300}{T}\right)\right] W.$$

P is the atmospheric pressure, *W* is the water vapor density, and *T* is the ambient temperature.

For each oxygen line, S_i depends on constants a_1 and a_2. Similarly, each water vapor line has constants b_1 and b_2. The ITU documentation cited at the end of this section contains tabulations of these constants.

The localized frequency bandwidth functions $F_i(f)$ are complicated functions of frequency described in the ITU references cited below. The functions depend on empirical model parameters that are also tabulated in the reference.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length, *R*. Then, the total attenuation is L_g= $R(\gamma_o + \gamma_w)$.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

fogpl | fspl | rainpl

Introduced in R2016a

fskdemod

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

Modulation and Demodulation of an FSK Signal in AWGN

Pass an FSK signal through an AWGN channel and estimate the resulting bit error rate (BER). Compare the estimated BER to the theoretical value.

Set the simulation parameters.

```
M = 2;          % Modulation order
k = log2(M);    % Bits per symbol
EbNo = 5;       % Eb/No (dB)
Fs = 16;        % Sample rate (Hz)
nsamp = 8;      % Number of samples per symbol
freqsep = 10;   % Frequency separation (Hz)
```

Generate random data symbols.

```
data = randi([0 M-1],5000,1);
```

Apply FSK modulation.

```
txsig = fskmod(data,M,freqsep,nsamp,Fs);
```

Pass the signal through an AWGN channel

```
rxSig  = awgn(txsig,EbNo+10*log10(k)-10*log10(nsamp),...
    'measured',[],'dB');
```

Demodulate the received signal.

```
dataOut = fskdemod(rxSig,M,freqsep,nsamp,Fs);
```

Calculate the bit error rate.

```
[num,BER] = biterr(data,dataOut);
```

Determine the theoretical BER and compare it to the estimated BER. Your BER value might vary because the example uses random numbers.

```
BER_theory = berawgn(EbNo,'fsk',M,'noncoherent');
[BER BER_theory]
```

```
ans = 1×2

    0.0958    0.1029
```

References

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

See Also

`fskmod` | `pskdemod` | `pskmod`

Topics

“Digital Modulation”

Introduced before R2006a

fskmod

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

FSK Signal Spectrum Plot

Generate an FSK modulated signal and display its spectral characteristics.

Set the function parameters.

```
M = 4;          % Modulation order
freqsep = 8;    % Frequency separation (Hz)
nsamp = 8;      % Number of samples per symbol
Fs = 32;        % Sample rate (Hz)
```

Generate random M-ary symbols.

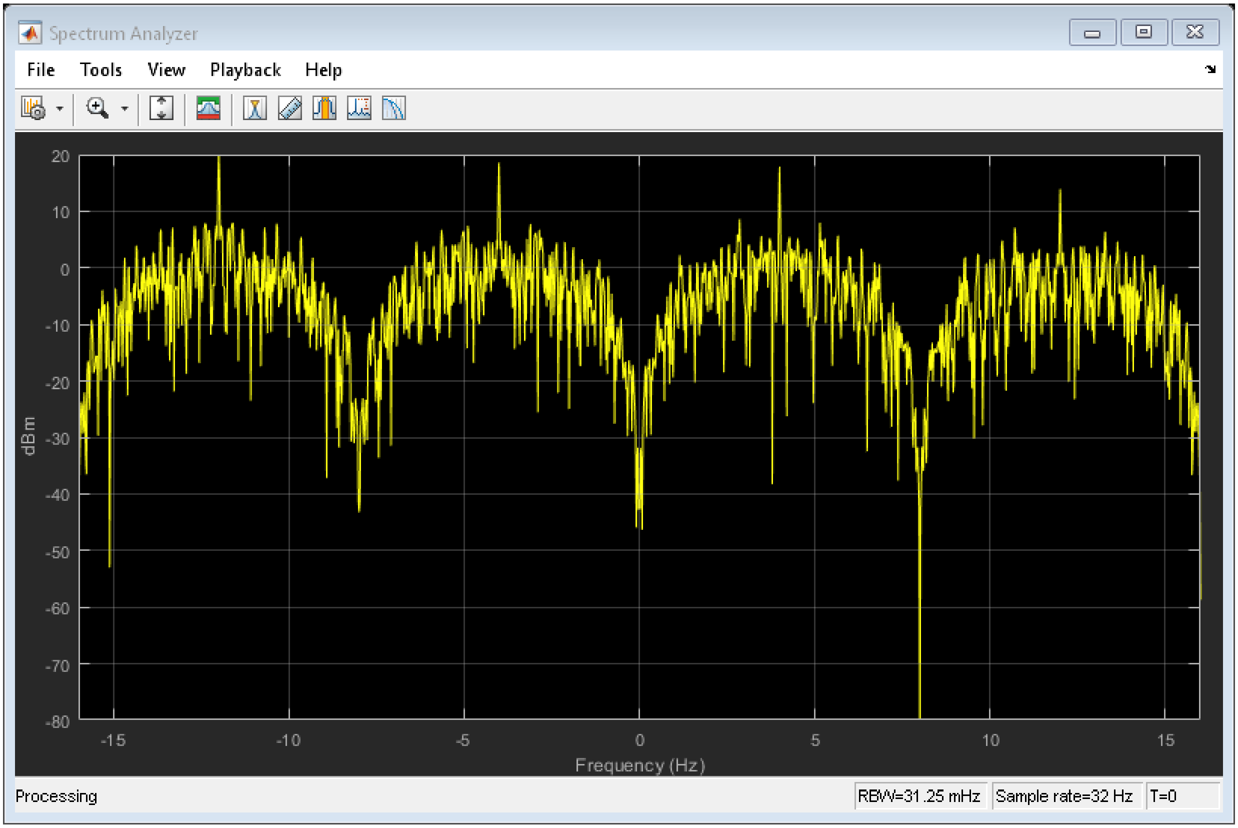
```
x = randi([0 M-1],1000,1);
```

Apply FSK modulation.

```
y = fskmod(x,M,freqsep,nsamp,Fs);
```

Create a spectrum analyzer System object™ and use its `step` method to display a plot of the signal spectrum.

```
h = dsp.SpectrumAnalyzer('SampleRate',Fs);
step(h,y)
```



References

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

See Also

`fskdemod` | `pskdemod` | `pskmod`

Topics

"Digital Modulation"

Introduced before R2006a

hex2poly

Convert hexadecimal character vector to binary coefficients

Syntax

```
b = hex2poly(hex)
b = hex2poly(hex,ord)
```

Description

`b = hex2poly(hex)` converts a hexadecimal character vector, `hex`, to a vector of binary coefficients, `b`.

`b = hex2poly(hex,ord)` specifies the power order, `ord`, of the coefficients that comprise the output. If omitted, `ord` is `'descending'`.

Examples

Convert Hexadecimal Polynomial to Binary Vector

Convert the hexadecimal polynomial `'1AF'` to a vector of binary coefficients. The coefficients represent the polynomial $x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$.

```
b = hex2poly('1AF')
```

```
b = 1×9

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

Convert Hexadecimal into Ascending Order Binary Vector

Convert hexadecimal `'0x82608EDB'` to a vector of binary coefficients. Specify that the binary coefficients are in ascending order.

```
b = hex2poly('0x82608EDB','ascending')
```

```
b = 1×32

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

The binary representation corresponds to a polynomial of $x^{31}+x^{25}+x^{22}+x^{21}+x^{15}+x^{11}+x^{10}+x^{9}+x^{7}+x^{6}+x^{4}+x^{3}+x+1$.

Input Arguments

hex — Hexadecimal number
character vector

Hexadecimal number, specified as a character vector.

Example: `'FF'`

Example: `'0x3FA'`

Data Types: `char`

`ord` — Power order
`'descending'` (default) | `'ascending'`

Power order of the vector of binary coefficients, specified as a character vector having a value of `'ascending'` or `'descending'`.

Data Types: `char`

Output Arguments

b — Binary coefficients
vector

Binary coefficients representing a polynomial, returned as a row vector having length equal to $p + 1$, where p is the order of hexadecimal input.

Data Types: `double`

See Also

`dec2hex` | `oct2poly`

Introduced in R2015b

oct2poly

Convert octal number to binary coefficients

Syntax

```
b = oct2poly(oct)
b = oct2poly(oct,ord)
```

Description

`b = oct2poly(oct)` converts an octal number, `oct`, to a vector of binary coefficients, `b`.

`b = oct2poly(oct,ord)` specifies the power order, `ord`, of the coefficients that comprise the output. If omitted, `ord` is `'descending'`.

Examples

Convert Octal Number to Binary Vector

Convert the octal number `11` to a binary vector.

```
b = oct2poly(11)
```

```
b = 1×4

     1     0     0     1
```

The binary vector corresponds to the polynomial $x^3 + 1$.

Convert Octal Number to Ascending Order Binary Vector

Convert the octal number 65 to an ascending order binary vector.

```
b = oct2poly(65,'ascending')
```

```
b = 1×6

     1     0     1     0     1     1
```

Sixty-five octal is the generator polynomial of a (15,10) Hamming code in the Bluetooth v4.0 standard. The binary representation of 65 octal is 110101 and the GF(2) polynomial is $1 + x^2 + x^4 + x^5$ or [1 0 1 0 1 1] in ascending powers.

Input Arguments

oct — Octal number
scalar

Octal number, specified as a positive integer scalar.

Example: 15

Example: 3177

Data Types: double

ord — Power order
'descending' (default) | 'ascending'

Power order of the binary coefficients vector, specified as a character vector having a value of 'ascending' or 'descending'.

Data Types: char

Output Arguments

b — Binary coefficients
vector

Binary coefficients representing a polynomial, returned as a row vector having length equal to $p + 1$, where p is the order of octal input.

Data Types: `double`

See Also

`bi2de` | `de2bi` | `hex2poly` | `oct2dec`

Introduced in R2015b

plotPhaseNoiseFilter

Plot response of phase noise filter block

Syntax

```
plotPhaseNoiseFilter(blockname)
```

Description

`plotPhaseNoiseFilter(blockname)` plots the response of the phase noise filter associated with the Phase Noise block specified by the variable `blockname`.

Examples

View Filter Response of Phase Noise Block

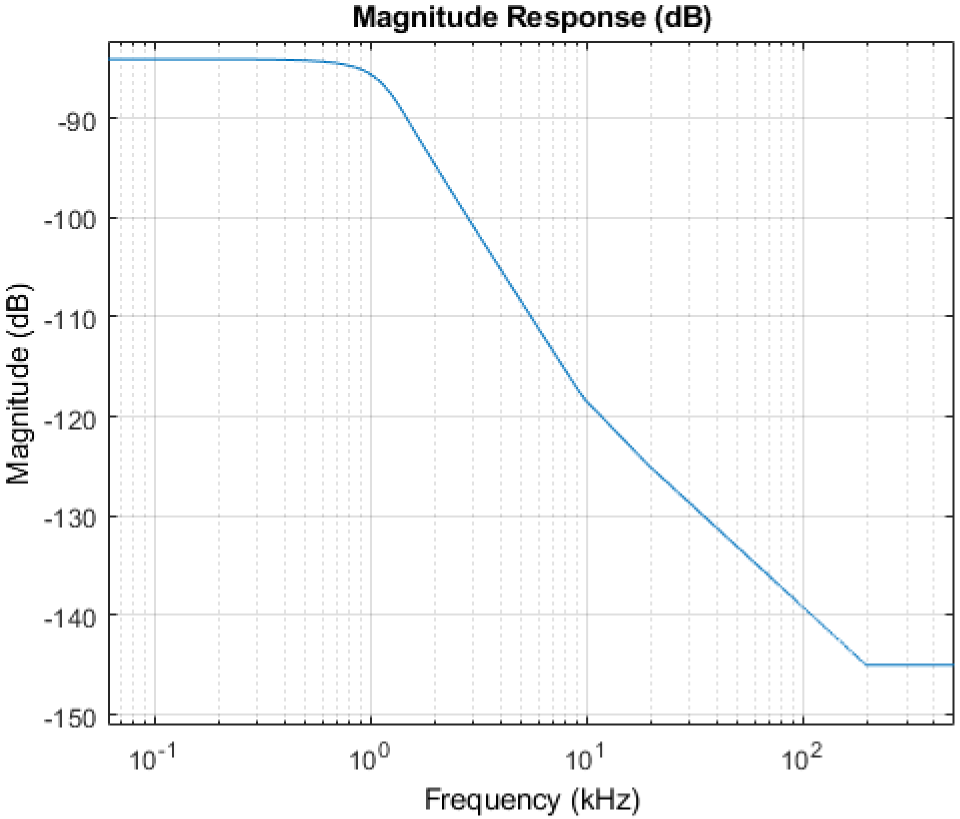
This example shows how to use the `plotPhaseNoiseFilter` function to view the filter response of a Phase Noise block in a Simulink model.

Load a Simulink model that contains a Phase Noise block. The `load_system` command loads a model into memory without making its model window visible. The function will also work with models whose window is visible. The example, `slex_phasenoise`, contains a `Phase Noise` block.

```
load_system('slex_phasenoise')
```

Run the `plotPhaseNoiseFilter` function to view the filter response of the block Phase Noise.

```
plotPhaseNoiseFilter('slex_phasenoise/Phase Noise')
```



Input Arguments

blockname — Phase noise block name

character vector

The name of a Phase Noise block in a Simulink model

Example: `plotPhaseNoiseFilter('Model Name/Phase Noise')`

Data Types: `char`

See Also

Phase Noise

Introduced in R2014b

gen2par

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

Convert Parity-Check Matrix for a Hamming Code to Generator Matrix

Convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

Create the parity-check matrix.

```
parmat = hammgen(3)
```

```
parmat = 3×7

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

Convert the parity-check matrix into the corresponding generator matrix.

```
genmat = gen2par(parmat)
```

```
genmat = 4×7

     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
```

Convert the generator matrix back again. The output, `parmat2`, should be the same as the original matrix, `parmat`.

```
parmat2 = gen2par(genmat)
```

```
parmat2 = 3×7

     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`

Topics

"Block Codes"

Introduced before R2006a

genqamdemod

General quadrature amplitude demodulation

Syntax

```
z = genqamdemod(y,const)
```

Description

`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

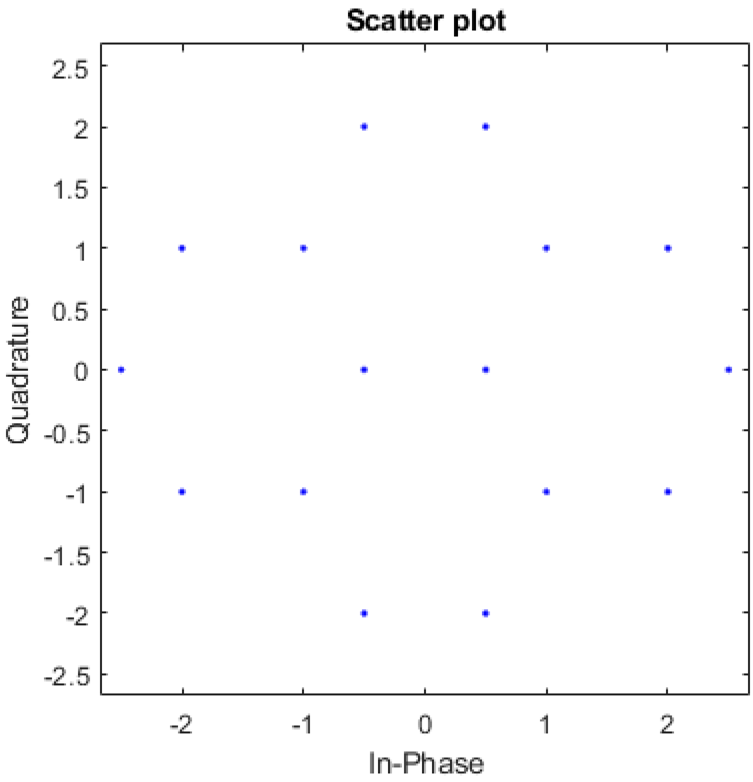
General QAM Modulation and Demodulation

Create the points that describe a 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 + 1i*quadr;
```

Plot the constellation.

```
h = scatterplot(const);
```



Generate input data symbols. Modulate the symbols using this constellation.

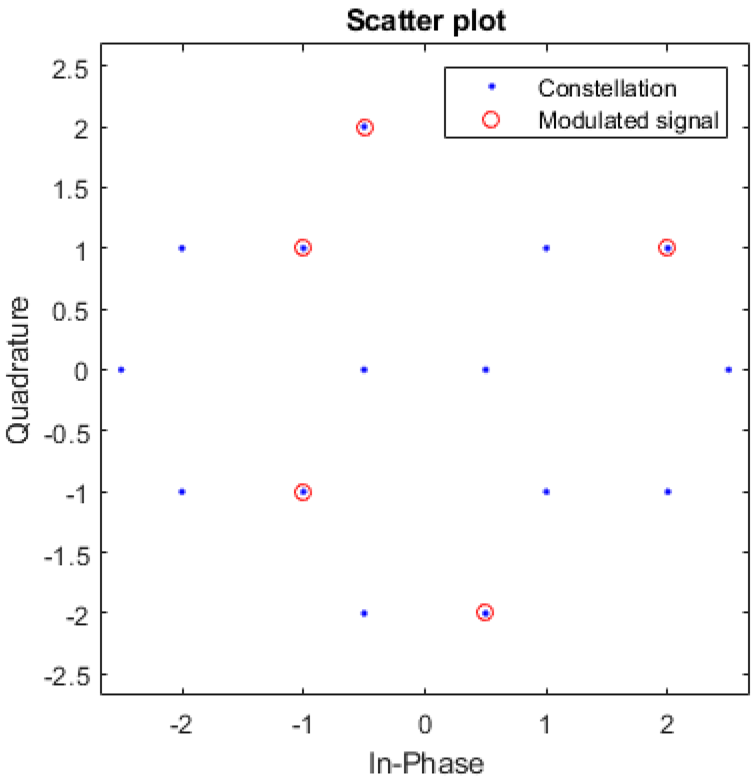
```
x = [3 8 5 10 7];
y = genqammod(x,const);
```

Demodulate the modulated signal, y.

```
z = genqamdemod(y,const);
```

Plot the modulated signal in same figure.

```
hold on;
scatterplot(y,1,0,'ro',h);
legend('Constellation','Modulated signal');
```



Determine the number of symbol errors between the demodulated data to the original sequence.

```
numErrs = symerr(x,z)
```

```
numErrs = 0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

genqammod | pamdemod | pammod | qamdemod | qammod

Topics

“Digital Modulation”

Introduced before R2006a

genqammod

General quadrature amplitude modulation

Syntax

```
y = genqammod(x,const)
```

Description

`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

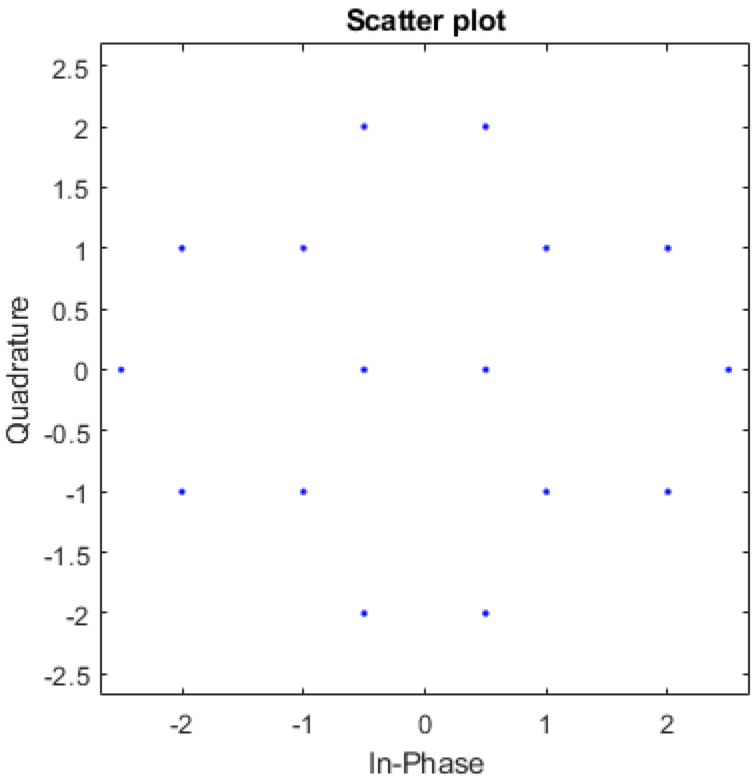
General QAM Modulation and Demodulation

Create the points that describe a 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 + 1i*quadr;
```

Plot the constellation.

```
h = scatterplot(const);
```



Generate input data symbols. Modulate the symbols using this constellation.

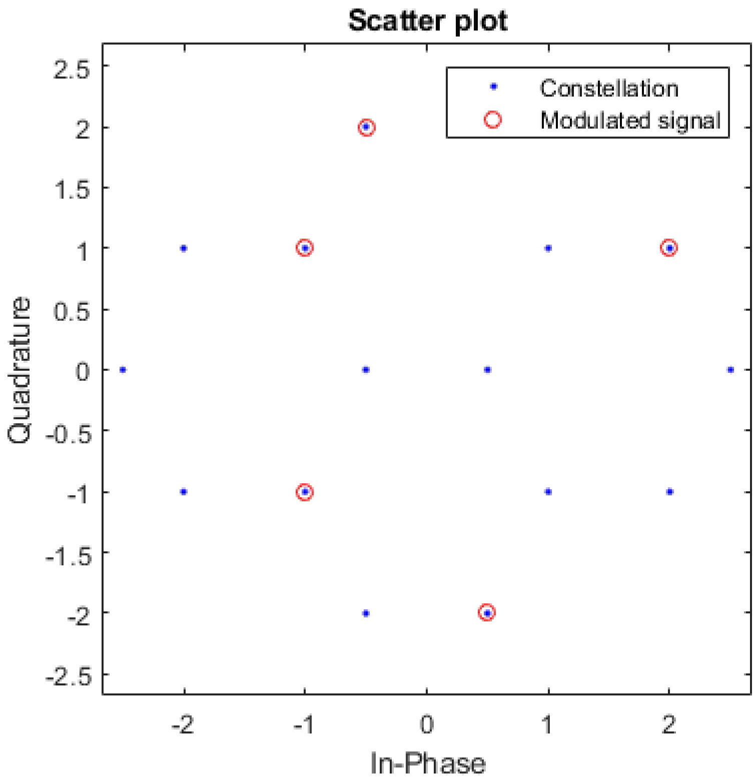
```
x = [3 8 5 10 7];
y = genqammod(x,const);
```

Demodulate the modulated signal, `y`.

```
z = genqamdemod(y,const);
```

Plot the modulated signal in same figure.

```
hold on;
scatterplot(y,1,0,'ro',h);
legend('Constellation','Modulated signal');
```



Determine the number of symbol errors between the demodulated data to the original sequence.

```
numErrs = symerr(x,z)
```

```
numErrs = 0
```

See Also

genqamdemod | pamdemod | pammod | qamdemod | qammod

Topics

"Digital Modulation"

Introduced before R2006a

gf

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 a polynomial character vector or 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

Create Sequence of GF(16) Elements

Set the order of the Galois field to 16, where the order equals 2^m. The elements of x must range from 0 to $2^m - 1$.

```
m = 4;
x = [3 2 9];
y = gf(x,m)
```

```
y = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements = 

   3   2   9
```

Create GF Sequence with Specified Primitive Polynomial

Create a sequence of integers. Create a Galois field array, where m = 5.

```
x = [17 8 11 27];
y = gf(x,5)
```

```
y = GF(2^5) array. Primitive polynomial = D^5+D^2+1 (37 decimal)

Array elements = 

   17    8   11   27
```

Determine all possible primitive polynomials for GF(2^5).

```
pp = primpoly(5,'all')
```

```
Primitive polynomial(s) = 

D^5+D^2+1
D^5+D^3+1
D^5+D^3+D^2+D^1+1
D^5+D^4+D^2+D^1+1
```

```
D^5+D^4+D^3+D^1+1
D^5+D^4+D^3+D^2+1

pp = 6×1

    37
    41
    47
    55
    59
    61
```

Generate a GF array using the primitive polynomial that has a decimal equivalent of 59.

```
z = gf(x,5,'D5+D4+D3+D+1')
```

```
z = GF(2^5) array. Primitive polynomial = D^5+D^4+D^3+D+1 (59 decimal)

Array elements = 

   17    8   11   27
```

Definitions

Galois Computations

Operations supported for Galois field arrays include:

Operation	Description
+ -	Addition and subtraction of Galois arrays
* / \	Matrix multiplication and division of Galois arrays
.* ./ .\	Elementwise multiplication and division of Galois arrays
^	Matrix exponentiation of Galois array
.^	Elementwise exponentiation of Galois array

Operation	Description
'.'	Transpose of Galois array
==, ~=	Relational operators for Galois arrays
all	True if all elements of a Galois vector are nonzero
any	True if any element of a Galois vector is nonzero
conv	Convolution of Galois vectors
convmtx	Convolution matrix of Galois field vector
deconv	Deconvolution and polynomial division
det	Determinant of square Galois matrix
dftmtx	Discrete Fourier transform matrix in a Galois field
diag	Diagonal Galois matrices and diagonals of a Galois matrix
fft	Discrete Fourier transform
filter (gf)	One-dimensional digital filter over a Galois field
ifft	Inverse discrete Fourier transform
inv	Inverse of Galois matrix
length	Length of Galois vector
log	Logarithm in a Galois field
lu	Lower-Upper triangular factorization of Galois array
minpol	Find the minimal polynomial for a Galois element
mldivide	Matrix left division \ of Galois arrays
polyval	Evaluate polynomial in Galois field
rank	Rank of a Galois array
reshape	Reshape Galois array
roots	Find polynomial roots across a Galois field

Operation	Description
size	Size of Galois array
tril	Extract lower triangular part of Galois array
triu	Extract upper triangular part of Galois array

See Also

`cosets` | `gftable` | `isprimitive` | `primpoly`

Topics

"Galois Field Computations"
"Error Detection and Correction"
"ElGamal Public Key Cryptosystem"

Introduced before R2006a

gfadd

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`, which can be either polynomial character vectors or numeric vectors. 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

Add Two GF Arrays

Sum $2+3x+x^2$ and $4+2x+3x^2$ over GF(5).

```
x = gfadd([2 3 1],[4 2 3],5)
```

```
x =

     1     0     4
```

Add the two polynomials and display the first two elements.

```
y = gfadd([2 3 1],[4 2 3],5,2)
```

```
y =

     1     0
```

For prime number `p` and exponent `m`, create a matrix listing all elements of GF(p^m) given primitive polynomial $2+2x+x^2$.

```
p = 3;
m = 2;
primpoly = [2 2 1];
field = gftuple((-1:p^m-2)',primpoly,p);
```

Sum A^2 and A^4. The result is A.

```
g = gfadd(2,4,field)
```

```
g =

     1
```

See Also

`gfconv` | `gfdeconv` | `gfdiv` | `gfmul` | `gfsub` | `gftuple`

Topics

“Arithmetic in Galois Fields”

Introduced before R2006a

gfconv

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, which can be either polynomial character vectors or numeric vectors. 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

Multiply Polynomials Over Galois Field

Multiply $1 + x + x^4$ and $x + x^2$ over GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)
```

```
gfc = 1×7

     0     1     2     1     0     1     1
```

The result corresponds to $x + 2x^2 + x^3 + x^5 + x^6$.

See Also

gfadd | gfdeconv | gfmul | gfsub | gftuple

Introduced before R2006a

gfcosets

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 1-566.

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`

Introduced before R2006a

gfdeconv

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). `a` and `b` can be either polynomial character vectors or numeric vectors.

`[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 1-584 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

`gfadd` | `gfconv` | `gfdiv` | `gfsub` | `gftuple`

Introduced before R2006a

gfdiv

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

`gfconv` | `gfdeconv` | `gfmul` | `gftuple`

Introduced before R2006a

gffilter (prime Galois field)

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.

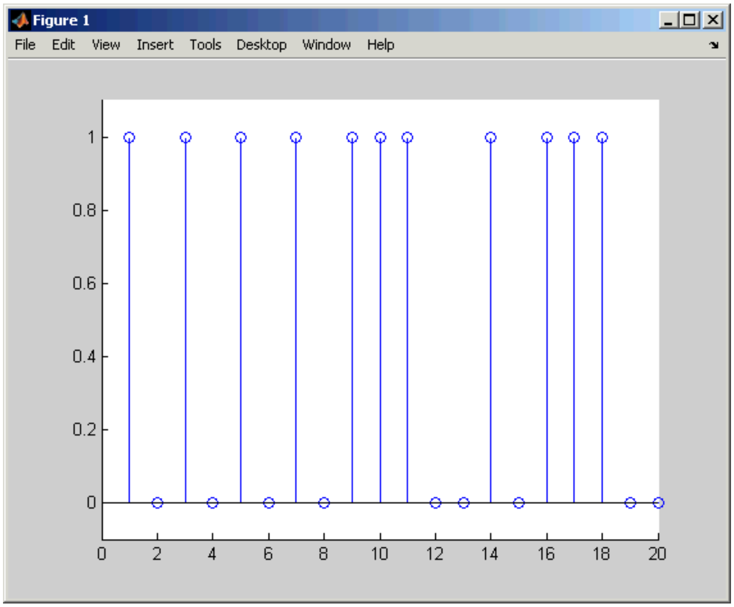
The vector a represents the degree-n_a polynomial

```
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

filter | gfadd | gfconv

Introduced before R2006a

gflineq

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

`conv` | `gfadd` | `gfconv` | `gfdiv` | `gfrank` | `gfroots`

Introduced before R2006a

gfminpol

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 polynomial character vector or 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

`gfcosets` | `gfprimdf` | `gfroots`

Introduced before R2006a

gfmul

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

gfadd | gfdeconv | gfdiv | gfsub | gftuple

Introduced before R2006a

gfpretty

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 `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 `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

gfprimdf | gftuple

Introduced before R2006a

gfprimck

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* in “Primitive Polynomials and Element Representations”.

`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.

`a` is either a polynomial character vector or a row vector representing the polynomial by listing its coefficients in ascending order. For example, in GF(5), `'4 + 3x + 2x^3'` and `[4 3 0 2]` are equivalent.

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

`gfadd` | `gfminpol` | `gfprimdf` | `gfprimfd` | `gftuple`

Introduced before R2006a

gfprimdf

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* in "Primitive Polynomials and Element Representations".

`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

gfminpol | gfprimck | gfprimfd | gftuple

Introduced before R2006a

gfprimfd

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* in “Primitive Polynomials and Element Representations”.

- 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

`gfminpol` | `gfprimck` | `gfprimdf` | `gftuple`

Introduced before R2006a

gfrank

Compute rank of matrix over Galois field

Syntax

```
rk = gfrank(A,p)
```

Description

Note This function performs computations in GF(p) 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.

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
```

Algorithms

`gfrank` uses an algorithm similar to Gaussian elimination.

Introduced before R2006a

gfrepcov

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.

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`

Introduced before R2006a

gfroots

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 polynomial character vector or 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

Introduced before R2006a

gfsub

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. Alternatively, `a` and `b` can be represented as polynomial character vectors.

`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

Subtract Two GF Arrays

Calculate $(2+3x+x^2)-(4+2x+3x^2)$ over GF(5).

```
x = gfsub([2 3 1],[4 2 3],5)
```

```
x = 1×3

     3     1     3
```

Subtract the two polynomials and display the first two elements.

```
y = gfsub([2 3 1],[4 2 3],5,2)
```

```
y = 1×2

     3     1
```

For prime number `p` and exponent `m`, create a matrix listing all elements of GF(p^m) given primitive polynomial $2+2x+x^2$.

```
p = 3;
m = 2;
primpoly = [2 2 1];
field = gftuple((-1:p^m-2)',primpoly,p);
```

Subtract A^4 from A^2. The result is A^7.

```
g = gfsub(2,4,field)
```

```
g = 7
```

See Also

`gfadd` | `gfconv` | `gfdeconv` | `gfdiv` | `gfmul` | `gftuple`

Introduced before R2006a

gftable

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`, which can be either a polynomial character vector or an integer. `prim_poly` represents a primitive polynomial for GF(`2^m`), where 1 < *m* < 16, 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`

Topics

“Speed and Nondefault Primitive Polynomials”

Introduced before R2006a

gftrunc

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` | `gfconv` | `gfdeconv` | `gfsub` | `gftuple`

Introduced before R2006a

gftuple

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 polynomial character vector or 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 character vector; 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` | `gfconv` | `gfdeconv` | `gfdiv` | `gfmul` | `gfprimdf`

Introduced before R2006a

gfweight

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 polynomial character vector or a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples

Calculate Minimum Distance of Linear Block Code

Calculate the minimum distance of a cyclic code using several methods.

Create the generate polynomial for a (7,4) cyclic code.

```
n = 7;
genpoly = cyclpoly(n,4);
```

Calculate the minimum distance for the cyclic code using:

1. Generator polynomial `genmat`
2. Parity check matrix `parmat`
3. Generator polynomial `genpoly`
4. Generator polynomial specified as a character vector

```
[parmat, genmat] = cyclgen(n,genpoly);
wts = [gfweight(genmat,'gen') gfweight(parmat,'par'),...
    gfweight(genpoly,n) gfweight('1+x2+x3',n)]
```

```
wts = 1×4

     3     3     3     3
```

See Also

`bchgenpoly` | `cyclpoly` | `hammgen`

Topics

“Block Codes”

Introduced before R2006a

gray2bin

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, matrix, or 3-D array. `modulation` is the modulation type and must be `'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

Binary to Gray Symbol Mapping

This example shows how to use the `bin2gray` and `gray2bin` functions to map integer inputs from a natural binary order symbol mapping to a Gray coded signal constellation and vice versa, assuming 16-QAM modulation. In addition, a visual representation of the difference between Gray and binary coded symbol mappings is shown.

Create a complete vector of 16-QAM integers.

```
x = (0:15)';
```

Convert the input vector from a natural binary order to a Gray encoded vector using `bin2gray`.

```
[y,mapy] = bin2gray(x,'qam',16);
```

Convert the Gray encoded symbols, `y`, back to a binary ordering using `gray2bin`.

```
z = gray2bin(y,'qam',16);
```

Verify that the original data, `x`, and the final output vector, `z` are identical.

```
isequal(x,z)
```

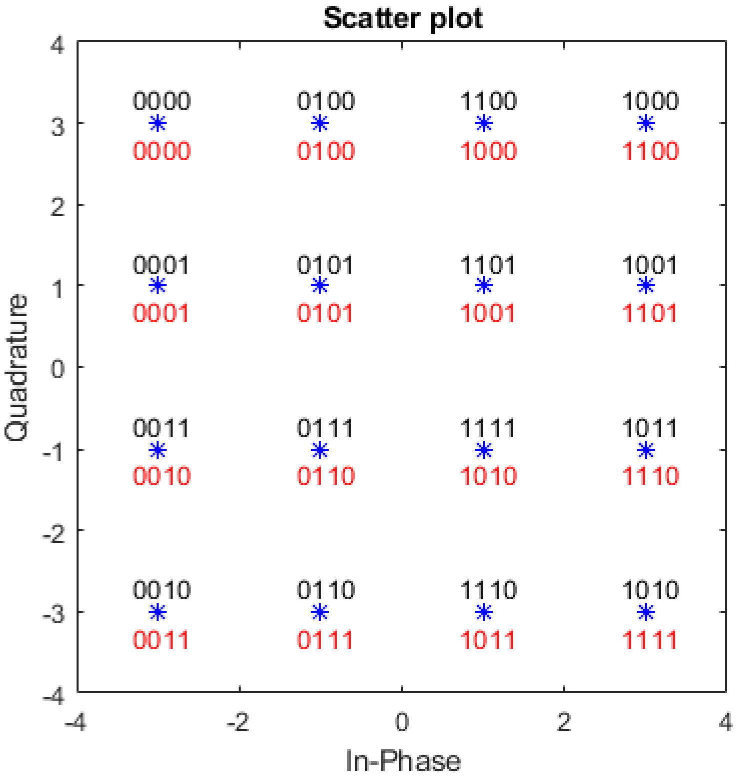
```
ans = logical
   1
```

To create a constellation plot showing the different symbol mappings, construct a 16-QAM modulator System object and use its associated `constellation` function to find the complex symbol values.

```
hMod = comm.RectangularQAMModulator;
symbols = constellation(hMod);
```

Plot the constellation symbols and label them using the Gray, `y`, and binary, `z`, output vectors. The binary representation of the Gray coded symbols is shown in black while the binary representation of the naturally ordered symbols is shown in red. Set the axes so that all points are displayed.

```
scatterplot(symbols,1,0,'b*');

for k = 1:16
    text(real(symbols(k))-0.3,imag(symbols(k))+0.3,...
        dec2base(mapy(k),2,4));

    text(real(symbols(k))-0.3,imag(symbols(k))-0.3,...
        dec2base(z(k),2,4),'Color',[1 0 0]);
end

axis([-4 4 -4 4])
```



Observe that only a single bit differs between adjacent constellation points when using Gray coding.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bin2gray`

Introduced before R2006a

hammgen

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 2. 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 2. 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 2. 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. Alternatively, you can specify `pol` as a polynomial character vector. `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

Generate Hamming Code Parity Check Matrices

Generate Hamming code matrices given codeword length.

Generate the parity check matrix `h`, the generator matrix `g`, the codeword length `n`, and the message length `k` for the Hamming code with $m = 3$.

```
[h,g,n,k] = hammgen(3)
```

```
h = 3×7

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


g = 4×7

     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
```

Generate the parity check matrices for $m = 4$ for the primitive polynomials $D^4 + D + 1$ and $D^4 + D^3 + 1$.

```
h1 = hammgen(4,'D^4+D+1');
h2 = hammgen(4,'D^4+D^3+1');
```

Remove the embedded 4-by-4 identity matrices (leftmost columns of both `h1` and `h2`) and verify that the two matrices differ.

```
h1(:,5:end)
```

```
ans = 4×11

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

```
h2(:,5:end)
```

```
ans = 4×11

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

`decode` | `encode` | `gen2par`

Topics

"Block Codes"

Introduced before R2006a

hank2sys

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

`hankel`

Introduced before R2006a

heldeintrlv

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`

Topics

“Interleaving”

Introduced before R2006a

helintrlv

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 matrix has been transposed 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`

Topics

"Interleaving"

Introduced before R2006a

helscandeintrlv

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`

Topics

“Interleaving”

Introduced before R2006a

helscanintrlv

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`

Topics

“Interleaving”

Introduced before R2006a

hilbiir

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 1-629 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 1-629.

[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.

Examples

For an example using the function's default values, type one of the following commands at the MATLAB prompt.

```
hilbiir
[num,den] = hilbiir
```

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].

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1980.

See Also

`grpdelay`

Introduced before R2006a

huffmandeco

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

Huffman Encoding and Decoding

Create unique symbols, and assign probabilities of occurrence to them.

```
symbols = 1:6;
p = [.5 .125 .125 .125 .0625 .0625];
```

Create a Huffman dictionary based on the symbols and their probabilities.

```
dict = huffmandict(symbols,p);
```

Generate a vector of random symbols.

```
sig = randsrc(100,1,[symbols; p]);
```

Encode the random symbols.

```
comp = huffmanenco(sig,dict);
```

Decode the data. Verify that the decoded data matches the original data.

```
dsig = huffmandeco(comp,dict);
isequal(sig,dsig)
```

```
ans = logical
   1
```

Convert the original signal to binary, and determine its length.

```
binarySig = de2bi(sig);
seqLen = numel(binarySig)
```

```
seqLen = 300
```

Convert the Huffman encoded signal and determine its length.

```
binaryComp = de2bi(comp);
encodedLen = numel(binaryComp)
```

```
encodedLen = 224
```

The Huffman encoded data required 224 bits, which is a 25% savings over the uncoded data.

Huffman Encoding and Decoding with Alphanumeric Signal

Define an alphanumeric signal in cell array form.

```
          sig = {'a2', 44, 'a3', 55, 'a1'}
```

```
sig = 1x5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

Define a dictionary. Codes for signal letters must be numeric.

```
          dict = {'a1',0; 'a2',[1,0]; 'a3',[1,1,0]; 44,[1,1,1,0]; 55,[1,1,1,1]}
```

```
dict = 5x2 cell array
    {'a1'}    {[       0]}
    {'a2'}    {1x2 double}
    {'a3'}    {1x3 double}
    {[44]}    {1x4 double}
    {[55]}    {1x4 double}
```

Encode the alphanumeric symbols.

```
comp = huffmanenco(sig,dict);
```

Decode the data and verify that the decoded data matches the original data.

```
dsig = huffmandeco(comp,dict)
```

```
dsig = 1x5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

```
isequal(sig,dsig)
```

```
ans = logical
   1
```

References

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

See Also

`huffmandict` | `huffmanenco`

Topics

“Huffman Coding”

Introduced before R2006a

huffmandict

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

Generate Huffman Code and View Return Values

Generate a binary Huffman code dictionary. Assign the second output to return the average code length.

Specify symbol alphabet and probability vectors.

```
symbols = (1:5); % Alphabet vector
prob = [.3 .3 .2 .1 .1]; % Symbol probability vector
```

Generate binary Huffman code. View the average code length and the cell array containing the codeword dictionary.

```
[dict, avglen] = huffmandict(symbols,prob);
avglen
```

```
avglen = 2.2000
```

```
dict
```

```
dict = 5x2 cell array
    {[1]}    {1x2 double}
    {[2]}    {1x2 double}
    {[3]}    {1x2 double}
    {[4]}    {1x3 double}
    {[5]}    {1x3 double}
```

View the fifth codeword from the dictionary.

```
samplecode = dict{5,2} % Codeword for fifth signal value
```

```
samplecode = 1×3

     1     1     0
```

Generate Ternary Huffman Codes

Use the code dictionary generator for Huffman coder function to generate binary and ternary Huffman codes.

Specify symbol alphabet and probability vectors

```
symbols = (1:5); % Alphabet vector
prob = [.3 .3 .2 .1 .1]; % Symbol probability vector
```

Generate binary Huffman code.

```
[dict, avglen] = huffmandict(symbols, prob);
dict(:, 2) = cellfun(@num2str, dict(:, 2), 'UniformOutput', false)
```

```
dict = 5x2 cell array
    {[1]}    {'0  1'   }
    {[2]}    {'0  0'   }
    {[3]}    {'1  0'   }
    {[4]}    {'1  1  1'}
    {[5]}    {'1  1  0'}
```

Generate ternary Huffman code

```
[dict,avglen] = huffmandict(symbols,prob, 3);
dict(:, 2) = cellfun(@num2str, dict(:, 2), 'UniformOutput', false)
```

```
dict = 5x2 cell array
    {[1]}    {'2'   }
    {[2]}    {'1'   }
    {[3]}    {'0  0'}
    {[4]}    {'0  2'}
    {[5]}    {'0  1'}
```

References

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

See Also

`huffmandeco` | `huffmanenco`

Topics

“Huffman Coding”

Introduced before R2006a

huffmanenco

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

Huffman Encoding and Decoding

Create unique symbols, and assign probabilities of occurrence to them.

```
symbols = 1:6;
p = [.5 .125 .125 .125 .0625 .0625];
```

Create a Huffman dictionary based on the symbols and their probabilities.

```
dict = huffmandict(symbols,p);
```

Generate a vector of random symbols.

```
sig = randsrc(100,1,[symbols; p]);
```

Encode the random symbols.

```
comp = huffmanenco(sig,dict);
```

Decode the data. Verify that the decoded data matches the original data.

```
dsig = huffmandeco(comp,dict);
isequal(sig,dsig)
```

```
ans = logical
   1
```

Convert the original signal to binary, and determine its length.

```
binarySig = de2bi(sig);
seqLen = numel(binarySig)
```

```
seqLen = 300
```

Convert the Huffman encoded signal and determine its length.

```
binaryComp = de2bi(comp);
encodedLen = numel(binaryComp)
```

```
encodedLen = 224
```

The Huffman encoded data required 224 bits, which is a 25% savings over the uncoded data.

Huffman Encoding and Decoding with Alphanumeric Signal

Define an alphanumeric signal in cell array form.

```
          sig = {'a2', 44, 'a3', 55, 'a1'}
```

```
sig = 1x5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

Define a dictionary. Codes for signal letters must be numeric.

```
          dict = {'a1',0; 'a2',[1,0]; 'a3',[1,1,0]; 44,[1,1,1,0]; 55,[1,1,1,1]}
```

```
dict = 5x2 cell array
    {'a1'}    {[       0]}
    {'a2'}    {1x2 double}
    {'a3'}    {1x3 double}
    {[44]}    {1x4 double}
    {[55]}    {1x4 double}
```

Encode the alphanumeric symbols.

```
comp = huffmanenco(sig,dict);
```

Decode the data and verify that the decoded data matches the original data.

```
dsig = huffmandeco(comp,dict)
```

```
dsig = 1x5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

```
isequal(sig,dsig)
```

```
ans = logical
   1
```

References

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

See Also

`huffmandeco` | `huffmandict`

Topics

"Huffman Coding"

Introduced before R2006a

ifft

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

`dftmtx` | `fft` | `gf`

Topics

“Signal Processing Operations in Galois Fields”

Introduced before R2006a

intdump

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`

Introduced before R2006a

intrlv

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`

Topics

“Interleaving”

Introduced before R2006a

iqcoef2imbal

Convert compensator coefficient to amplitude and phase imbalance

Syntax

```
[A,P] = iqcoef2imbal(C)
```

Description

`[A,P] = iqcoef2imbal(C)` converts compensator coefficient `C` to its equivalent amplitude and phase imbalance.

Examples

Estimate I/Q Imbalance from Compensator Coefficient

Use `iqcoef2imbal` to estimate the amplitude and phase imbalance for a given complex coefficient. The coefficients are an output from the `step` function of the `IQImbalanceCompensator`.

Create a QAM modulator and a raised cosine transmit filter to generate a 64-QAM signal.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',64);
hTxFilter = comm.RaisedCosineTransmitFilter;
```

Modulate and filter random 64-ary symbols.

```
data = randi([0 63],100000,1);
dataMod = step(hMod,data);
txSig = step(hTxFilter,dataMod);
```

Specify amplitude and phase imbalance.

```
ampImb = 2; % dB
phImb = 15; % degrees
```

Apply the specified I/Q imbalance.

```
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Normalize the power of the received signal

```
rxSig = rxSig/std(rxSig);
```

Remove the I/Q imbalance using the `comm.IQImbalanceCompensator` System object. Set the compensator object such that the complex coefficients are made available as an output argument.

```
hIQComp = comm.IQImbalanceCompensator('CoefficientOutputPort',true);
[compSig,coef] = step(hIQComp,rxSig);
```

Estimate the imbalance from the last value of the compensator coefficient.

```
[ampImbEst,phImbEst] = iqcoef2imbal(coef(end));
```

Compare the estimated imbalance values with the specified ones. Notice that there is good agreement.

```
[ampImb phImb; ampImbEst phImbEst]
```

```
ans = 2×2

    2.0000   15.0000
    2.0178   14.5740
```

Input Arguments

C — Compensator coefficient

complex-valued scalar or vector

Coefficient used to compensate for an I/Q imbalance, specified as a complex-valued vector.

Example: `0.4+0.6i`

Example: `[0.1+0.2i; 0.3+0.5i]`

Data Types: `single` | `double`

Output Arguments

`A` — Amplitude imbalance
real-valued vector

Amplitude imbalance in dB, returned as a real-valued vector with the same dimensions as `C`.

`P` — Phase imbalance
real-valued vector

Phase imbalance in degrees, returned as a real-valued vector with the same dimensions as `C`.

Definitions

I/Q Imbalance Compensation

The function `iqcoef2imbal` is a supporting function for the `comm.IQImbalanceCompensator` System object.

Given a scaling and rotation factor, G, compensator coefficient, C, and received signal, x, the compensated signal, y, has the form

$$y = G\left[x + C\,\mathrm{conj}(x)\right].$$

In matrix form, this can be rewritten as

$$\mathbf{Y} = \mathbf{R}\mathbf{X}\ ,$$

where **X** is a 2-by-1 vector representing the imbalanced signal [X_I, X_Q] and **Y** is a 2-by-1 vector representing the compensator output [Y_I, Y_Q].

The matrix **R** is expressed as

$$\mathbf{R} = \begin{bmatrix} 1+\mathrm{Re}\{C\} & \mathrm{Im}\{C\} \\ \mathrm{Im}\{C\} & 1-\mathrm{Re}\{C\} \end{bmatrix}$$

For the compensator to perfectly remove the I/Q imbalance, **R** = **K**$^{-1}$ because $\mathbf{X = KS}$, where **K** is a 2-by-2 matrix whose values are determined by the amplitude and phase imbalance and **S** is the ideal signal. Define a matrix **M** with the form

$$\mathbf{M} = \begin{bmatrix} 1 & -\alpha \\ \alpha & 1 \end{bmatrix}$$

Both **M** and **M**$^{-1}$ can be thought of as scaling and rotation matrices that correspond to the factor *G*. Because **K** = **R**$^{-1}$, the product **M**$^{-1}$ **R K M** is the identity matrix, where **M**$^{-1}$ **R** represents the compensator output and **K M** represents the I/Q imbalance. The coefficient *α* is chosen such that

$$\mathbf{KM} = L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\cos(\theta_Q) \\ I_{gain}\sin(\theta_I) & Q_{gain}\sin(\theta_Q) \end{bmatrix}$$

where *L* is a constant. From this form, we can obtain I_{gain}, Q_{gain}, θ_I, and θ_Q. For a given phase imbalance, Φ_{Imb}, the in-phase and quadrature angles can be expressed as

$$\theta_I = -(\pi/2)(\Phi_{Imb}/180)$$
$$\theta_Q = \pi/2 + (\pi/2)(\Phi_{Imb}/180)$$

Hence, cos(θ_Q) = sin(θ_I) and sin(θ_Q) = cos(θ_I) so that

$$L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\cos(\theta_Q) \\ I_{gain}\sin(\theta_I) & Q_{gain}\sin(\theta_Q) \end{bmatrix} = L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\sin(\theta_I) \\ I_{gain}\sin(\theta_I) & Q_{gain}\cos(\theta_I) \end{bmatrix}$$

The I/Q imbalance can be expressed as

$$\begin{aligned} \mathbf{KM} &= \begin{bmatrix} K_{11}+\alpha K_{12} & -\alpha K_{11}+K_{12} \\ K_{21}+\alpha K_{22} & -\alpha K_{21}+K_{22} \end{bmatrix} \\ &= L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\sin(\theta_I) \\ I_{gain}\sin(\theta_I) & Q_{gain}\cos(\theta_I) \end{bmatrix} \end{aligned}$$

Therefore,

$$\left(K_{21}+\alpha K_{22}\right)/\left(K_{11}+\alpha K_{12}\right)=\left(-\alpha K_{11}+K_{12}\right)/\left(-\alpha K_{21}+K_{22}\right)=\sin(\theta_I)/\cos(\theta_I)$$

The equation can be written as a quadratic equation to solve for the variable α, that is $D_1\alpha^2 + D_2\alpha + D_3 = 0$, where

$$D_1 = -K_{11}K_{12} + K_{22}K_{21}$$
$$D_2 = K_{12}^2 + K_{21}^2 - K_{11}^2 - K_{22}^2$$
$$D_3 = K_{11}K_{12} - K_{21}K_{22}$$

When $|C| \leq 1$, the quadratic equation has the following solution:

$$\alpha = \frac{-D_2 - \sqrt{D^2 - 4D_1D_3}}{2D_1}$$

Otherwise, when $|C| > 1$, the solution has the following form:

$$\alpha = \frac{-D_2 + \sqrt{D^2 - 4D_1D_3}}{2D_1}$$

Finally, the amplitude imbalance, A_{Imb}, and the phase imbalance, Φ_{Imb}, are obtained.

$$\mathbf{K}' = \mathbf{K}\begin{bmatrix} 1 & -\alpha \\ \alpha & 1 \end{bmatrix}$$
$$A_{Imb} = 20\log_{10}\left(K'_{11} / K'_{22}\right)$$
$$\Phi_{Imb} = -2\tan^{-1}\left(K'_{21} / K'_{11}\right)(180/\pi)$$

Note

- If C is real and $|C| \leq 1$, the phase imbalance is 0 and the amplitude imbalance is $20\log_{10}((1\text{–}C)/(1+C))$
- If C is real and $|C| > 1$, the phase imbalance is 180° and the amplitude imbalance is $20\log_{10}((C+1)/(C-1))$.

- If C is imaginary, $A_{Imb} = 0$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.IQImbalanceCompensator` | `iqimbal2coef`

Introduced in R2014b

iqimbal2coef

Convert I/Q imbalance to compensator coefficient

Syntax

```
C = iqimbal2coef(A,P)
```

Description

`C = iqimbal2coef(A,P)` converts an I/Q amplitude and phase imbalance to its equivalent compensator coefficient.

Examples

Generate Coefficients for I/Q Imbalance Compensation

Generate coefficients for the I/Q imbalance compensator System object™ using `iqimbal2coef`. The compensator corrects for an I/Q imbalance using the generated coefficients.

Create a QAM modulator and a raised cosine transmit filter to generate a 64-QAM signal.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',64);
hTxFilter = comm.RaisedCosineTransmitFilter;
```

Modulate and filter random 64-ary symbols.

```
data = randi([0 63],100000,1);
dataMod = step(hMod,data);
txSig = step(hTxFilter,dataMod);
```

Specify amplitude and phase imbalance.

```
ampImb = 2; % dB
phImb = 15; % degrees
```

Apply the specified I/Q imbalance.

```
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Normalize the power of the received signal.

```
rxSig = rxSig/std(rxSig);
```

Remove the I/Q imbalance by creating and applying a `comm.IQImbalanceCompensator` object. Set the compensator such that the complex coefficients are made available as an output argument.

```
hIQComp = comm.IQImbalanceCompensator('CoefficientOutputPort',true);
[compSig,coef] = step(hIQComp,rxSig);
```

Compare the final compensator coefficient to the coefficient generated by the `iqimbal2coef` function. Observe that there is good agreement.

```
idealcoef = iqimbal2coef(ampImb,phImb);
[coef(end); idealcoef]
```

```
ans = 2×1 complex

  -0.1137 + 0.1296i
  -0.1126 + 0.1334i
```

Input Arguments

A — Amplitude imbalance

real-valued scalar or vector

Amplitude imbalance in dB, specified as a real-valued row or column vector.

Example: 3

Example: `[0; 5]`

Data Types: `single` | `double`

P — Phase imbalance
real-valued scalar or vector

Phase imbalance in degrees, specified as a real-valued row or column vector.

Example: `10`

Example: `[15; 45]`

Data Types: `single` | `double`

Output Arguments

C — Compensator coefficient
complex-valued vector

Coefficient that perfectly compensates for the I/Q imbalance, returned as a complex-valued vector having the same dimensions as `A` and `P`.

More About

I/Q Imbalance Compensation

The function `iqimbal2coef` is a supporting function for the `comm.IQImbalanceCompensator` System object.

Define **S** and **X** as 2-by-1 vectors representing the I and Q components of the ideal and I/Q imbalanced signals, respectively.

$$\mathbf{X} = \mathbf{K} \cdot \mathbf{S}$$

where **K** is a 2-by-2 matrix whose values are determined by the amplitude imbalance, *A*, and phase imbalance, *P*. *A* is expressed in dB and *P* is expressed in degrees.

The imbalance can be expressed as:

$$
\begin{aligned}
I_{gain} &= 10^{0.5A/20} \\
Q_{gain} &= 10^{-0.5A/20} \\
\theta_i &= -\left(\frac{P}{2}\right)\left(\frac{\pi}{180}\right) \\
\theta_q &= \frac{\pi}{2} + \left(\frac{P}{2}\right)\left(\frac{\pi}{180}\right)
\end{aligned}
$$

Then **K** has the form:

$$
\mathbf{K} = \begin{bmatrix} I_{gain}\cos(\theta_i) & Q_{gain}\cos(\theta_q) \\ I_{gain}\sin(\theta_i) & Q_{gain}\sin(\theta_q) \end{bmatrix}
$$

The vector **Y** is defined as the I/Q imbalance compensator output.

$$
\mathbf{Y} = \mathbf{R} \cdot \mathbf{X}
$$

For the compensator to perfectly remove the I/Q imbalance, **R** must be the matrix inversion of **K**, namely:

$$
\mathbf{R} = \mathbf{K}^{-1}
$$

Using complex notation, the vector **Y** can be rewritten as:

$$
\begin{aligned}
y &= w_1 x + w_2 \operatorname{conj}(x) \\
&= w_1\left(x + \left({}^{w_2}\!/_{w_1}\right)\operatorname{conj}(x)\right)
\end{aligned}
$$

where,

$$
\begin{aligned}
\operatorname{Re}\{w_1\} &= (R_{11} + R_{22})/2 \\
\operatorname{Im}\{w_1\} &= (R_{21} - R_{12})/2 \\
\operatorname{Re}\{w_2\} &= (R_{11} - R_{22})/2 \\
\operatorname{Im}\{w_2\} &= (R_{21} + R_{12})/2
\end{aligned}
$$

The output of the function is w_2/w_1. To exactly obtain the original signal, the compensator output needs to be scaled and rotated by the complex number w_1.

Note There are cases for which the output of `iqimbal2coef` is unreliable.

- If the phase imbalance is ±90°, the in-phase and quadrature components will become co-linear; consequently, the I/Q imbalance cannot be compensated.
- If the amplitude imbalance is 0 dB and the phase imbalance is 180°, w_1 = 0 and w_2 = 1i; therefore, the compensator takes the form of y = 1i*conj(x).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.IQImbalanceCompensator` | `iqcoef2imbal`

Introduced in R2014b

iqimbal

Apply I/Q imbalance to input signal

Syntax

```
y = iqimbal(x,A)
y = iqimbal(x,A,P)
```

Description

`y = iqimbal(x,A)` applies I/Q amplitude imbalance `A` to input signal `x`.

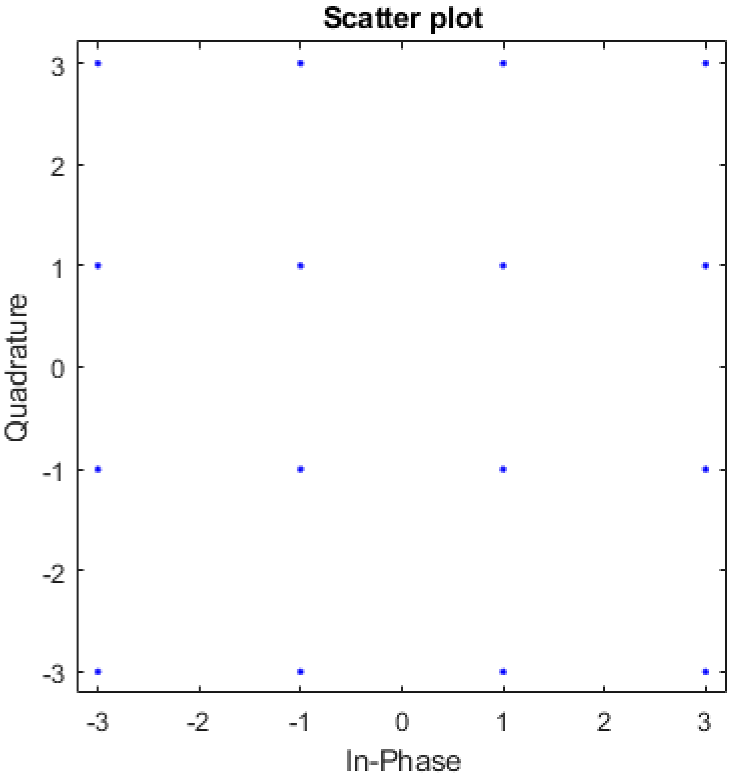
`y = iqimbal(x,A,P)` applies I/Q amplitude imbalance `A` and phase imbalance `P` to input signal `x`.

Examples

Apply Amplitude Imbalance to 16-QAM

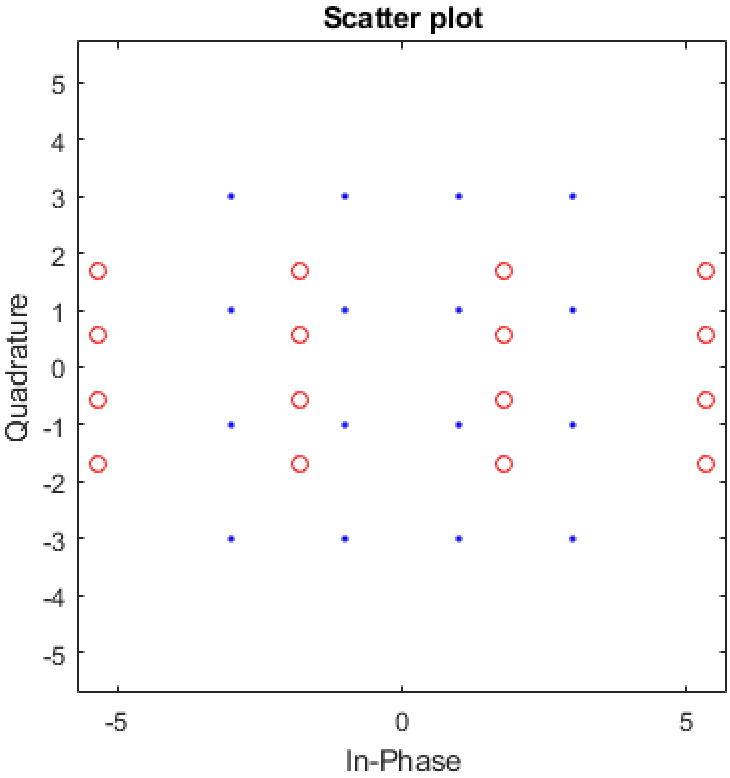
Generate a 16-QAM signal. Display the scatter plot.

```
x = qammod(randi([0 15],1000,1),16);
h = scatterplot(x);
hold on
```



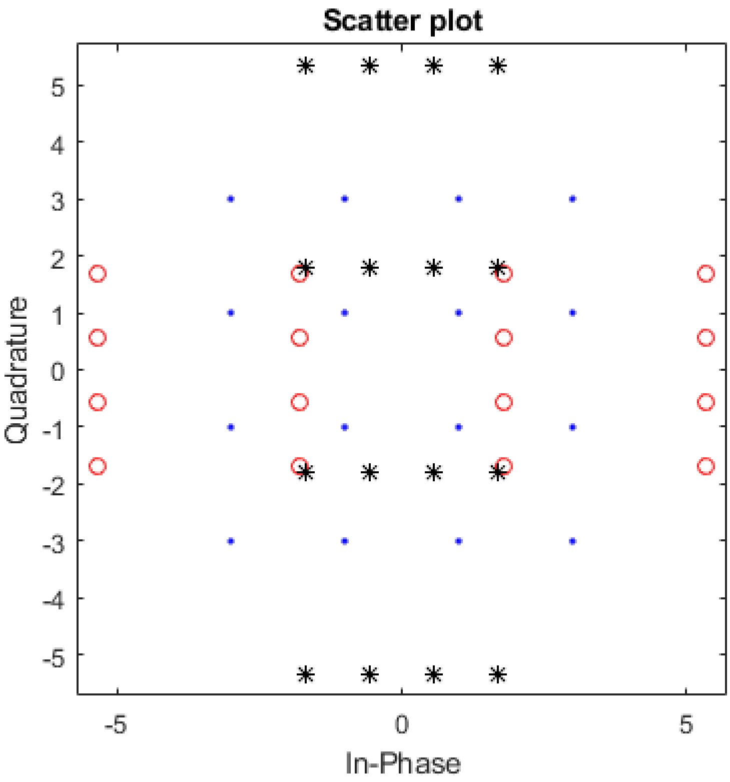
Apply a 10 dB amplitude imbalance. A positive amplitude imbalance causes horizontal stretching of the constellation.

```
y = iqimbal(x,10);
scatterplot(y,1,0,'ro',h)
```



Apply a -10 dB amplitude imbalance. A negative amplitude imbalance causes vertical stretching of the constellation.

```
z = iqimbal(x,-10);
scatterplot(z,1,0,'k*',h)
hold off
```



Apply Phase and Amplitude Imbalance to 16-QAM Signal

Generate a 16-QAM signal having two channels.

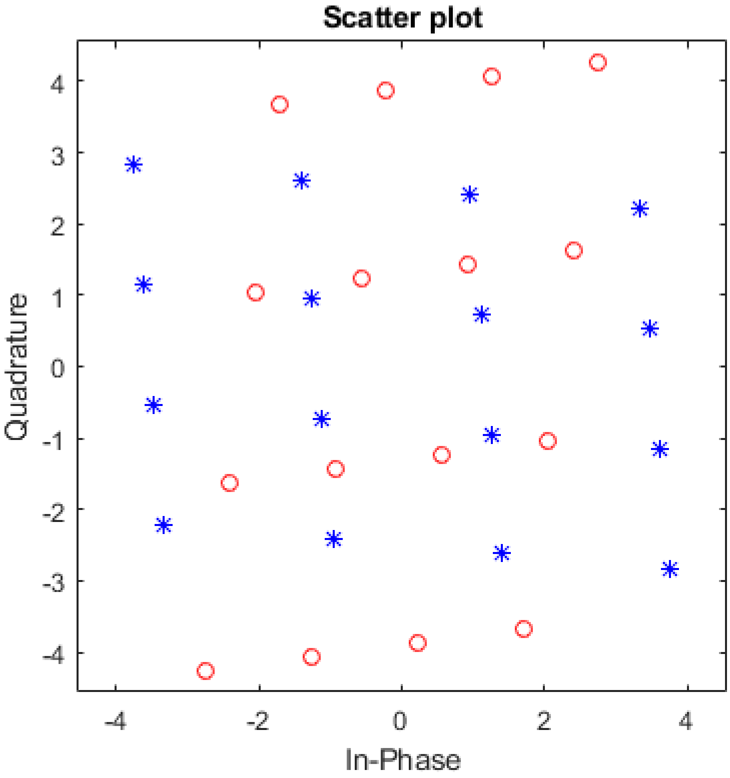
```
x = qammod(randi([0 15],1000,2),16);
```

Apply a 3 dB amplitude imbalance and a 10 degree phase imbalance to the first channel. Apply a –5 dB amplitude imbalance and a –15 degree phase imbalance to the second channel.

```
y = iqimbal(x,[3 -5],[10 -15]);
```

Plot the constellation diagram of both channels of the impaired signal.

```
h = scatterplot(y(:,1),1,0,'b*');
hold on
scatterplot(y(:,2),1,0,'ro',h)
hold off
```



The first channel is stretched horizontally, and the second channel is stretched vertically.

Apply I/Q Imbalance and DC Offset to QPSK

Apply a 1 dB, 5 degree I/Q imbalance to a QPSK signal. Then apply a DC offset. Visualize the offset using a spectrum analyzer.

Generate a QPSK sequence.

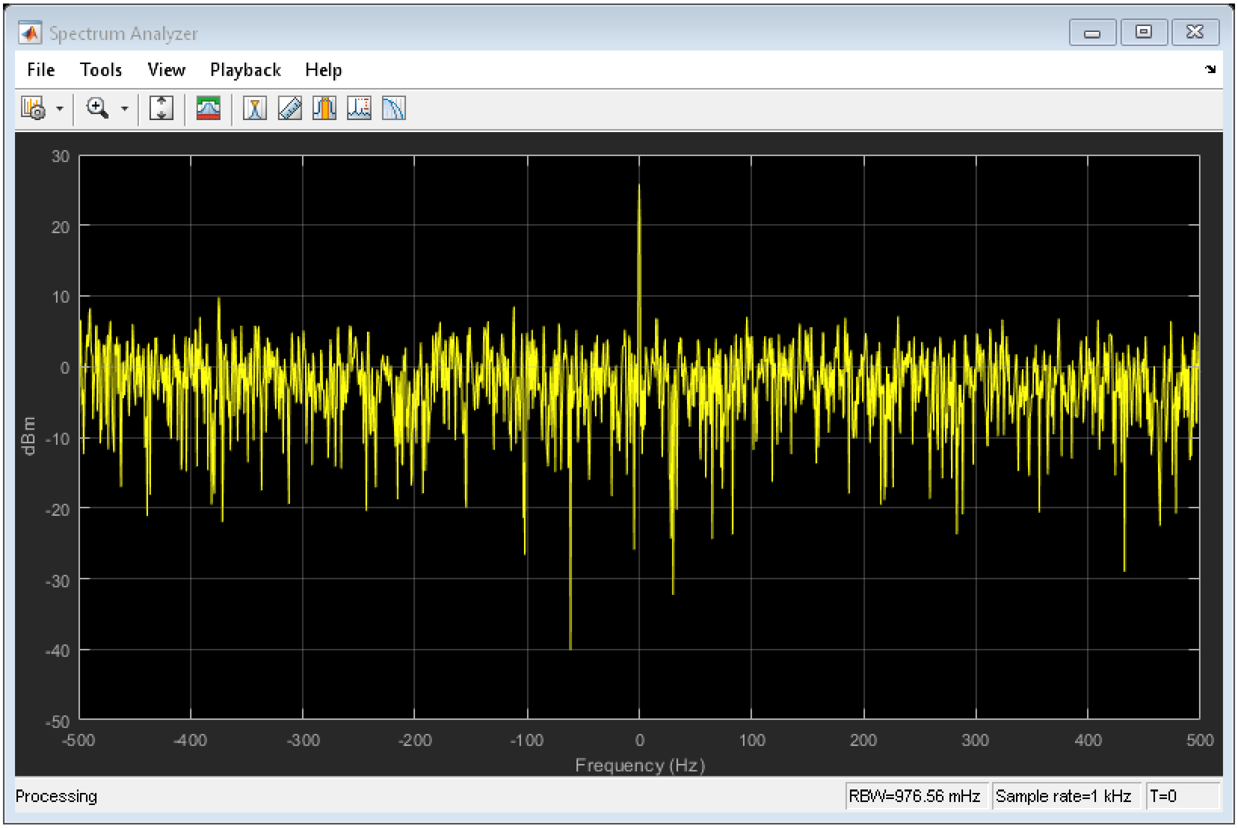
```
x = pskmod(randi([0 3],1e4,1),4,pi/4);
```

Apply a 1 dB amplitude imbalance and 5 degree phase imbalance to a QPSK signal. Apply a 0.5 + 0.3i DC offset.

```
y = iqimbal(x,1,5);
z = y + complex(0.5,0.3);
```

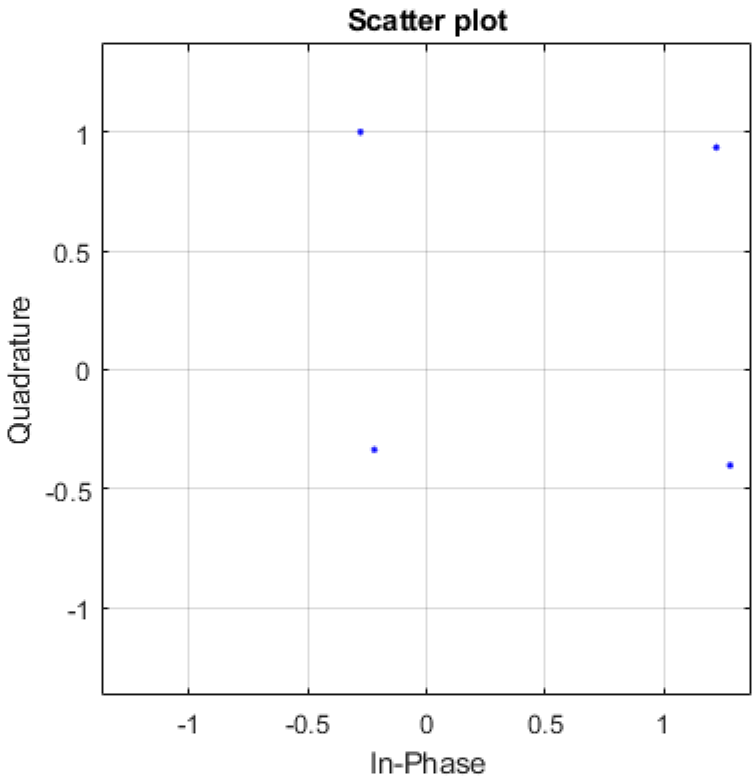
Plot the spectrum of the impaired signal.

```
sa = dsp.SpectrumAnalyzer('SampleRate',1000,'YLimits',[-50 30]);
sa(z)
```



Display the corresponding scatter plot.

```
scatterplot(z)
grid
```



The effect of the I/Q imbalance and the DC offset is observable.

Correct I/Q Imbalance on Noisy 8-PSK Signal

Generate random data and apply 8-PSK modulation.

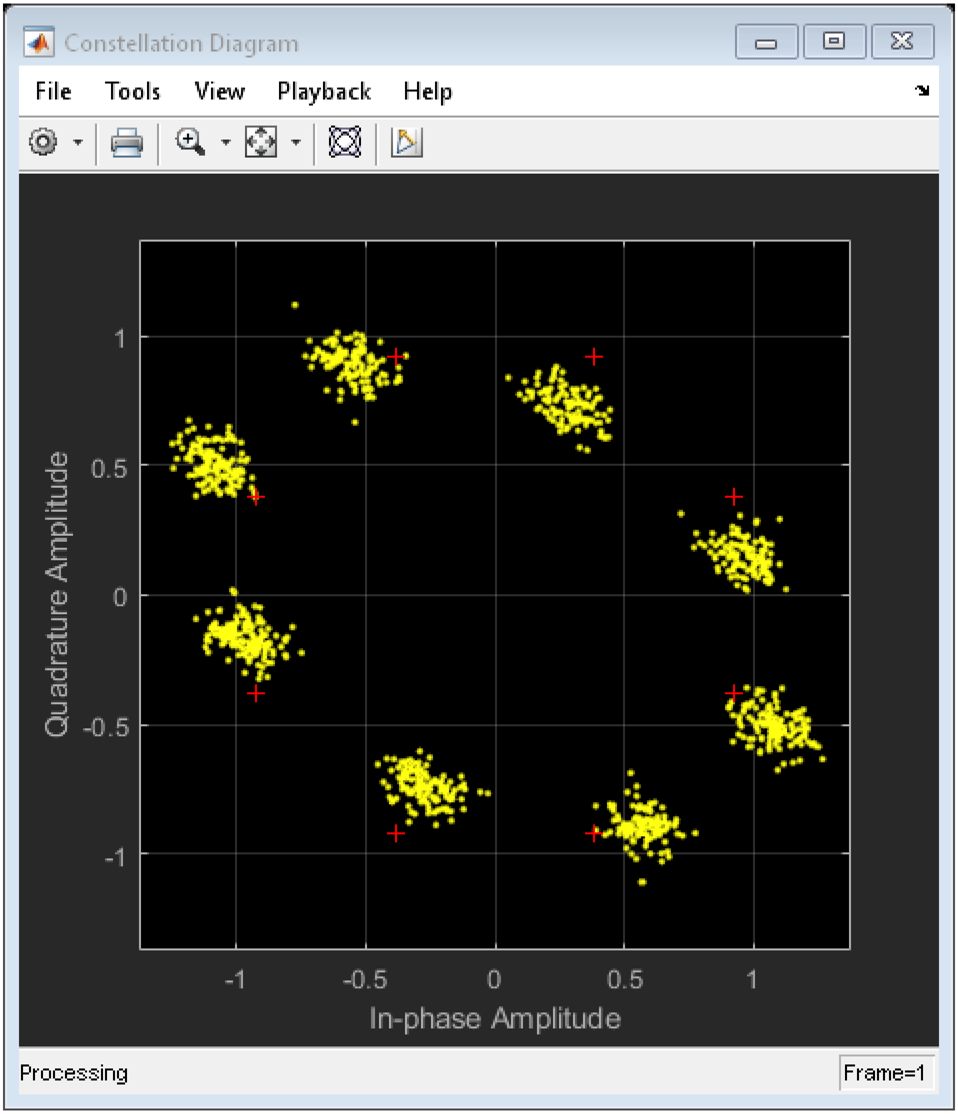
```
data = randi([0 7],2000,1);
txSig = pskmod(data,8,pi/8);
```

Pass the transmitted signal through an AWGN channel. Apply an I/Q imbalance.

```
noisySig = awgn(txSig,20);
rxSig = iqimbal(noisySig,2,20);
```

Create a constellation diagram object that displays only the last 1000 symbols. Plot the constellation diagram of the impaired signal.

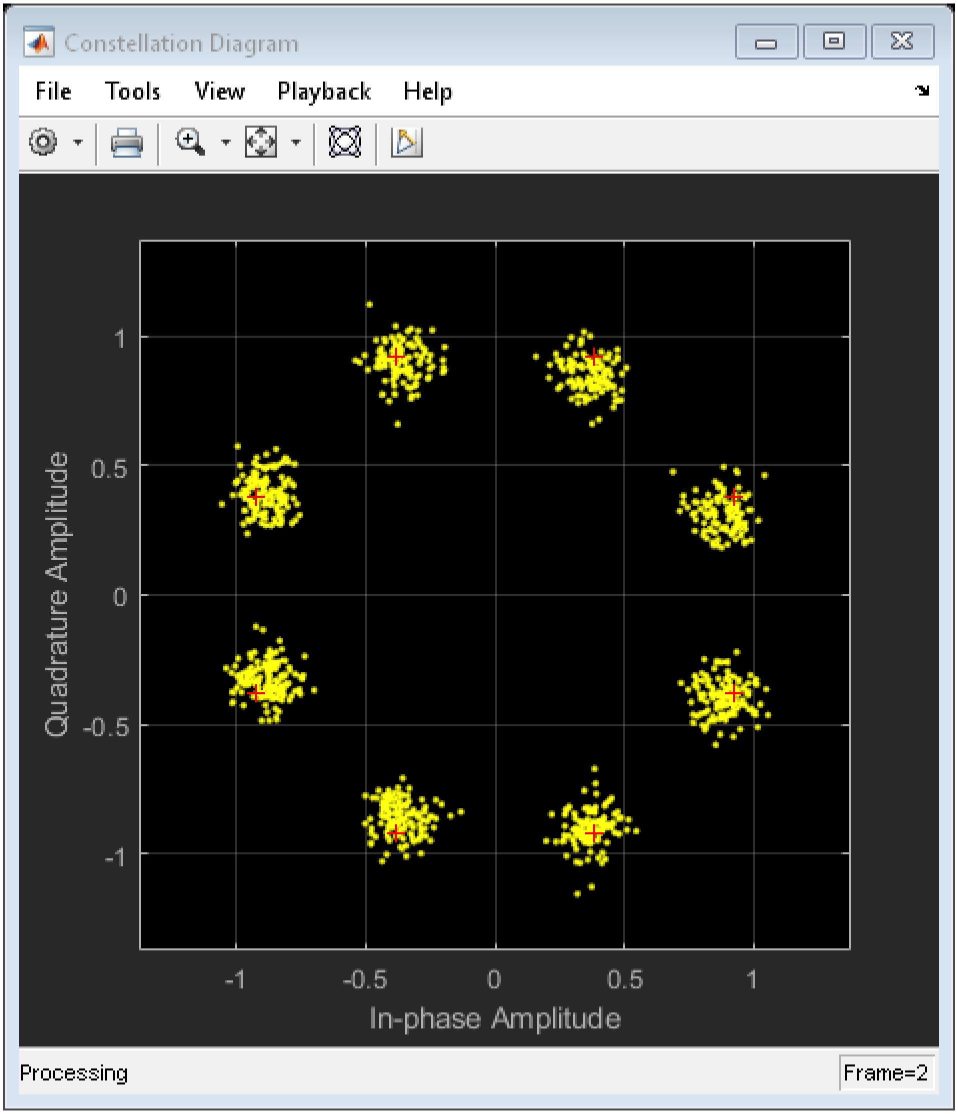
```
cd = comm.ConstellationDiagram('ReferenceConstellation',pskmod(0:7,8,pi/8), ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',1000);
cd(rxSig)
```



Correct for the I/Q imbalance by using a `comm.IQImbalanceCompensator` object. Plot the constellation diagram of the signal after compensation.

```
iqComp = comm.IQImbalanceCompensator('StepSize',1e-3);
compSig = iqComp(rxSig);
```

```
cd(compSig)
```



The compensator removes the I/Q imbalance.

Input Arguments

x — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. The function supports multichannel operations, where the number of columns corresponds to the number of channels.

Example: `pskmod(randi([0 3],100,1),4,pi/4)`

Data Types: `single` | `double`

A — Amplitude imbalance

real scalar | row vector

Amplitude imbalance in dB, specified as a real scalar or row vector.

- If `A` is a scalar, the function applies the same amplitude imbalance to each channel.
- If `A` is a vector, then each element specifies the amplitude imbalance that is applied to the corresponding column (channel) of the input signal. The number of elements in `A` must equal the number of columns in `x`.

Example: `3`

Example: `[0 5]`

Data Types: `single` | `double`

P — Phase imbalance

`0` (default) | real scalar | row vector

Phase imbalance in degrees, specified as a real scalar or row vector.

- If `P` is omitted, a phase imbalance of zero degrees is used.
- If `P` is a scalar, the function applies the same phase imbalance to each channel.
- If `P` is a vector, then each element specifies the phase imbalance that is applied to the corresponding column (channel) of the input signal. The number of elements in `P` must equal the number of columns in `x`.

Example: `10`

Example: `[2.5 7]`

Data Types: `single` | `double`

Output Arguments

y — Output signal
vector | matrix

Output signal, returned as a vector or matrix having the same dimensions as `x`. The number of columns in `y` corresponds to the number of channels.

Data Types: `single` | `double`

Algorithms

The `iqimbal` function applies an I/Q amplitude and phase imbalance to an input signal.

Given amplitude imbalance I_a in dB, the gain, g, resulting from the imbalance is defined as

$$g \triangleq g_r + i g_i = \left[10^{0.5\frac{I_a}{20}}\right] + i\left[10^{-0.5\frac{I_a}{20}}\right].$$

Applying the I/Q imbalance to input signal x results in output signal y such that

$$y = \mathrm{Re}(x) \cdot g_r e^{-i0.5 I_p (\pi/180)} + i\,\mathrm{Im}(x) \cdot g_i e^{i0.5 I_p (\pi/180)} \,,$$

where g is the imbalance gain and I_p is the phase imbalance in degrees.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

I/Q Imbalance | `comm.IQImbalanceCompensator` | `iqcoef2imbal` | `iqimbal2coef`

Introduced in R2016b

iscatastrophic

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`.

Examples

Determine if a Convolutional Code is Catastrophic

Determine if a convolutional code causes catastrophic error propagation.

Create the trellis for the standard, rate 1/2, constraint length 7 convolutional code.

```
t = poly2trellis(7,[171 133]);
```

Verify that the code is not catastrophic.

```
iscatastrophic(t)
```

```
ans = logical
   0
```

Create a trellis for a different convolutional code using the `poly2trellis` function.

```
u = poly2trellis(7,[161 143]);
```

Verify that the code is catastrophic.

```
iscatastrophic(u)

ans = logical
   1
```

References

[1] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, 1995, pp. 274-275.

See Also

`convenc` | `istrellis` | `poly2trellis` | `struct`

Topics

“Convolutional Codes”

Introduced before R2006a

isprimitive

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

`gf` | `primpoly`

Topics

“Galois Field Computations”

Introduced before R2006a

istrellis

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 character vector. Otherwise, `isok` is 0 and `status` 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”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convenc` | `poly2trellis` | `struct` | `vitdec`

Topics

"Convolutional Codes"

Introduced before R2006a

legacychannelsim

(To be removed) Toggles random number generation mode for channel objects

Syntax

```
b = legacychannelsim
legacychannelsim(true)
legacychannelsim(false)
oldmode = legacychannelsim(newmode)
```

Note `legacychannelsim` will be removed in a future release. Use `comm.RayleighChannel` or `comm.RicianChannel` instead.

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`.

Introduced in R2009b

lineareq

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 “Linear Equalizers”.
`WeightInputs`	Vector of tap weight inputs. This is the set of u_i values in the schematic in “Linear 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`.

Property	Description
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

cma | dfe | equalize | lms | normlms | rls | signlms | varlms

Topics

“Equalization”

Introduced before R2006a

lloyds

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

`dpcmopt` | `quantiz`

Topics

“Source Coding”

Introduced before R2006a

lms

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

Property	Description
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

cma | dfe | equalize | lineareq | normlms | rls | signlms | varlms

Topics

“Equalization”

Introduced before R2006a

log

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
```

See Also

`gf`

Introduced before R2006a

lteZadoffChuSeq

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.

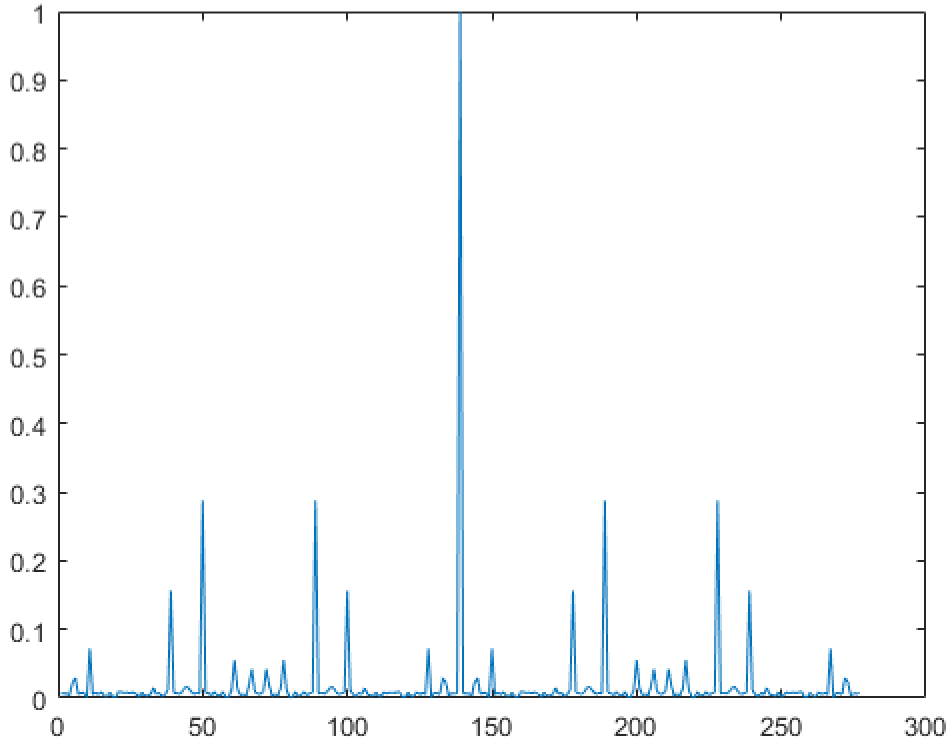
Examples

Examine the Correlation Properties of a Zadoff-Chu Sequence

Generate the 25th root length-139 Zadoff-Chu sequence.

Use `lteZadoffChuSeq` to generate the sequence and then plot its absolute values.

```
seq = lteZadoffChuSeq(25,139);
plot(abs(xcorr(seq)./length(seq)))
```



Input Arguments

R — Root of the Zadoff-Chu sequence

positive integer scalar

Example: 25

Data Types: `single` | `double`
Complex Number Support: Yes

N — Length of the Zadof-Chu sequence

positive integer scalar

Example: 139

Data Types: `single` | `double`
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`.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.GoldSequence` | `comm.PNSequence`

Introduced in R2012b

marcumq

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, $I_{\text{m-1}}$ 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.

Examples

Generate and Plot Marcum Q Function Data

This example shows how to use the `marcumq` function.

Create an input vector, `x`.

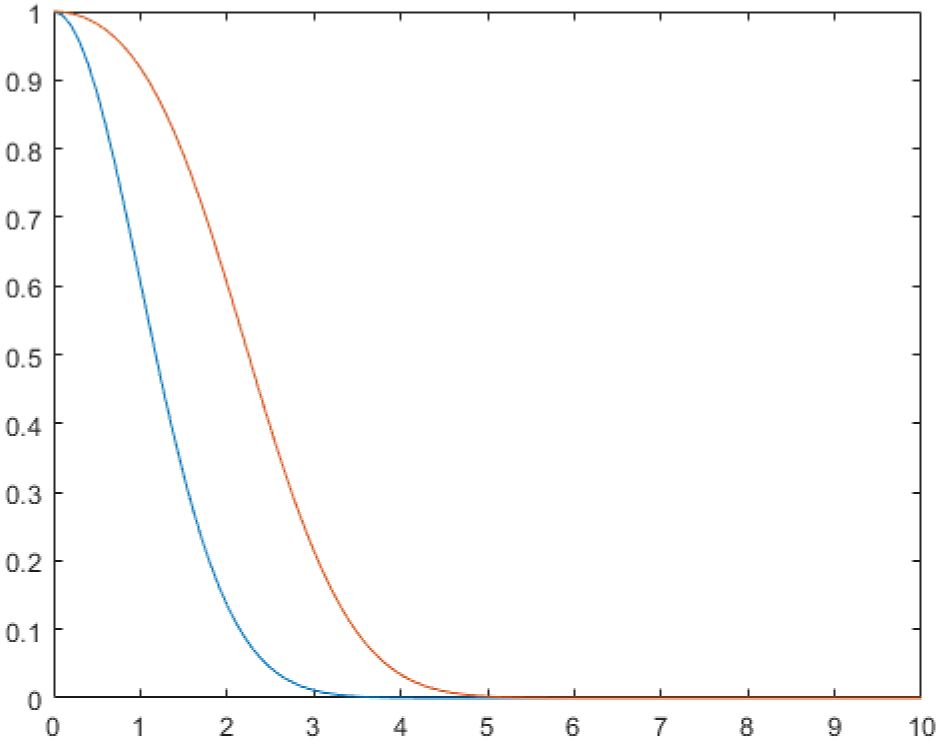
```
x = (0:0.1:10)';
```

Generate two output vectors for a=0 and a=2.

```
Q1 = marcumq(0,x);
Q2 = marcumq(2,x);
```

Plot the resultant Marcum Q functions.

```
plot(x,[Q1 Q2])
```



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`

Introduced before R2006a

mask2shift

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 polynomial character vector
- 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

Convert Mask to Shift

Convert masks into shifts for a linear feedback shift register.

Convert 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.

```
s1 = mask2shift([1 1 0 0 1],[1 0 0 1])
```

```
s1 = 4
```

Convert a mask of `1` to a shift. The mask is equivalent to a shift of `0`.

```
s2 = mask2shift([1 1 0 0 1],[0 0 0 1])
```

```
s2 = 0
```

Convert a mask of x^2 into an equivalent shift for the primitive polynomial x^3+x+1.

```
s3 = mask2shift('x3+x+1','x2')
```

```
s3 = 2
```

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

`isprimitive` | `log` | `primpoly` | `shift2mask`

Introduced before R2006a

matdeintrlv

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`

Topics

"Interleaving"

Introduced before R2006a

matintrlv

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`

Topics

“Interleaving”

Introduced before R2006a

mil188qamdemod

MIL-STD-188-110 B/C standard-specific quadrature amplitude demodulation

Syntax

```
z = mil188qamdemod(y,M)
z = mil188qamdemod(y,M,Name,Value)
```

Description

`z = mil188qamdemod(y,M)` performs QAM demodulation on an input signal, `y`, that was modulated in accordance with MIL-STD-188-110 and the modulation order, `M`. For a description of MIL-STD-188-110 QAM demodulation, see “Algorithms” on page 1-716.

`z = mil188qamdemod(y,M,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate MIL-STD-188-110B Specific 16-QAM Signal

Demodulate a 16-QAM signal that was modulated as specified in MIL-STD-188-110B. Plot the received constellation and verify that the output matches the input.

Set the modulation order and generate random data.

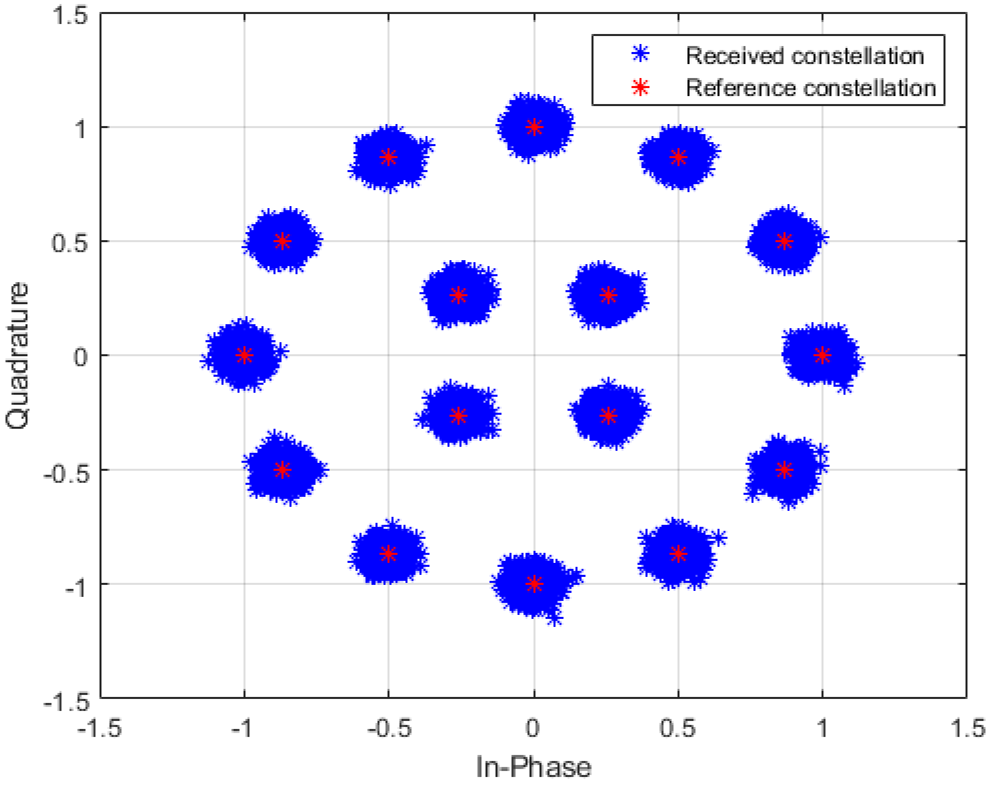
```
M = 16;
numSym = 20000;
x = randi([0 M-1],numSym,1);
```

Modulate the data and pass through a noisy channel.

```
txSig = mil188qammod(x,M);
rxSig = awgn(txSig,25,'measured');
```

Plot the transmitted and received signal.

```
plot(rxSig,'b*')
hold on; grid
plot(txSig,'r*')
xlim([-1.5 1.5]);
ylim([-1.5 1.5])
xlabel('In-Phase')
ylabel('Quadrature')
legend('Received constellation','Reference constellation')
```



Demodulate the received signal. Compare the demodulated data to the original data.

```
z = mil188qamdemod(rxSig,M);
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate MIL-STD-188-110C Specific 64-QAM Signal

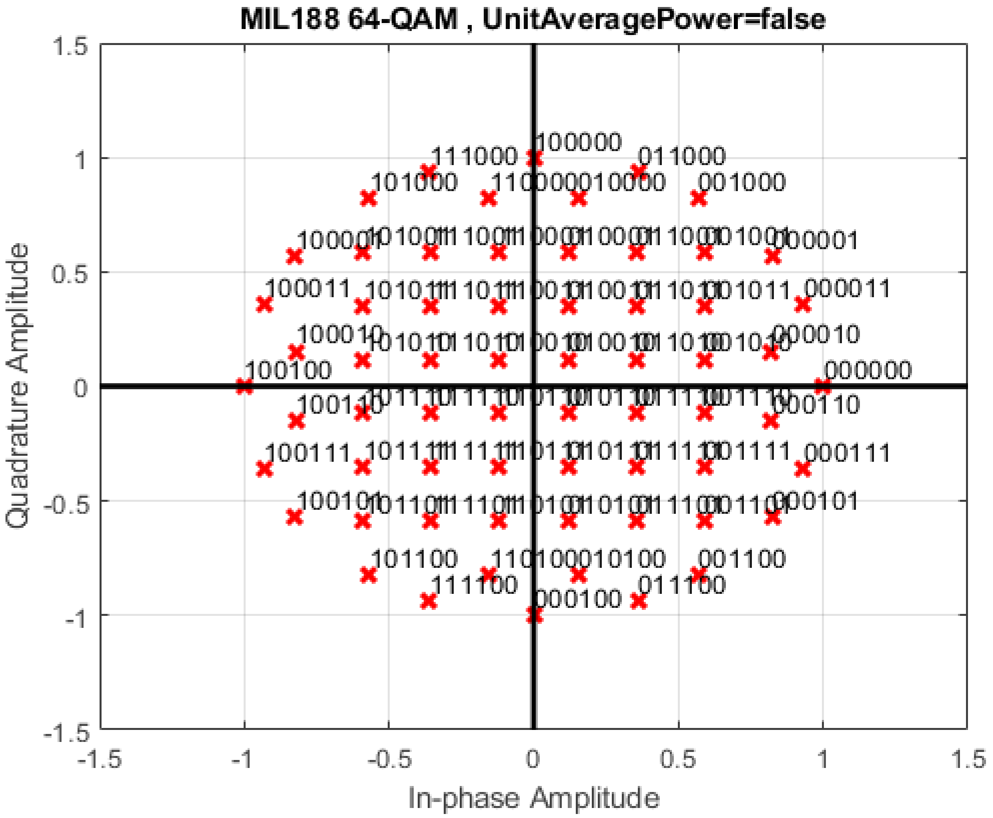
Demodulate a 64-QAM signal that was modulated as specified in MIL-STD-188-110C. Compute hard decision bit output and that verify the output matches the input.

Set the modulation order and generate random bit data.

```
M = 64;
numBitsPerSym = log2(M);
x = randi([0 1],1000*numBitsPerSym,1);
```

Modulate the data. Use name-value pairs to specify bit input data and to plot the constellation.

```
txSig = mil188qammod(x,M,'InputType','bit','PlotConstellation',true);
```



Demodulate the received signal. Compare the demodulated data to the original data.

```
z = mil188qamdemod(txSig,M,'OutputType','bit');
isequal(z,x)
```

```
ans = logical
   1
```

Soft Bit Demodulate MIL-STD-188-110 Specific 32-QAM Signal

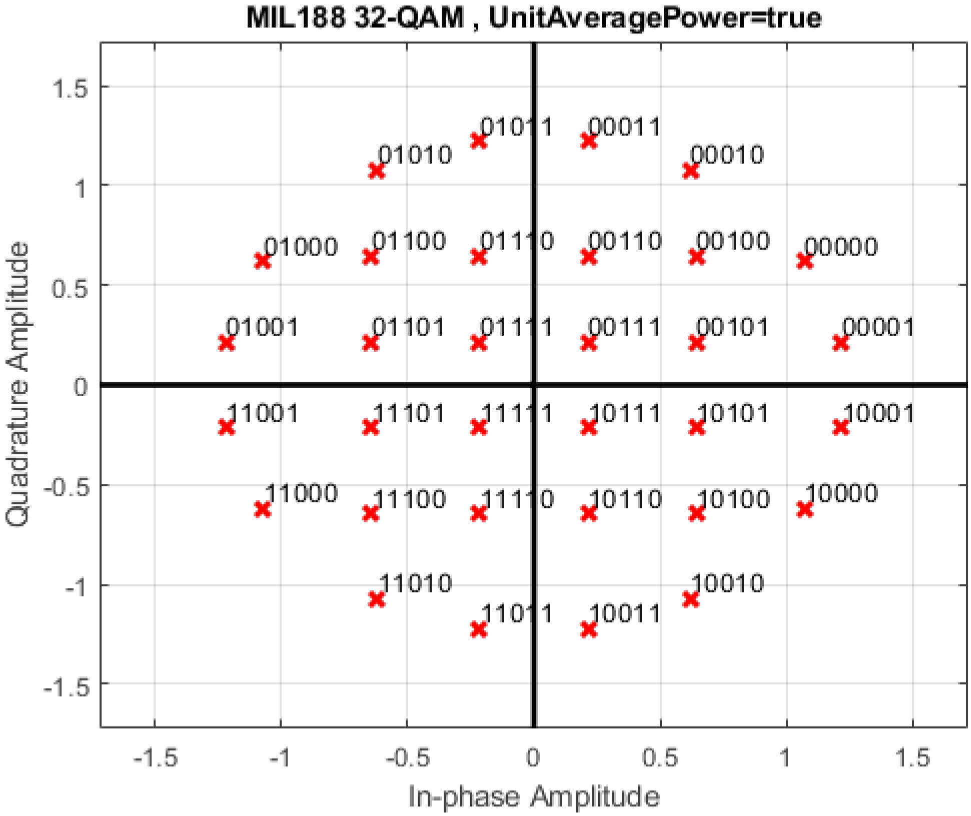
Demodulate a 32-QAM signal and calculate soft bits.

Set the modulation order and generate a random bit sequence.

```
M = 32;
numSym = 20000;
numBitsPerSym = log2(M);
x = randi([0 1], numSym*numBitsPerSym,1);
```

Modulate the data. Use name-value pairs to specify bit input data and unit average power, and to plot the constellation.

```
txSig = mil188qammod(x,M,'InputType','bit','UnitAveragePower',true, ...
    'PlotConstellation',true);
```

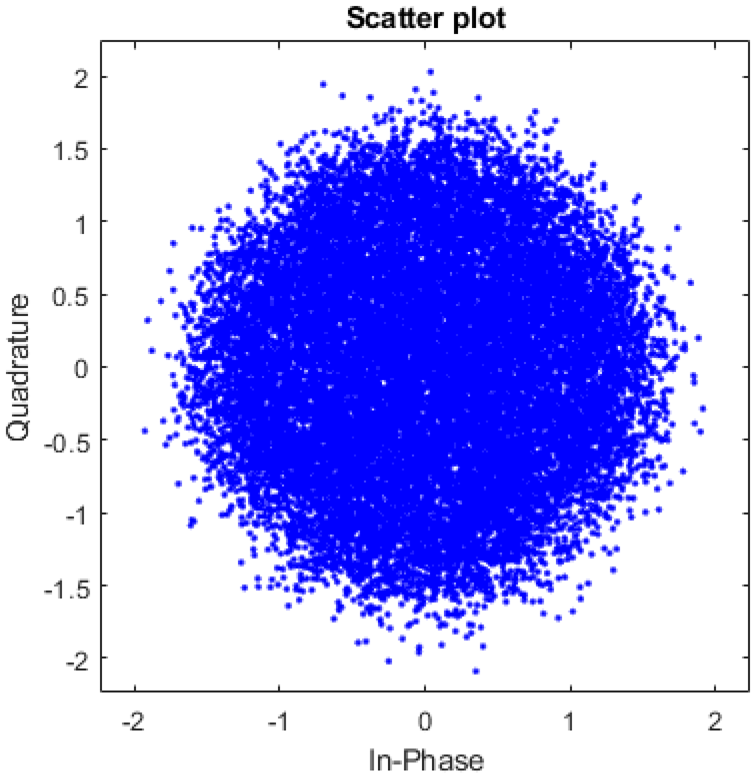


Pass the transmitted data through white Gaussian noise.

```
rxSig = awgn(txSig,10,'measured');
```

View the constellation using a scatter plot.

```
scatterplot(rxSig)
```



Demodulate the signal, computing soft bits using the approximate LLR algorithm.

```
z = mil188qamdemod(rxSig,M,'OutputType','approxllr', ...
     'NoiseVariance',10^(-1));
```

Input Arguments

y — Modulated signal
scalar | vector | matrix

Modulated signal, specified as a complex scalar, vector, or matrix. When `y` is a matrix, each column is treated as an independent channel.

`y` must be modulated in accordance with MIL-STD-188-110 [1].

Data Types: `single` | `double`
Complex Number Support: Yes

`M` — Modulation order
integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Example: `16`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = mil188qamdemod(x,M,'OutputType','bit','OutputDataType','single');`

`OutputType` — Output type
`'integer'` (default) | `'bit'` | `'llr'` | `'approxllr'`

Output type, specified as the comma-separated pair consisting of `'OutputType'` and `'integer'`, `'bit'`, `'llr'`, or `'approxllr'`. For a description of returned output, see `z`.

Data Types: `char` | `string`

`OutputDataType` — Output data type
`'double'` (default) | `...`

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and one of the indicated data types. Acceptable values for `'OutputDataType'` depend on the `'OutputType'` value.

'OutputType' Value	**Acceptable 'OutputDataType' Values**
'integer'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', or 'uint32'
'bit'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', 'uint32', or 'logical'

Dependencies

This name-value pair argument applies only when `'OutputType'` is set to `'integer'` or `'bit'`.

Data Types: `char` | `string`

UnitAveragePower — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the functon scales the constellation based on specifications in the relevant standard, as described in [1].

Data Types: `logical`

NoiseVariance — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of `'NoiseVariance'` and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

Tip When the output type is set to 'llr', an exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- Inf or -Inf if the noise variance is a very large value
- NaN if both the noise variance and signal power are a very small values

When the output returns any of these values, try setting output type to 'approxllr' instead. The approximate LLR algorithm does not compute exponentials.

Dependencies

This name-value pair argument applies only when OutputType is set to 'llr' or 'approxllr'.

Data Types: double

PlotConstellation — Option to plot constellation

false (default) | true

Option to plot constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

z — Demodulated signal

scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of z depend on the specified 'OutputType' value.

'OutputType' Value	Return Value of mil188qamdemod	Dimensions of z
'integer'	Demodulated integer values from 0 to (M – 1)	z has the same dimensions as input y.

'OutputType' Value	Return Value of mil188qamdemod	Dimensions of z
'bit'	Demodulated bits	The number of rows in z is $\log_2$(sum(M)) times the number of rows in y. Each demodulated symbol is mapped to a group of $\log_2$(sum(M)) elements in a column, where the first element represents the MSB and the last element represents the LSB.
'llr'	Log-likelihood ratio value for each bit	
'approxllr'	Approximate log-likelihood ratio value for each bit	

Algorithms

MIL-STD-188-110 QAM Hard Demodulation

The hard demodulation algorithm uses optimum decision region-based demodulation. Since all the constellation points are equally probable, maximum a posteriori probability (MAP) detection reduces to a maximum likelihood (ML) detection. The ML detection rule is equivalent to choosing the closest constellation point to the received symbol. The decision region for each constellation point is designed by drawing perpendicular bisectors between adjacent points. A received symbol is mapped to the proper constellation point based on which decision region it lies in.

Since all MIL-STD constellations are quadrant-based symmetric, for each symbol the optimum decision region-based demodulation:

- Maps the received symbol into the first quadrant
- Choses the decision region for the symbol
- Maps the constellation point back to its original quadrant using the sign of real and imaginary parts of the received symbol

MIL-STD-188-110 QAM Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio algorithms are available: exact LLR and approximate LLR. Exact LLR provides the greatest accuracy but is slower, while approximate LLR is less accurate but faster.

For a description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems." Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskdemod` | `genqamdemod` | `mil188qammod` | `pskdemod` | `qamdemod`

System Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator` | `comm.RectangularQAMDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

mil188qammod

MIL-STD-188-110 B/C standard-specific quadrature amplitude modulation (QAM)

Syntax

```
y = mil188qammod(x,M)
y = mil188qammod(x,M,Name,Value)
```

Description

`y = mil188qammod(x,M)` performs QAM modulation on the input signal, `x`, in accordance with MIL-STD-188-110 and the modulation order, `M`. For more information, see “MIL-STD-188-110” on page 1-725.

`y = mil188qammod(x,M,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply 32-QAM to Data Per MIL-STD-188-110C

Modulate data using 32-QAM as specified in the MIL-188-110C standard. Display the result using a scatter plot.

Set M to 32 and create a data vector containing all possible symbols.

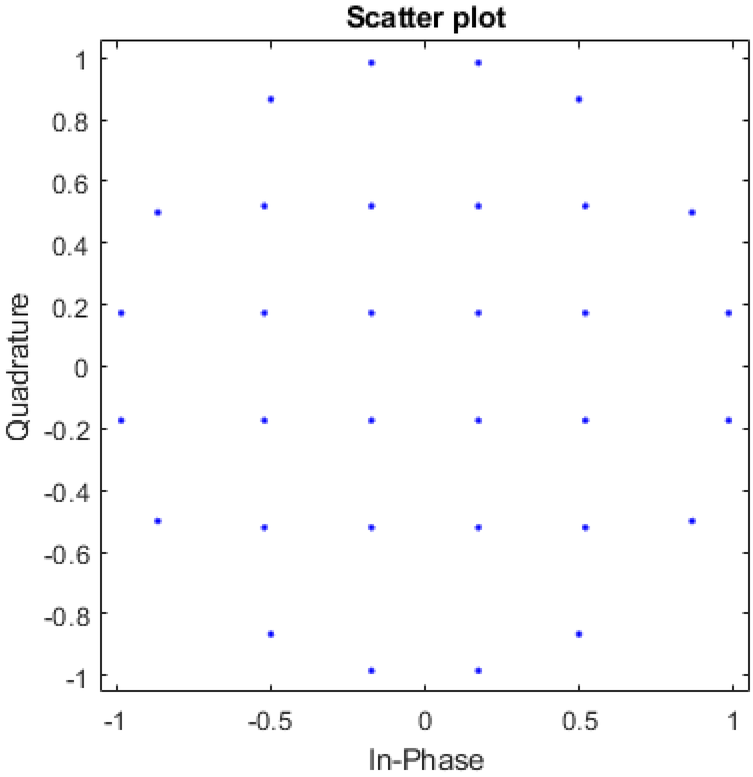
```
M = 32;
x = (0:M-1);
```

Modulate the data using QAM as specified in MIL-STD-188-110C.

```
y = mil188qammod(x,M);
```

Display the constellation as a scatter plot.

```
scatterplot(y)
```



Normalize 16-QAM Modulated MIL-STD-188-110B Signal by Average Power

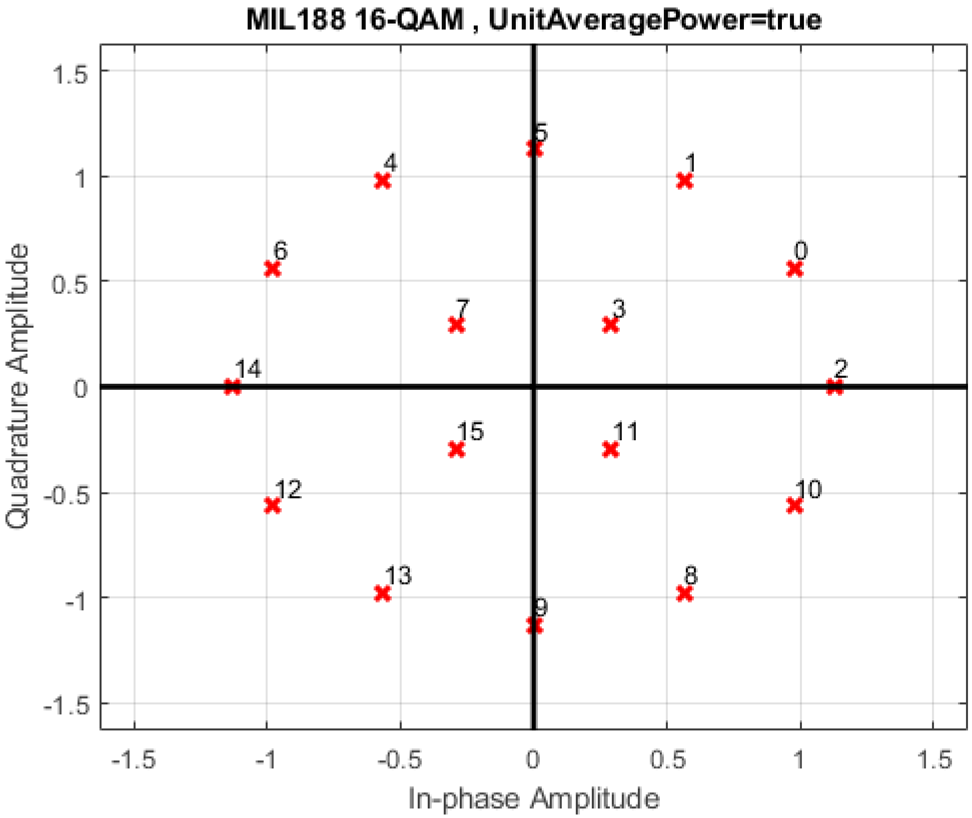
Modulate data using 16-QAM as specified in the MIL-STD-188-110B standard. Normalize the modulator output so that it has an average signal power of 1 W.

Set M and generate random data.

```
M = 16;
x = randi([0 M-1],1e5,1);
```

Modulate the data applying 16-QAM as specified in MIL-STD-188-100B. Using name-value pairs, set the unit average power to `true` and enable the constellation plot.

```
y = mil188qammod(x,M,'UnitAveragePower',true,'PlotConstellation',true);
```



Verify that the signal has unit average power.

```
avgPow = mean(abs(y).^2)
```

```
avgPow = 1.0012
```

Apply 64-QAM MIL-STD-188-110B Modulation to Bit Data

Modulate a sequence of bits using 64-QAM as specified by MIL-STD188-110B. Display the constellation.

Set the modulation order and generate a sequence of random bits.

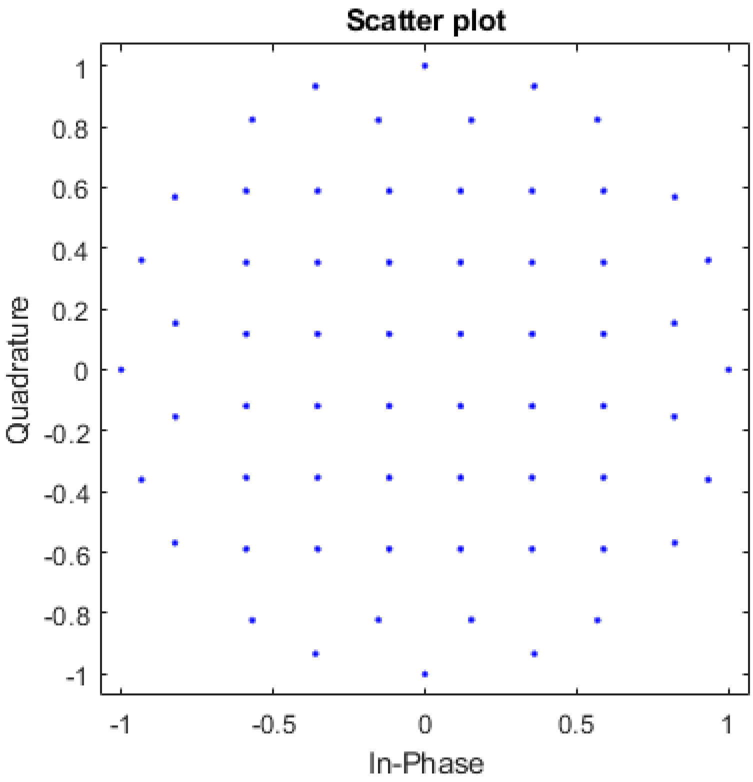
```
M = 64;
numBitsPerSym = log2(M);
data = randi([0 1],1000*numBitsPerSym,1);
```

Modulate the data applying 64-QAM as specified by MIL-STD-188-110B, and output constellation symbols of single data type.

```
y = mil188qammod(data,M,'InputType','bit','OutputDataType','single');
```

Plot the result constellation using a scatter plot.

```
scatterplot(y)
```



Input Arguments

x — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers that range from 0 to (M – 1), where M is the modulation order.

Note To process input signal as binary elements, set the `'InputType'` value to `'bit'`. For binary inputs, the number of rows must be an integer multiple of $\log_2$(M). Groups of

$\log_2$(M) bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

M — Modulation order
integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Example: 16

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: y = mil188qammod(data,M,'InputType','bit','OutputDataType','single');

InputType — Input type
'integer' (default) | 'bit'

Input type, specified as the comma-separated pair consisting of 'InputType' and either 'integer' or 'bit'. If you specify 'integer', the input signal must consist of integers from 0 to M – 1. If you specify 'bit', the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2$(M).

Data Types: char | string

OutputDataType — Output data type
'double' (default) | 'single'

Output data type, specified as the comma-separated pair consisting of OutputDataType and 'double' or 'single'.

Data Types: char | string

UnitAveragePower — Unit average power flag

false (default) | true

Unit average power flag, specified as the comma-separated pair consisting of 'UnitAveragePower' and a logical scalar. When this flag is true, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is false, the functon scales the constellation based on specifications in the relevant standard, as described in [1].

Data Types: logical

PlotConstellation — Option to plot constellation

false (default) | true

Option to plot constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

y — Modulated signal

scalar | vector | matrix

Modulated signal, returned as a complex scalar, vector, or matrix. The dimension of the output depends on the specified InputType value.

InputType	Dimensions of Output
'integer'	y has the same dimensions as input x.
'bit'	The number of rows in y equals the number of rows in x divided by $\log_2(M)$ times.

Data Types: double | single

Definitions

MIL-STD-188-110

MIL-STD-188-110 is a US Department of Defense standard for HF communications using serial PSK mode of both data and voice signals.

The standard specifies physical layer modulation schemes for tactical and long-haul communications. The modulation scheme specified by the standard is a mix of QAM and APSK. For a detailed description of the modulation scheme, see [1].

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems". Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`apskmod` | `dvbsapskmod` | `genqammod` | `mil188qamdemod` | `pskmod` | `qammod`

System Objects
`comm.GeneralQAMModulator` | `comm.PSKModulator` | `comm.RectangularQAMModulator`

Introduced in R2018a

minpol

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` | `gf`

Topics

“Polynomials over Galois Fields”

Introduced before R2006a

mldivide

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 matrix, in which there are more rows than columns, such that A'*A is nonsingular
- A matrix, in which there are more columns than rows, such that A*A' is nonsingular

Algorithms

If A is an M-by-N matrix where M > N, A \ B is the same as (A'*A) \ (A'*B).

If A is an M-by-N matrix where M < N, A \ B is the same as A' * ((A*A') \ B). This solution is not unique.

See Also

gf

Topics

"Linear Algebra in Galois Fields"

Introduced before R2006a

mlseeq

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', nsamp);
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 “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`

Topics

“MLSE Equalizers”

Introduced before R2006a

modnorm

Scaling factor for normalizing modulation output

Syntax

```
normfactor = modnorm(refconst,type,power)
```

Description

`normfactor = modnorm(refconst,type,power)` returns a scale factor for normalizing a PAM or QAM modulator output using the specified reference constellation, normalization type, and output power.

Examples

Normalize Power of QAM Signal

Generate a 16-QAM reference constellation.

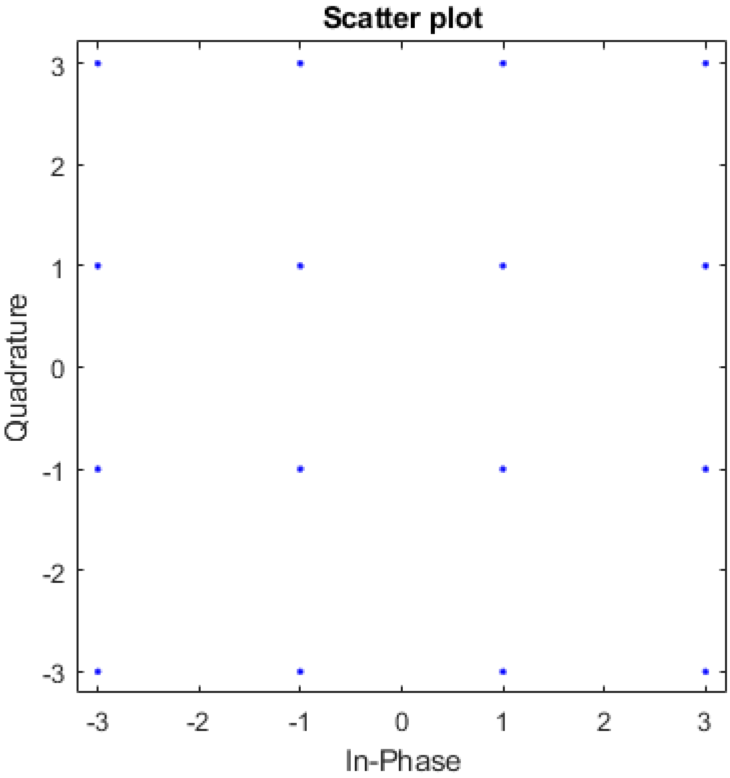
```
refconst = qammod(0:15,16);
```

Generate random symbols and apply 16-QAM modulation.

```
x = randi([0 15],1000,1);
y = qammod(x,16);
```

Plot the constellation.

```
h = scatterplot(y);
```



Compute the normalization factor so that the output signal has a peak power of 1 W.

```
nf = modnorm(refconst,'peakpow',1);
z = nf*y;
```

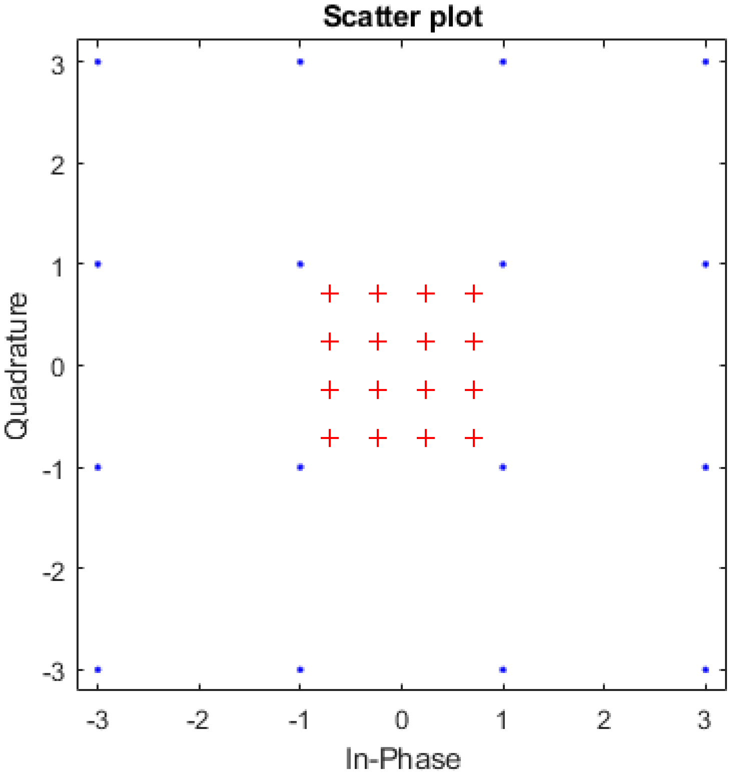
Confirm that no element of the normalized signal has a power greater than 1 W.

```
max(z.*conj(z))
```

```
ans = 1.0000
```

Plot the scatter plot of the normalized constellation.

```
hold on
scatterplot(z,1,0,'r+',h)
hold off
```



Input Arguments

`refconst` — Reference constellation

vector

Reference constellation, specified as a vector of complex elements that comprise the reference constellation points.

Example: `qammod(0:15,16)`

Data Types: `double` | `single`
Complex Number Support: Yes

`type` — Normalization type
`'avpow'` | `'peakpow'`

Normalization type, specified as either `'avpow'` or `'peakpow'`.

- If `type` is `'avpow'`, the normalization factor is calculated based on average power.
- If `type` is `'peakpow'`, the normalization factor is calculated based on peak power.

Data Types: `char`

`power` — Target power
scalar

Target power, specified as a real scalar. The target power is the intended power of the modulated signal multiplied by `normfactor`.

Data Types: `double` | `single`

Output Arguments

`normfactor` — Normalization factor
scalar

Normalization factor, returned as a real scalar. When a modulated signal is multiplied by the normalization factor, its average or peak power matches the target power. The function assumes that the signal you want to normalize has a minimum distance of 2.

Data Types: `double` | `single`

See Also

pamdemod | pammod | qamdemod | qammod

Introduced before R2006a

mskdemod

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

`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

MSK Demodulation

Modulate and demodulate a noisy MSK signal. Display the number of received errors.

Define the number of samples per symbol for the MSK signal.

```
nsamp = 16;
```

Initialize the simulation parameters.

```
numerrs = 0;
modPhase = zeros(1,2);
demodPhase = zeros(1,2);
demodState = complex(zeros(nsamp,2));
```

The main processing loop includes these steps:

- Generate binary data.
- MSK modulate the data.
- Pass the signal through an AWGN channel.
- Demodulate the MSK signal.
- Determine the number of bit errors.

```
for iRuns = 1:20
    txData = randi([0 1],100,2);
    [modSig,modPhase] = mskmod(txData,nsamp,[],modPhase);
    rxSig = awgn(modSig,20,'measured');
    [rxData,demodPhase,demodState] = mskdemod(rxSig,nsamp,[],demodPhase,demodState);
    numerrs = numerrs + biterr(txData,rxData);
end
```

Display the number of bit errors.

```
numerrs
```

```
numerrs = 0
```

References

[1] Pasupathy, Subbarayan, “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`comm.MSKDemodulator` | `fskdemod` | `fskmod` | `mskmod`

Topics

“Digital Modulation”

Introduced before R2006a

mskmod

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

`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

Eye Diagram of MSK Signal

Generate a random binary signal.

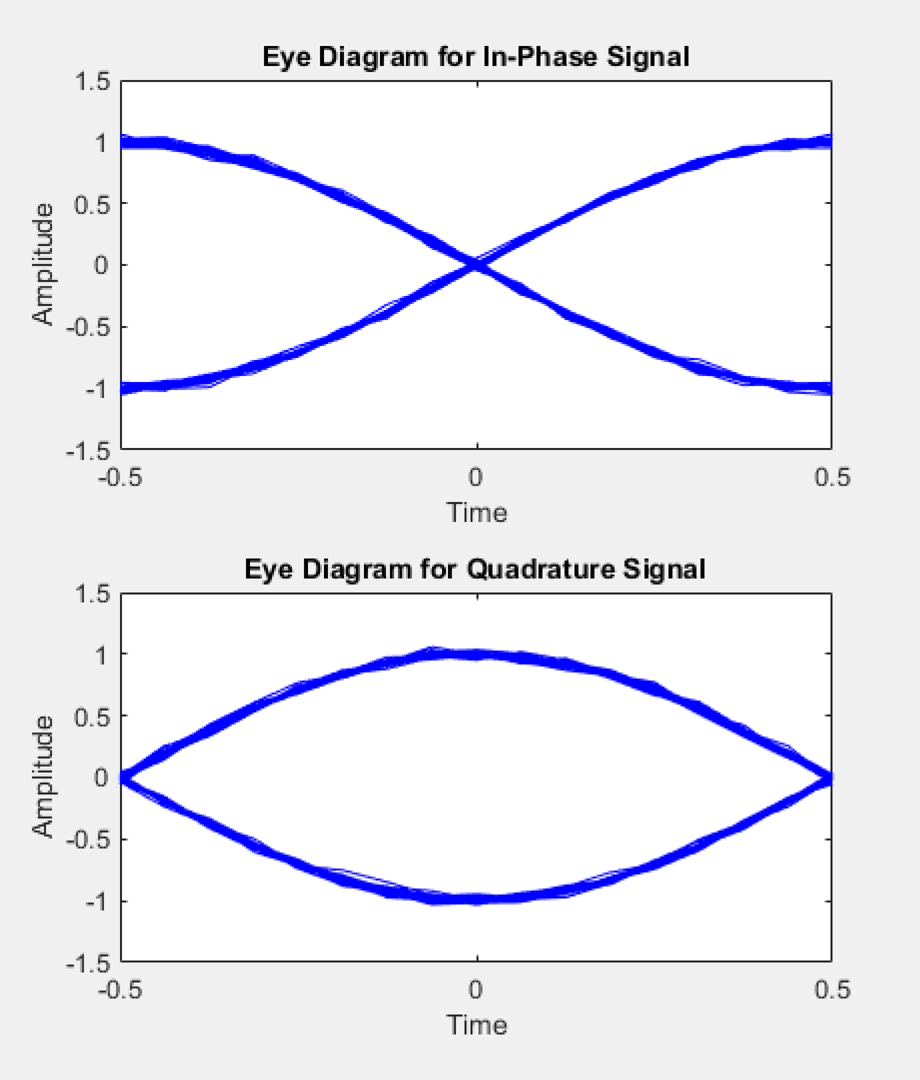
```
x = randi([0 1],100,1);
```

MSK modulate the data.

```
y = mskmod(x,8,[],pi/2);
```

Pass the signal through an AWGN channel. Display the eye diagram.

```
z = awgn(y,30,'measured');
eyediagram(z,16);
```


Eye Diagram for In-Phase Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time
Eye Diagram for Quadrature Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time

References

[1] Pasupathy, Subbarayan, “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`comm.MSKModulator` | `fskdemod` | `fskmod` | `mskdemod`

Introduced before R2006a

muxdeintrlv

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`

Topics

“Interleaving”

Introduced before R2006a

muxintrlv

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

`convintrlv` | `helintrlv` | `muxdeintrlv`

Topics

"Interleaving"

Introduced before R2006a

noisebw

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

Noise Equivalent Bandwidth of Butterworth Filter

Computes the equivalent noise bandwidth of a Butterworth filter over 100 samples of the impulse response.

Set the sampling rate, Nyquist frequency, and carrier frequency.

```
fs = 16;
fNyq = fs/2;
fc = 0.5;
```

Generate the Butterworth filter.

```
[num,den] = butter(2,fc/fNyq);
```

Determine the noise bandwidth.

```
bw = noisebw(num,den,100,fs)
```

```
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.

Introduced before R2006a

normlms

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

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

Topics

“Equalization”

Introduced before R2006a

oct2dec

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

Convert Octal Matrix to Decimal Equivalent

Convert a 2-by-2 octal matrix its decimal equivalent.

```
d = oct2dec([12 144;0 25])
```

```
d = 2×2

    10   100
     0    21
```

The octal number 144 is equivalent to 100 because $144 = 1(8^2) + 4(8^1) + 4(8^0) = 100$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bi2de`

Introduced before R2006a

ofdmdemod

Demodulate time-domain signal using orthogonal frequency division multiplexing (OFDM)

Syntax

```
outSym = ofdmdemod(ofdmSig,nfft,cplen)
outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset)
outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx)
[outSym,pilots] = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx,
pilotidx)
```

Description

`outSym = ofdmdemod(ofdmSig,nfft,cplen)` performs OFDM demodulation on the input time domain signal specified in `ofdmSig`, using an FFT size specified by `nfft` and cyclic prefix length specified by `cplen`. For information, see “OFDM Demodulation” on page 1-764.

`outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset)` applies the symbol sampling offset, `symOffset`, for each OFDM symbol before demodulation of the input.

`outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx)` removes null subcarriers from the locations specified in `nullidx`. For this syntax, the symbol sampling offset is applied to each OFDM symbol and the number of rows in the output is `nfft – length(nullidx)`, which accounts for the removal of null subcarriers. Use null subcarriers to account for guard bands and DC subcarriers. For information, see “Subcarrier Allocation and Guard Bands” on page 1-765.

`[outSym,pilots] = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx, pilotidx)` returns pilot subcarriers for the pilot indices specified in `pilotidx`. For this syntax, the symbol sampling offset is applied to each OFDM symbol and number of rows in the output is `nfft – length(nullidx) – length(pilotidx)`, which accounts for the removal of null and pilot subcarriers. The function assumes that pilot subcarrier locations are the same across each OFDM symbol and transmit antenna.

Examples

OFDM Demodulation with Different CP Lengths

OFDM-demodulate a signal with different CP lengths for different symbols.

Initialize input parameters defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft = 64;
cplen = [16 32];
nSym = 2;
dataIn = complex(randn(nfft,nSym),randn(nfft,nSym));
y1 = ofdmmod(dataIn,nfft,cplen);
```

Demodulate the OFDM symbols. Compare the results to the original input data. The difference between the signals is negligible.

```
x1 = ofdmdemod(y1,nfft,cplen);
max(x1-dataIn)
```

```
ans = 1×2 complex
10^-15 ×

   0.2220 - 0.7772i   0.2498 - 0.8882i
```

OFDM Mod-Demod SISO link

Apply OFDM multiplexing to a 16-QAM signal SISO link with Rayleigh fading.

```
s1 = RandStream('mt19937ar','Seed',12345);
nFFT = 64;
cpLen = 16;
nullIdx = [1:6 33 64-4:64].';
numTones = nFFT-length(nullIdx);

modOrd = 4;
qam = comm.RectangularQAMModulator('ModulationOrder',2^modOrd, ...
    'BitInput',true,'NormalizationMethod','Average power');

maxDopp = 1;
pathDelays = [0 4e-3 8e-3];
pathGains = [0 -2 -3];
sRate = 1000;
sampIdx = round(pathDelays/(1/sRate)) + 1;

chan = comm.RayleighChannel('PathGainsOutputPort',true, ...
    'MaximumDopplerShift',maxDopp,'PathDelays',pathDelays, ...
    'AveragePathGains',pathGains,'SampleRate',sRate, ...
    'RandomStream','mt19937ar with seed');

data = randi(s1,[0 1],modOrd*numTones,1);
modOut = qam(data);
```

Apply OFDM modulation and pass the signal through the channel.

```
y = ofdmmod(modOut,nFFT,cpLen,nullIdx);
[fadSig, pg] = chan(y);
```

Determine symbol sampling offset. OFDM demodulate the received signal.

```
symOffset = min(max(sampIdx),cpLen)
```

```
symOffset = 9
```

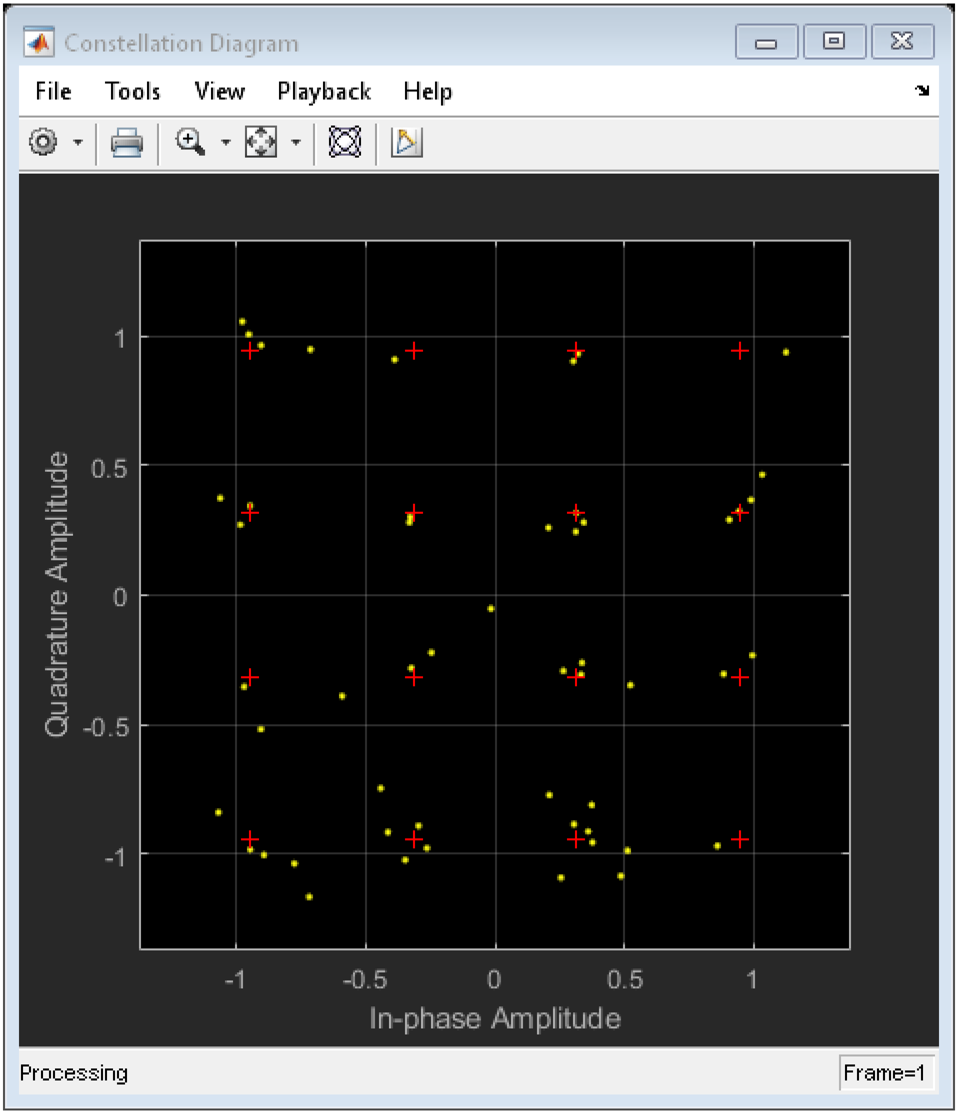
```
x = ofdmdemod(fadSig,nFFT,cpLen,symOffset,nullIdx); % with a time shift
```

Convert path gains, `pg`, to scalar taps gains. Use the tap gains for equalization during signal recovery.

```
hImp = complex(zeros(1,nFFT));
hImp(:,sampIdx) = mean(pg,1);
hall = fftshift(fft(hImp.'),1);
dataIdx = double(setdiff((1:nFFT)',nullIdx));
h = hall(dataIdx);
```

Equalize the signal.

```
eqH = conj(h)./(conj(h).*h);
eqSig = eqH.*x;

cdScope = comm.ConstellationDiagram('ShowReferenceConstellation',true, ...
    'ReferenceConstellation',constellation(qam));
cdScope(eqSig);
```



Demodulate the 16-QAM symbols to recover the signal. Compute the bit error rate.

```
qamdem = comm.RectangularQAMDemodulator('ModulationOrder',2^modOrd, ...
    'BitOutput',true,'NormalizationMethod','Average power');
rxBits = qamdem(eqSig);
```

```
numErr = biterr(data,rxBits);
disp(['Number of bit errors: ' num2str(numErr) ' out of ' num2str(length(data)) ' bits
```

```
Number of bit errors: 3 out of 208 bits.
```

OFDM Demodulation with Null and Pilot Packing

OFDM-demodulate data input that includes null and pilot packing.

Initialize input parameters, defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft     = 64;
cplen    = 16;
nSym     = 10;
nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';
numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
dataIn = complex(randn(numDataCarrs,nSym),randn(numDataCarrs,nSym));
pilots = repmat(pskmod((0:3).',4),1,nSym);
y2 = ofdmmod(dataIn,nfft,cplen,nullIdx,pilotIdx,pilots);
```

Demodulate the OFDM symbols. Compare the results to the original input data to show that there is negligible difference between the demodulated signal and the original data and pilot signals.

```
symOffset = cplen;
[x2,rxPilots] = ofdmdemod(y2,nfft,cplen,symOffset,nullIdx,pilotIdx);
max(x2-dataIn)
```

```
ans = 1×10 complex
10^-15 ×

   0.5551 + 0.2220i   0.2220 + 0.4441i   0.4441 - 0.2220i   0.4718 - 0.3331i  -0.1665
```

```
max(rxPilots-pilots)
```

```
ans = 1×10 complex
10^-15 ×
```

```
   0.0000 + 0.3331i   0.1837 - 0.2220i  -0.4441 - 0.2776i   0.2220 + 0.2220i   0.2220 
```

Input Arguments

`ofdmSig` — Modulated OFDM symbols

2-D array of complex symbols

Modulated OFDM symbols, specified as a 2-D array of complex symbols.

- If `cplen` is a scalar, the array size is ((`nfft` + `cplen`) × N_{Sym})-by-N_{R}.
- If `cplen` is a row vector, the array size is ((`nfft` × N_{Sym}) + `sum`(`cplen`))-by-N_{R}.

  N_{Sym} is the number of symbols per antenna and N_{R} is the number of receive antennas.

Data Types: `double` | `single`
Complex Number Support: Yes

`nfft` — FFT length

integer greater than or equal to 8

FFT length, specified as an integer greater than or equal to 8. `nfft` is equivalent to the number of subcarriers used in the demodulation process.

Data Types: `double`

`cplen` — Cyclic prefix length

scalar | row vector of length N_{Sym}

Cyclic prefix length, specified as a scalar or as a row vector of length N_{Sym}.

- When you specify `cplen` as a scalar, the cyclic prefix length is the same for all symbols through all antennas.
- When you specify `cplen` as a row vector of length N_{Sym}, the cyclic prefix length can vary across symbols but remains the same length through all antennas.

Data Types: `double`

`symOffset` — Symbol sampling offset

`cplen` (default) | scalar | row vector

Symbol sampling offset, specified as values from 0 to `cplen`.

- If you do not specify `symOffset`, the default value is an offset equal to `cplen`.
- If you specify `symOffset` as a scalar, the same offset is used for all symbols.
- If you specify `symOffset` as a row vector, the offset value can be different for each symbol.

For information, see “Windowing and Symbol Offset” on page 1-766.

Data Types: `double`

`nullidx` — Indices of null subcarrier locations

column vector

Indices of null subcarrier locations, specified as a column vector with element values from 1 to `nfft`. If you specify `nullidx`, the number of rows in `outSym` is `(nfft-length(nullidx))`. For information, see “Subcarrier Allocation and Guard Bands” on page 1-765.

Data Types: `double`

`pilotidx` — Indices of pilot subcarrier locations

column vector

Indices of pilot subcarrier locations, specified as a column vector with element values from 1 to `nfft`. If you specify `pilotidx`, the number of rows in `outSym` is `(nfft-length(nullidx)-length(pilotidx))`.For information, see “Subcarrier Allocation and Guard Bands” on page 1-765.

Data Types: `double`

Output Arguments

`outSym` — Output demodulated symbols

numeric 3-D array

Output demodulated symbols, returned as an N_D-by-N_{Sym}-by-N_R numeric array of symbols. N_D must equal `nfft – length(nullidx) – length(pilotidx)`. N_{Sym} is the number of OFDM symbols per antenna. N_R is the number of receive antennas. For information, see “OFDM Demodulation” on page 1-764.

`pilots` — Pilot subcarriers
numeric 3-D array

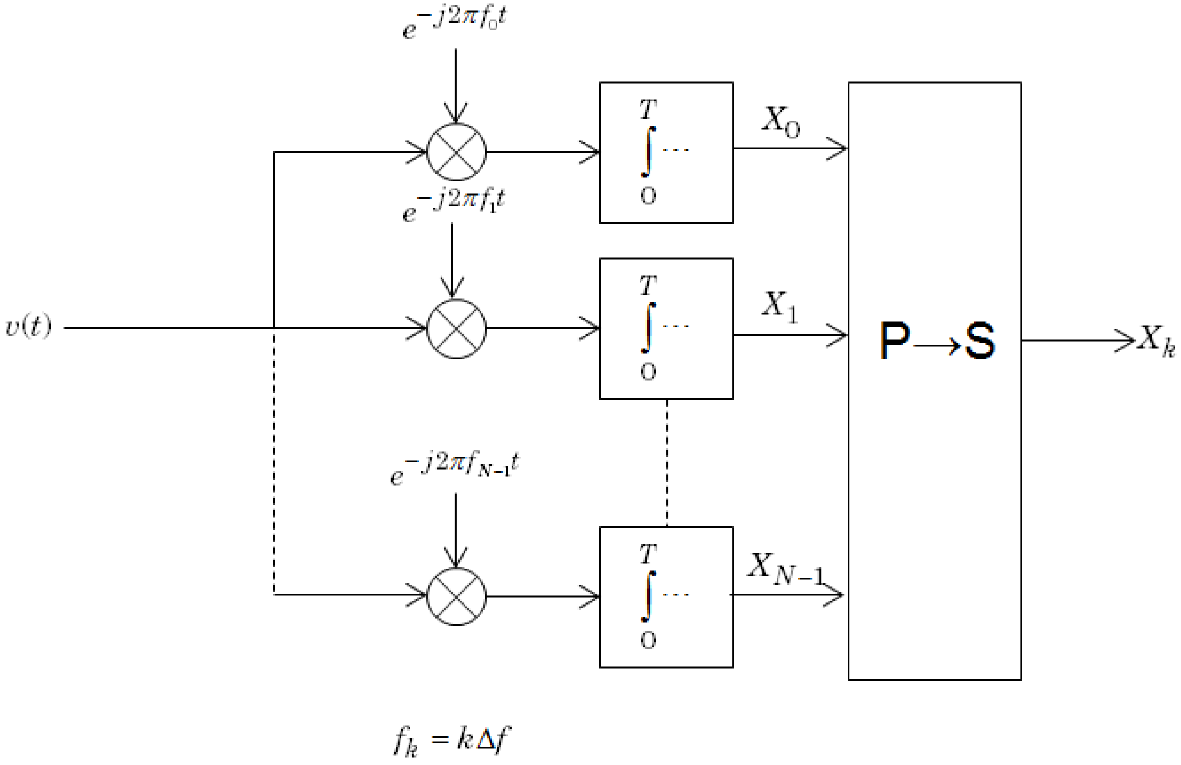
Pilot subcarriers, returned as an N_{Pilot}-by-N_{Sym}-by-N_{R} numeric array of symbols. N_{Pilot} must equal the length of `pilotidx`. N_{Sym} is the number of OFDM symbols per antenna. N_{R} is the number of receive antennas. The function assumes that the pilot subcarrier locations are the same across each OFDM symbol and transmit antenna. Use the `comm.OFDMDemodulator` to vary pilot subcarrier locations across OFDM symbols or antennas.

Definitions

OFDM Demodulation

An OFDM demodulator demultiplexes a multi-subcarrier time-domain signal using orthogonal frequency division modulation.

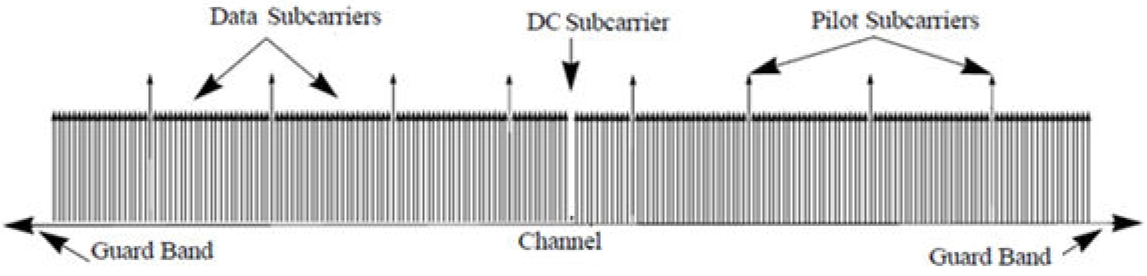
The OFDM demodulation uses an FFT operation that results in N parallel data streams. An OFDM demodulator consists of a bank of N correlators, with one correlator assigned to each OFDM subcarrier, followed by a parallel-to-serial conversion.



Subcarrier Allocation and Guard Bands

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, of guard band subcarriers.



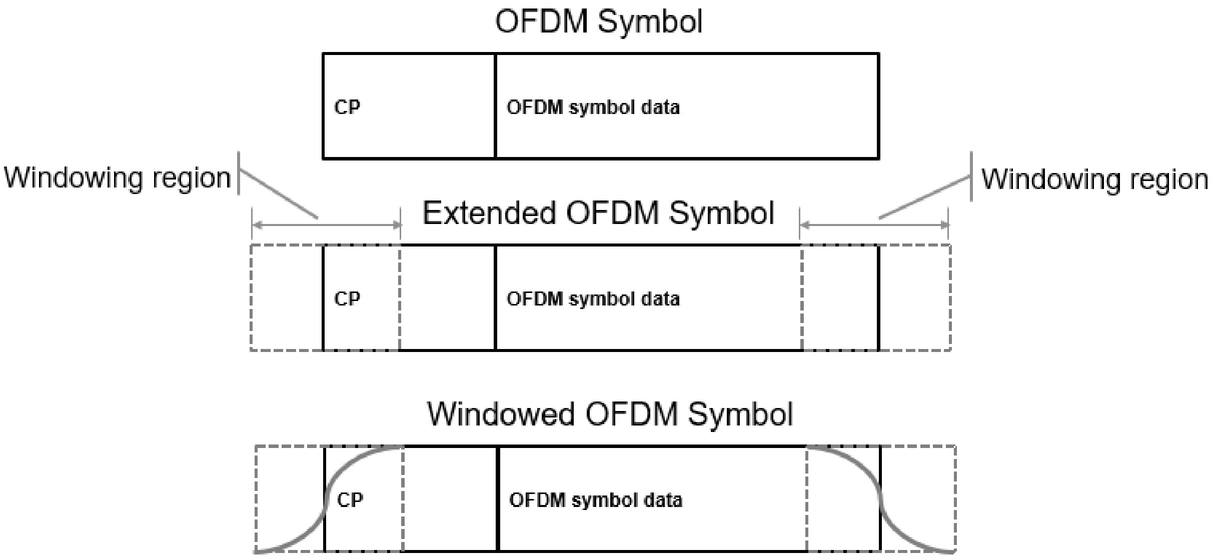
- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.
  - The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
  - The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

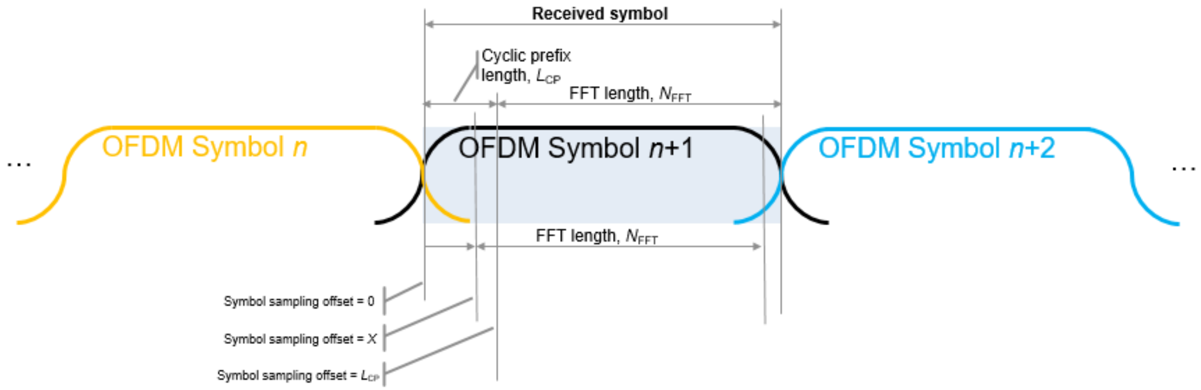
Windowing and Symbol Offset

To reduce intersymbol interference (ISI) introduced by signal windowing applied at the transmitter, the function applies a fractional symbol offset before demodulation of each OFDM symbol. Signal windowing is often applied to transmitted OFDM symbols to smooth the discontinuity between consecutive OFDM symbols. Windowing reduces intersymbol out-of-band emissions but increases ISI.

The windowed OFDM symbol consists of the cyclic prefix (CP), ODFM symbol data, plus windowing regions at the beginning and end of the symbol. The leading and trailing windowing shoulders have tails as shown in the figure.



To reduce ISI, you can align signal sample timing by specifying a symbol sampling offset that gets applied before OFDM symbol demodulation.


Received symbol
Cyclic prefix length, L_{CP}
FFT length, N_{FFT}
...
OFDM Symbol n
OFDM Symbol n+1
OFDM Symbol n+2
...
FFT length, N_{FFT}
Symbol sampling offset = 0
Symbol sampling offset = X
Symbol sampling offset = L_{CP}

Specify the symbol sampling offset as a value from 0 to L_{CP}.

- When the symbol sampling offset is a scalar from 0 to L_{CP}, the FFT window begins at the X+1 sample of the CP length.

- When the symbol sampling offset is zero, no offset is applied and the FFT window starts at the first sample of the symbol.
- When the symbol sampling offset is the cyclic prefix length, L_{CP}, the FFT window begins after the last CP sample. This offset is the default setting if symbol sampling offset is not specified.

See Also

Functions
genqamdemod | ofdmmod | qamdemod

System Objects
comm.GeneralQAMDemodulator | comm.OFDMDemodulator | comm.OQPSKDemodulator | comm.RectangularQAMDemodulator

Introduced in R2018a

ofdmmod

Modulate frequency-domain signal using orthogonal frequency division multiplexing (OFDM)

Syntax

```
ofdmSig = ofdmmod(inSym,nfft,cplen)
ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx)
ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx,pilotidx,pilots)
```

Description

`ofdmSig = ofdmmod(inSym,nfft,cplen)` performs OFDM modulation on the frequency-domain input data subcarriers, `inSym`, using an FFT size specified by `nfft` and cyclic prefix length specified by `cplen`. For information, see “OFDM Modulation” on page 1-774.

`ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx)` inserts null subcarriers into the frequency domain input data signal prior to performing OFDM modulation. The null subcarriers are inserted at index locations from 1 to `nfft`, as specified by `nullidx`. For this syntax, the number of rows in the input `inSym` must be `nfft – length(nullidx)`. Use null carriers to account for guard bands and DC subcarriers. For information, see “Subcarrier Allocation and Guard Bands” on page 1-775.

`ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx,pilotidx,pilots)` inserts null and pilot subcarriers into the frequency domain input data symbols prior to performing OFDM modulation. The null subcarriers are inserted at the index locations specified by `nullidx`. The pilot subcarriers, `pilots`, are inserted at the index locations specified by `pilotidx`. For this syntax, the number of rows in the input `inSym` must be `nfft – length(nullidx) – length(pilotidx)`. The function assumes pilot subcarrier locations are the same across each OFDM symbol and transmit antenna.

Examples

OFDM Modulation Over Two Antennas

OFDM-modulate a fully packed input over two transmit antennas.

Initialize input parameters, generate random data, and perform OFDM modulation.

```
nfft  = 128;
cplen = 16;
nSym  = 5;
nt    = 2;
dataIn = complex(randn(nfft,nSym,nt),randn(nfft,nSym,nt));

y1 = ofdmmod(dataIn,nfft,cplen);
```

Apply OFDM Assigning Null Subcarriers

Apply OFDM modulation assigning null subcarriers.

Initialize input parameters and generate random data.

```
M = 16; % Modulation order for 16QAM
nfft  = 64;
cplen = 16;
nSym  = 10;
nullIdx  = [1:6 33 64-4:64]';
numDataCarrs = nfft-length(nullIdx);
inSig = randi([0 M-1],numDataCarrs,nSym);
```

QAM modulate data. Perform OFDM modulation.

```
qamSym = qammod(inSig,M,'UnitAveragePower',true);
outSig = ofdmmod(qamSym,nfft,cplen,nullIdx);
```

Perform OFDM Modulation Varying Cyclic Prefix per Symbol

Perform OFDM modulation to input frequency domain data signal varying cyclic prefix length applied to each symbol.

Initialize input parameters and generate random data.

```
M = 16; % Modulation order for 16QAM
nfft  = 64;
cplen = [4 8 10 7 2 2 4 11 16 3];
nSym  = 10;
nullIdx  = [1:6 33 64-4:64]';
numDataCarrs = nfft-length(nullIdx);
inSig = randi([0 M-1],numDataCarrs,nSym);
```

QAM modulate data. Perform OFDM modulation.

```
qamSym = qammod(inSig,M,'UnitAveragePower',true);
outSig = ofdmmod(qamSym,nfft,cplen,nullIdx);
```

Apply OFDM to QPSK Signal Spatially Multiplexed Over Two Antennas

Apply OFDM modulation to a QPSK signal that is spatially multiplexed over two transmit antennas.

Initialize input parameters and generate random data for each antenna.

```
M = 4; % Modulation order for QPSK
nfft  = 64;
cplen = 16;
nSym  = 5;
nt    = 2;
nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';
numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
pilots = repmat(pskmod((0:M-1).',M),1,nSym,2);

ant1 = randi([0 M-1],numDataCarrs,nSym);
ant2 = randi([0 M-1],numDataCarrs,nSym);
```

QPSK modulate data individually for each antenna. Perform OFDM modulation.

```
qpskSym(:,:,1) = pskmod(ant1,M);
qpskSym(:,:,2) = pskmod(ant2,M);
y1 = ofdmmod(qpskSym,nfft,cplen,nullIdx,pilotIdx,pilots);
```

OFDM Modulation with Null and Pilot Packing

OFDM-modulate data input, specifying null and pilot packing.

Initialize input parameters, defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft     = 64;
cplen    = 16;
nSym     = 10;

nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';

numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
dataIn = complex(randn(numDataCarrs,nSym),randn(numDataCarrs,nSym));
pilots = repmat(pskmod((0:3).',4),1,nSym);

y2 = ofdmmod(dataIn,nfft,cplen,nullIdx,pilotIdx,pilots);
```

Input Arguments

`inSym` — Input data subcarriers

numeric 3-D array

Input data subcarriers, specified as an N_D-by-N_{Sym}-by-N_T numeric array of symbols. The number of data subcarriers, N_D, must equal `nfft – length(nullidx) – length(pilotidx)`. N_{Sym} is the number of OFDM symbols per transmit antenna, N_T is the number of transmit antennas.

Input data symbols to an OFDM modulator are typically created with a baseband digital modulator, such as `qammod`.

Data Types: `double` | `single`
Complex Number Support: Yes

`nfft` — FFT length

integer greater than or equal to 8

FFT length, specified as an integer greater than or equal to 8. `nfft` is equivalent to the number of subcarriers used in the modulation process.

Data Types: `double`

`cplen` — Cyclic prefix length

scalar | row vector of length N_{Sym}

Cyclic prefix length, specified as a scalar or as a row vector of length N_{Sym}.

- When you specify `cplen` as a scalar, the cyclic prefix length is the same for all symbols through all antennas.
- When you specify `cplen` as a row vector of length N_{Sym}, the cyclic prefix length can vary across symbols but remains the same length through all antennas.

For more information, see “Guard Intervals” on page 1-776.

Data Types: `double`

`nullidx` — Indices of null subcarrier locations

column vector

Indices of null subcarrier locations, specified as a column vector with element values from 1 to `nfft`.

Data Types: `double`

`pilotidx` — Indices of pilot subcarrier locations

column vector

Indices of pilot subcarrier locations, specified as a column vector with element values from 1 to `nfft`.

Data Types: `double`

`pilots` — Pilot subcarriers

numeric 3-D array

Pilot subcarriers, specified as an N_{Pilot}-by-N_{Sym}-by-N_{T} numeric array of symbols. N_{Pilot} must equal the length of `pilotidx`. N_{Sym} is the number of OFDM symbols per transmit antenna. N_{T} is the number of transmit antennas. The function assumes pilot subcarrier locations are the same across each OFDM symbol and transmit antenna. Use the `comm.OFDMModulator` to vary pilot subcarrier locations across OFDM symbols or antennas.

Data Types: `double` | `single`

Output Arguments

ofdmSig — Modulated OFDM symbols
2-D array of complex symbols

Modulated OFDM symbols, returned as a 2-D array of complex symbols.

- If `cplen` is a scalar, the array size is ((`nfft` + `cplen`) × N_{Sym})-by-N_{T}.
- If `cplen` is a row vector, the array size is ((`nfft` × N_{Sym}) + `sum`(`cplen`))-by-N_{T}.

  N_{Sym} is the number of symbols per transmit antenna and N_{T} is the number of transmit antennas.

Definitions

OFDM Modulation

An OFDM modulator multiplexes a frequency-domain input signal over multiple subcarriers using orthogonal frequency division modulation.

The OFDM operation divides a high-rate transmit data stream into N lower data rate substreams. The individual substreams are sent over N parallel and orthogonal subchannels. Using an inverse fast Fourier transform (IFFT) to process the transmission data, OFDM can be transmitted with a single radio. Intersymbol interference (ISI) is reduced because the lower data rate substreams have symbol durations larger than the channel delay spread.

The output is a baseband representation of the modulated signal:

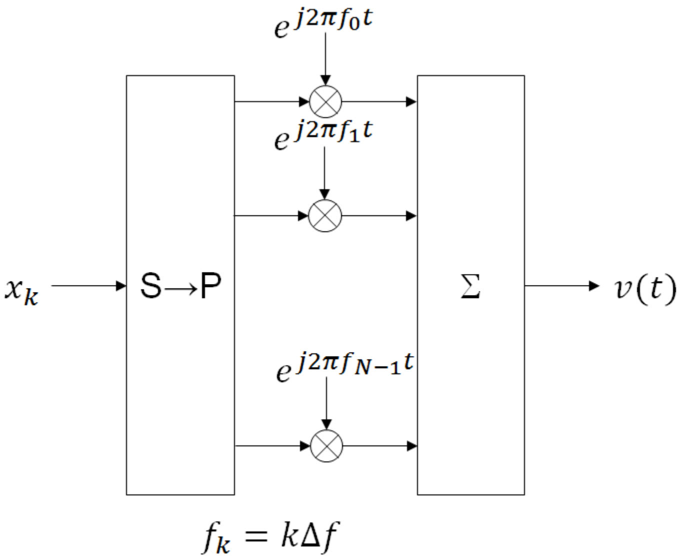
$$v(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi k \Delta f t}, \quad 0 \le t \le T,$$

where $\{X_{\text{k}}\}$ are data symbols, N is the number of subcarriers, and T is the OFDM symbol time. Using a subcarrier spacing of $\Delta f = 1/T$, the subcarriers are orthogonal over each symbol period, as expressed in this equation:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi m \Delta f t}\right)^* \left(e^{j2\pi n \Delta f t}\right) dt = \frac{1}{T}\int_0^T e^{j2\pi(m-n)\Delta f t}\ dt = 0 \quad \text{for } m \ne n.$$

The data symbols, X_k, are typically complex and can be from any digital modulation alphabet (for example, QPSK, 16-QAM, 64-QAM).

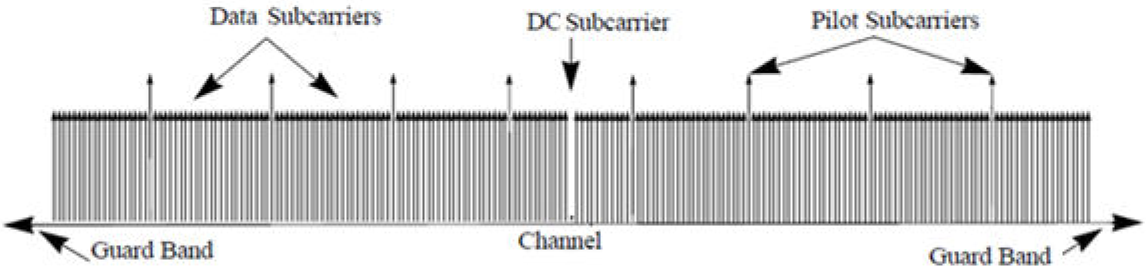
An OFDM modulator consists of a serial-to-parallel conversion followed by a bank of N complex modulators, individually corresponding to each OFDM subcarrier.



Subcarrier Allocation and Guard Bands

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, of guard band subcarriers.



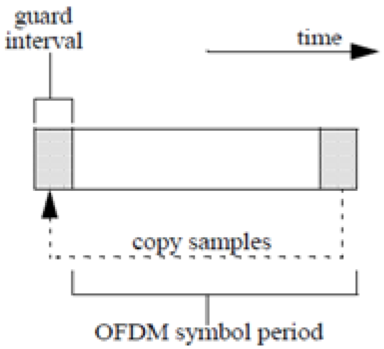
- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.
  - The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
  - The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

Guard Intervals

Similar to guard bands, guard intervals are used in OFDM to protect the integrity of transmitted signals by reducing intersymbol interference.

Assignment of guard intervals is analogous to the assignment of guard bands. You can model guard intervals to provide temporal separation between OFDM symbols. The guard intervals help preserve intersymbol orthogonality after the signal passes through time-dispersive channels. Guard intervals are created by using cyclic prefixes. Cyclic prefix insertion copies the last part of an OFDM symbol as the first part of the OFDM symbol.



As long as the span of the time dispersion does not exceed the duration of the cyclic prefix, the benefit of cyclic prefix insertion is maintained.

Inserting a cyclic prefix results in a fractional reduction of user data throughput because the cyclic prefix occupies bandwidth that could be used for data transmission.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`genqammod` | `ofdmdemod` | `qammod`

System Objects

`comm.GeneralQAMModulator` | `comm.OFDMModulator` | `comm.OQPSKModulator` | `comm.RectangularQAMModulator`

Introduced in R2018a

oqpskdemod

(To be removed) Offset quadrature phase shift keying demodulation

Note `oqpskdemod` will be removed in a future release. Use `comm.OQPSKDemodulator` instead.

Syntax

```
z = oqpskdemod(y)
z = oqpskdemod(y,ini_phase)
```

Description

`z = oqpskdemod(y)` demodulates the complex envelope of an OQPSK modulated signal. The function upsamples by a factor of 2, because OQPSK does not permit an odd number of samples per symbol.

`z = oqpskdemod(y,ini_phase)` specifies the initial phase of the modulated signal.

Examples

Modulate and Demodulate OQPSK Signal in AWGN

Generate random 4-ary data. Create modulator and demodulator objects.

```
dataIn = randi([0 3],100,1);
oqpskmod = comm.OQPSKModulator;
oqpskdemod = comm.OQPSKDemodulator;
```

OQPSK modulate the data, and pass it through an AWGN channel.

```
txSig = oqpskmod(dataIn);
rxSig = awgn(txSig,10);
```

OQPSK demodulate the received signal. Determine the number of symbol errors.

```
dataOut = oqpskdemod(rxSig);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 79
```

Input Arguments

y — OQPSK-modulated input signal

vector | matrix

OQPSK-modulated input signal, specified as a complex vector or matrix. If y is a matrix, the function processes the columns independently.

Data Types: single | double
Complex Number Support: Yes

ini_phase — Initial phase

0 (default) | scalar

Initial phase of the OQPSK modulation, specified in radians as a real scalar.

Example: pi/4

Data Types: double | single

Output Arguments

z — OQPSK-demodulated output signal

vector | matrix

OQPSK-demodulated output signal, returned as a vector or matrix having the same number of columns as input signal y.

Data Types: double | single

See Also

Functions

`modnorm`

System Objects

`comm.OQPSKDemodulator` | `comm.OQPSKModulator`

Topics

“Phase Modulation”

Introduced before R2006a

oqpskmod

(To be removed) Offset quadrature phase shift keying modulation

Note `oqpskmod` will be removed in a future release. Use `comm.OQPSKModulator` instead.

Syntax

```
y = oqpskmod(x)
y = oqpskmod(x,ini_phase)
```

Description

`y = oqpskmod(x)` modulates the input signal, `x`, using offset quadrature phase shift keying (OQPSK). The function upsamples by a factor of 2, because OQPSK does not permit an odd number of samples per symbol.

`y = oqpskmod(x,ini_phase)` specifies the initial phase of the modulated signal.

Examples

Modulate and Demodulate OQPSK Signal in AWGN

Generate random 4-ary data. Create modulator and demodulator objects.

```
dataIn = randi([0 3],100,1);
oqpskmod = comm.OQPSKModulator;
oqpskdemod = comm.OQPSKDemodulator;
```

OQPSK modulate the data, and pass it through an AWGN channel.

```
txSig = oqpskmod(dataIn);
rxSig = awgn(txSig,10);
```

OQPSK demodulate the received signal. Determine the number of symbol errors.

```
dataOut = oqpskdemod(rxSig);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 79
```

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of `x` must have values in the range of [0, 3].

Data Types: `double` | `single`

`ini_phase` — Initial phase

0 (default) | scalar

Initial phase of the OQPSK modulation, specified in radians as a real scalar.

Example: `pi/4`

Data Types: `double` | `single`

Output Arguments

y — OQPSK-modulated output signal

vector | matrix

Complex baseband representation of an OQPSK-modulated output signal, returned as a vector or matrix. The columns of `y` represent independent channels.

Data Types: `double` | `single`

See Also

Functions
`modnorm`

System Objects
`comm.OQPSKDemodulator` | `comm.OQPSKModulator`

Topics

“Phase Modulation”

Introduced before R2006a

pamdemod

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

Demodulate PAM Signal

Modulate and demodulate random integers using pulse amplitude modulation. Verify that the output data matches the original data.

Set the modulation order and generate 100 M-ary data symbols.

```
M = 12;
dataIn = randi([0 M-1],100,1);
```

Perform modulation and demodulation operations.

```
modData = pammod(dataIn,M);
dataOut = pamdemod(modData,M);
```

Compare the first five symbols.

```
[dataIn(1:5) dataOut(1:5)]
```

```
ans = 5×2

     9     9
    10    10
     1     1
    10    10
     7     7
```

Verify that there are no symbol errors in the entire sequence.

```
symErrors = symerr(dataIn,dataOut)
```

```
symErrors = 0
```

See Also

pammod | pskdemod | pskmod | qamdemod | qammod

Topics

“Digital Modulation”
“Comparing Theoretical and Empirical Error Rates”

Introduced before R2006a

pammod

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

Modulate Data Symbols with PAM

Generate random data symbols and apply pulse amplitude modulation.

Set the modulation order.

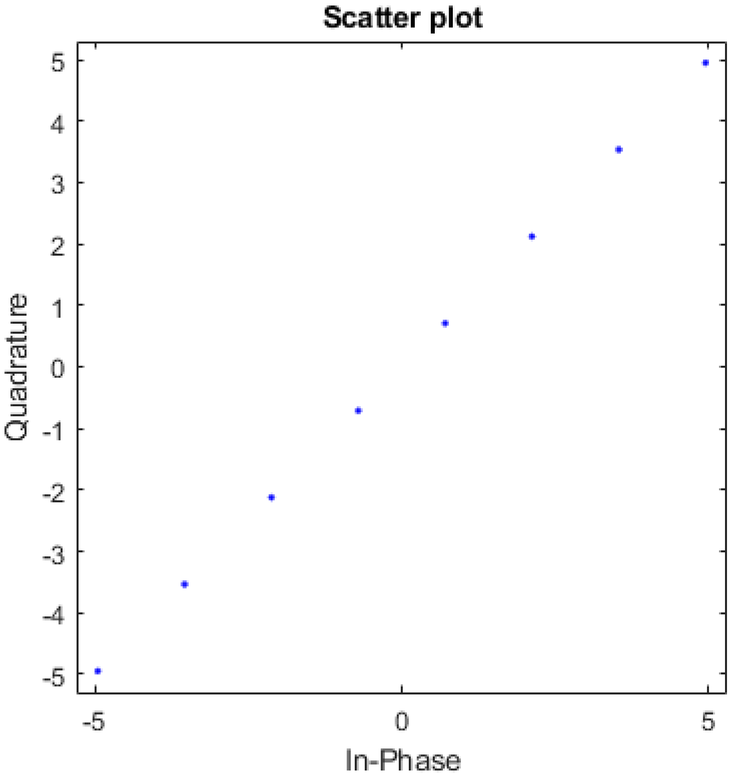
```
M = 8;
```

Generate random integers and apply PAM modulation having an initial phase of π/4.

```
data = randi([0 M-1],100,1);
modData = pammod(data,M,pi/4);
```

Display the PAM constellation diagram.

```
scatterplot(modData)
```



See Also

pamdemod | pskdemod | pskmod | qamdemod | qammod

Topics

“Digital Modulation”
“Comparing Theoretical and Empirical Error Rates”

Introduced before R2006a

plot (channel)

(To be removed) Plot channel characteristics with channel visualization tool

Syntax

```
plot(h)
```

Note This function will be removed in a future release. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

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.

ExamplesVisualize RF Impairments

Apply various RF impairments to a QAM signal. Observe the effects by using constellation diagrams, time-varying error vector magnitude (EVM) plots, and spectrum plots. Estimate the equivalent signal-to-noise ratio (SNR).

Initialization

Set the sample rate, modulation order, and SNR. Calculate the reference constellation points.

```
fs = 1000;
M = 16;
snrdB = 30;
refConst = qammod(0:M-1,M,'UnitAveragePower',true);
```

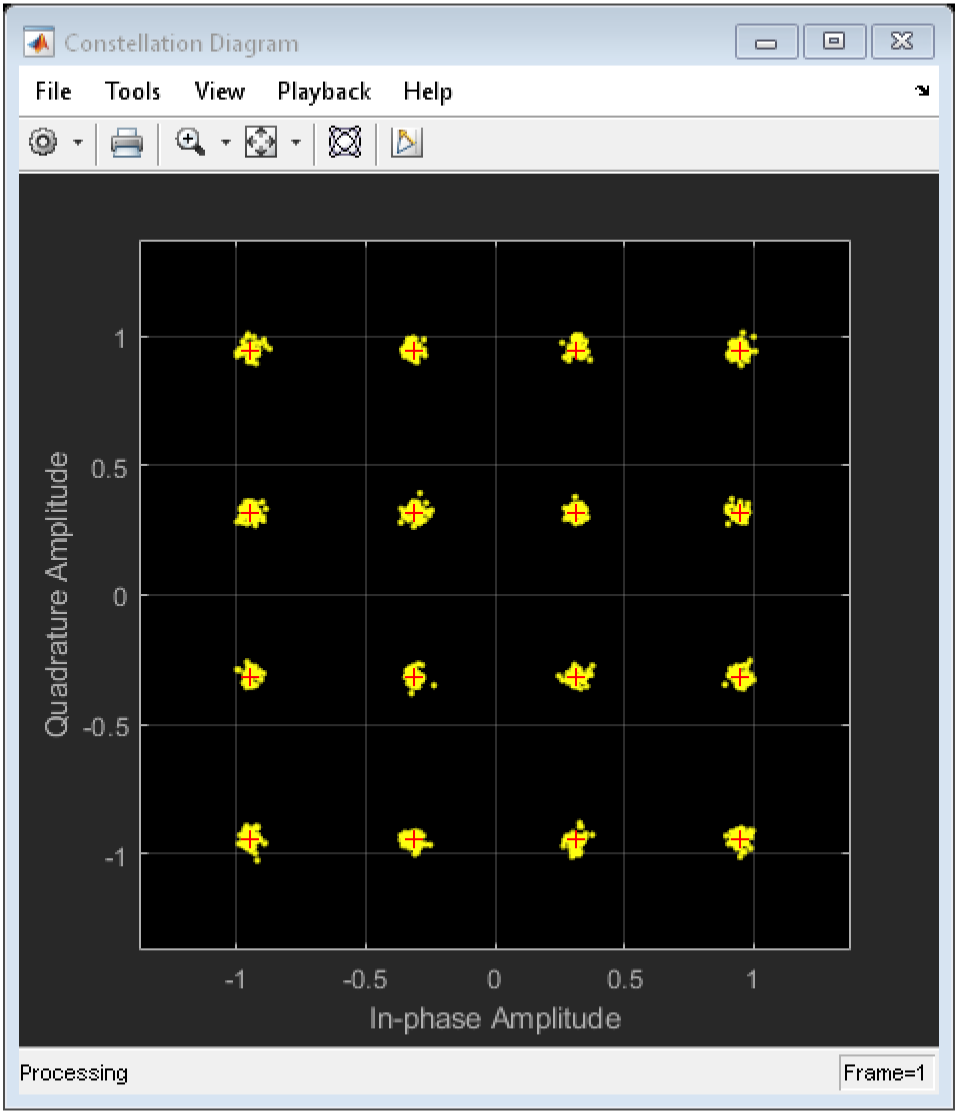
Create constellation diagram and time scope objects to visualize the impairment effects.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
```

White Noise

Generate a 16-QAM signal, and pass it through an AWGN channel. Plot its constellation.

```
data = randi([0 M-1],1000,1);
modSig = qammod(data,M,'UnitAveragePower',true);
noisySig = awgn(modSig,snrdB);

constDiagram(noisySig)
```



Estimate the EVM of the noisy signal from the reference constellation points.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power');
```

```
rmsEVM = evm(noisySig)
```

```
rmsEVM = 3.1768
```

The modulation error rate (MER) closely corresponds to the SNR. Create an MER object, and estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(noisySig)
```

```
snrEst = 30.1071
```

The estimate is quite close to the specified SNR of 30 dB.

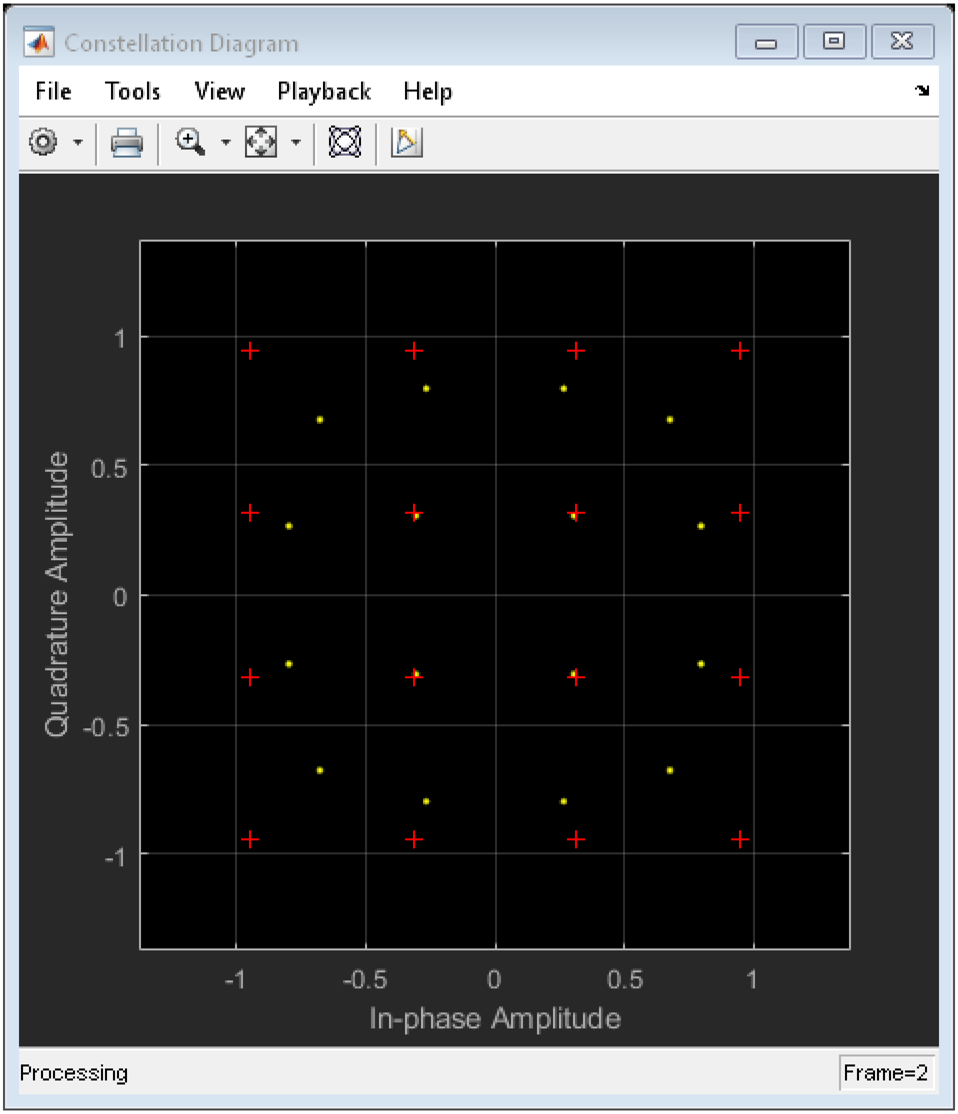
Amplifier Distortion

Create an amplifier using the memoryless nonlinearity object.

```
amp = comm.MemorylessNonlinearity('IIP3',38,'AMPMConversion',0);
```

Pass the modulated signal through the nonlinear amplifier, and plot its constellation diagram.

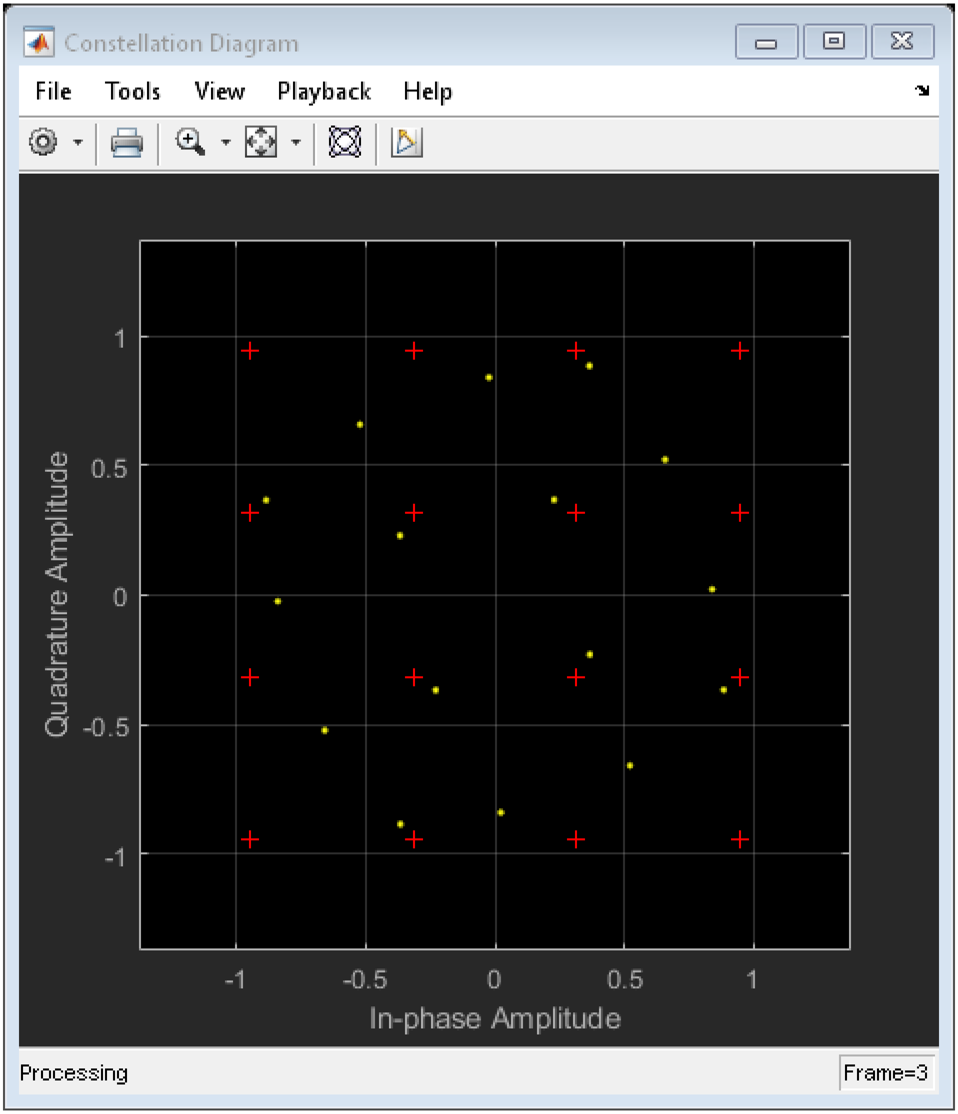
```
txSig = amp(modSig);
constDiagram(txSig)
```



The corner points of the constellation have moved toward the origin due to amplifier gain compression.

Introduce a small AM/PM conversion, and display the received signal constellation.

```
amp.AMPMConversion = 1;
txSig = amp(modSig);
constDiagram(txSig)
```

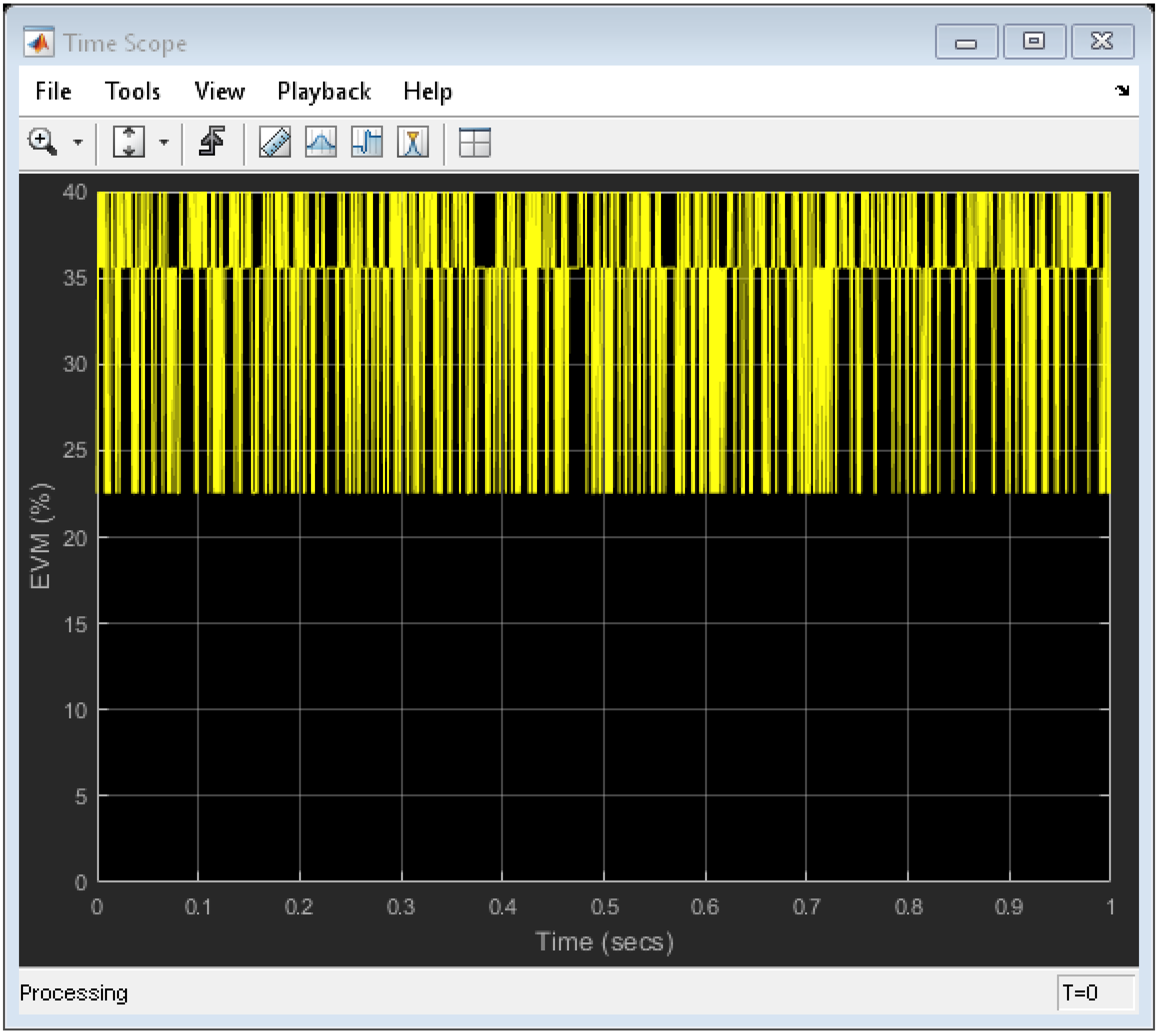


The constellation has rotated due to the AM/PM conversion. To compute the time-varying EVM, release the EVM object and set the AveragingDimensions property to 2. To estimate the EVM against an input signal, omit the ReferenceSignalSource property definition. This method produces more accurate results.

```
evm = comm.EVM('AveragingDimensions',2);
evmTime = evm(modSig,txSig);
```

Plot the time-varying EVM of the distorted signal.

```
timeScope(evmTime)
```



Compute the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 35.5919
```

Compute the MER.

```
mer = comm.MER;
snrEst = mer(modSig,txSig)
```

```
snrEst = 8.1392
```

The SNR (≈8 dB) is reduced from its initial value (∞) due to amplifier distortion.

Specify input power levels ranging from 0 to 40 dBm. Convert those levels to their linear equivalent in W. Initialize the output power vector.

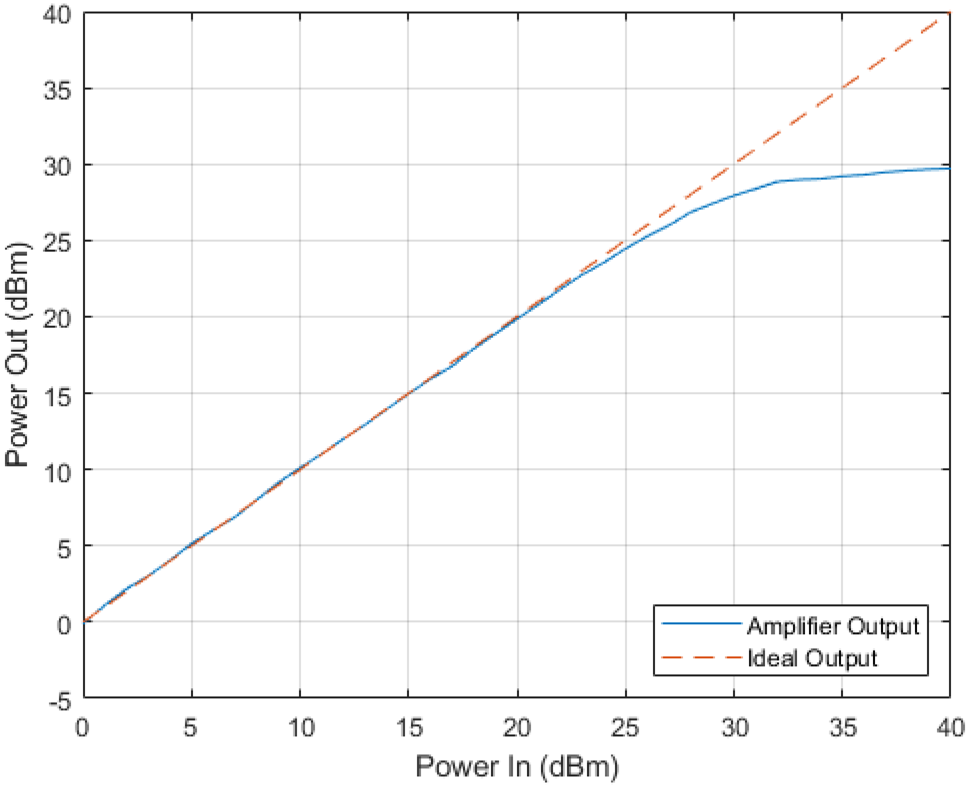
```
powerIn = 0:40;
pin = 10.^((powerIn-30)/10);
powerOut = zeros(length(powerIn),1);
```

Measure the amplifier output power for the range of input power levels.

```
for k = 1:length(powerIn)
    data = randi([0 15],1000,1);
    txSig = qammod(data,16,'UnitAveragePower',true)*sqrt(pin(k));
    ampSig = amp(txSig);
    powerOut(k) = 10*log10(var(ampSig))+30;
end
```

Plot the power output versus power input curve.

```
figure
plot(powerIn,powerOut,powerIn,powerIn,'--')
legend('Amplifier Output','Ideal Output','location','se')
xlabel('Power In (dBm)')
ylabel('Power Out (dBm)')
grid
```



The output power levels off at 30 dBm. The amplifier exhibits nonlinear behavior for input power levels greater than 25 dBm.

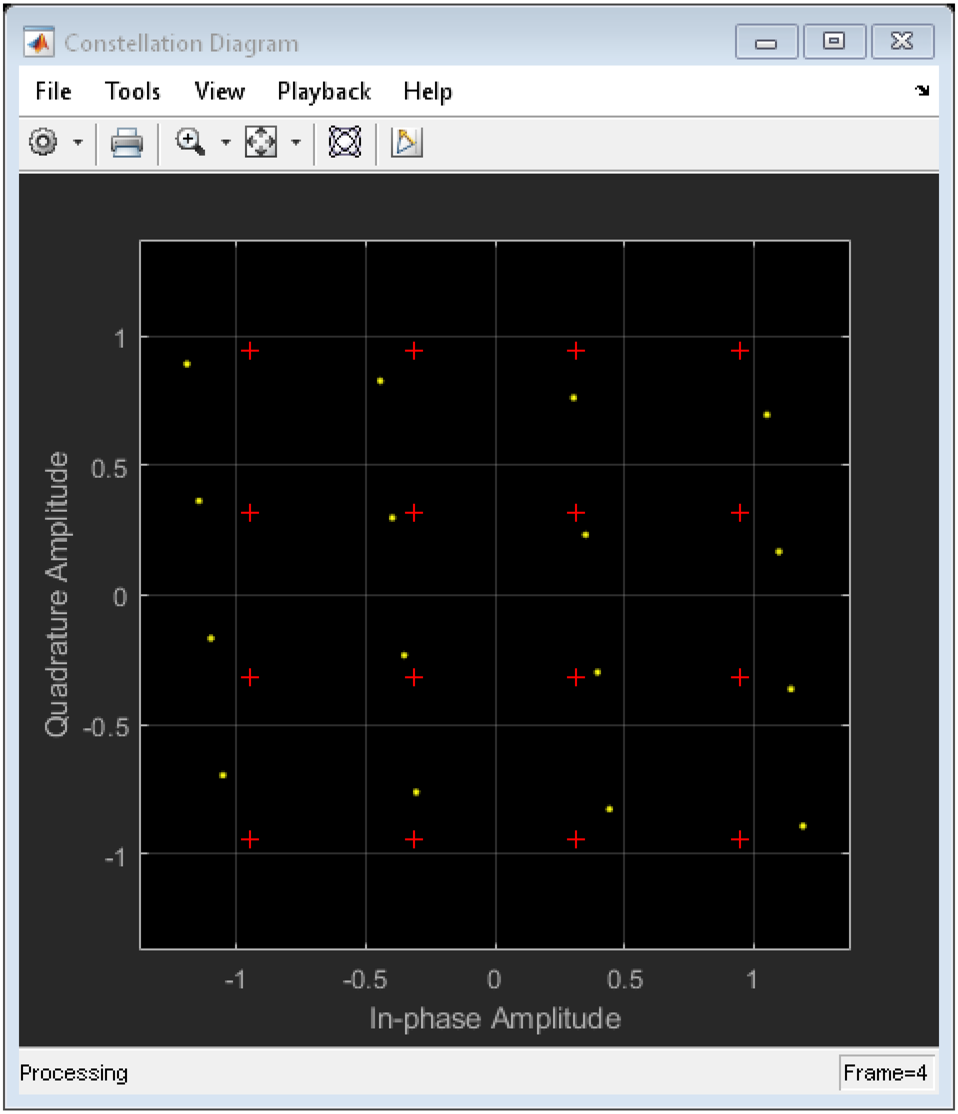
I/Q Imbalance

Apply an amplitude and phase imbalance to the modulated signal.

```
ampImb = 3;
phImb = 10;
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(modSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(modSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the received constellation.

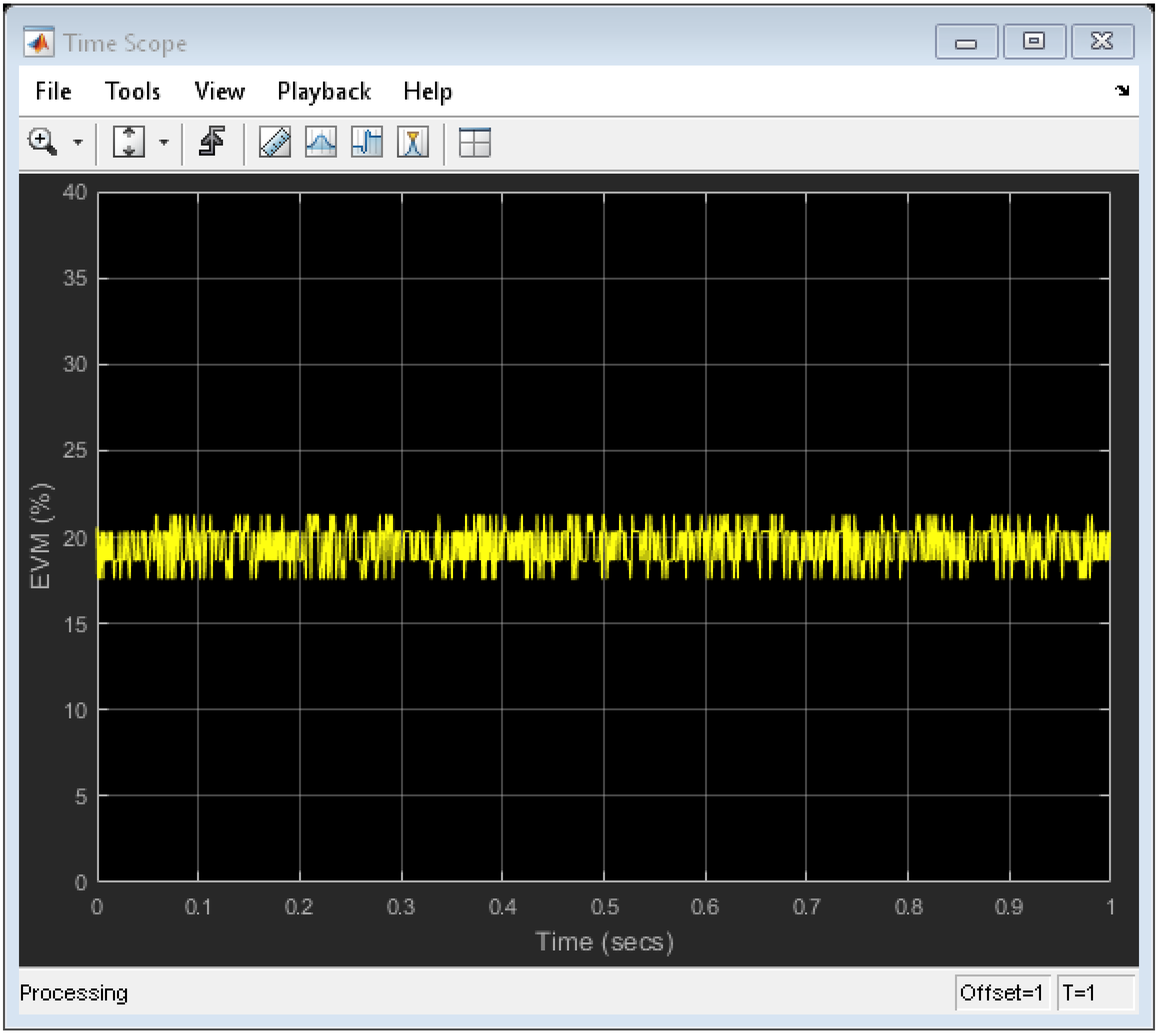
```
constDiagram(rxSig)
```



The magnitude and phase of the constellation has changed as a result of the I/Q imbalance.

Calculate and plot the time-varying EVM.

```
evmTime = evm(modSig,rxSig);
timeScope(evmTime)
```



The EVM exhibits a behavior that is similar to that experienced with a nonlinear amplifier though the variance is smaller.

Create a 100 Hz sine wave having a 1000 Hz sample rate.

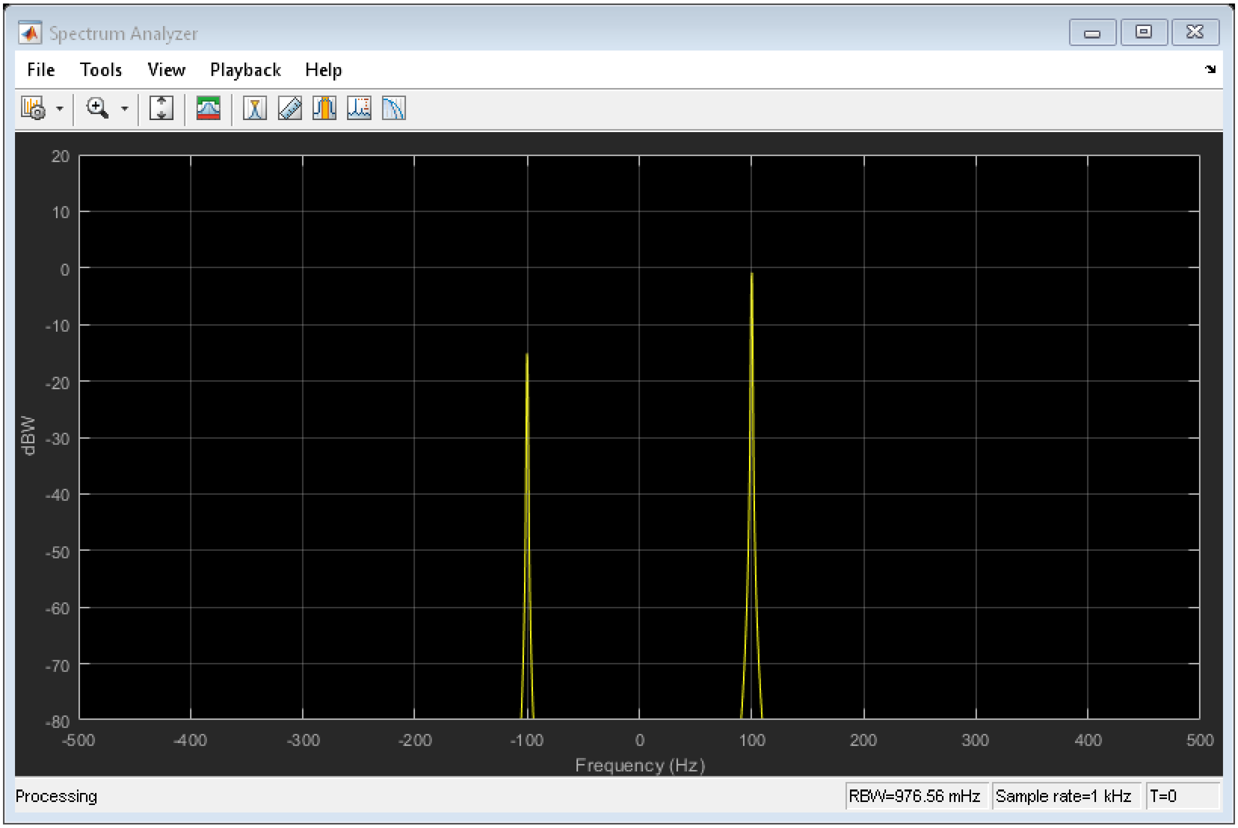
```
sinewave = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
    'SamplesPerFrame',1e4,'ComplexOutput',true);

x = sinewave();
```

Apply the same 3 dB and 10 degree I/Q imbalance.

```
ampImb = 3;
phImb = 10;
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(x)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(x)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
y = imbI + imbQ;
```

Plot the spectrum of the imbalanced signal.

```
spectrum = dsp.SpectrumAnalyzer('SampleRate',1000,'PowerUnits','dBW');

spectrum(y)
```

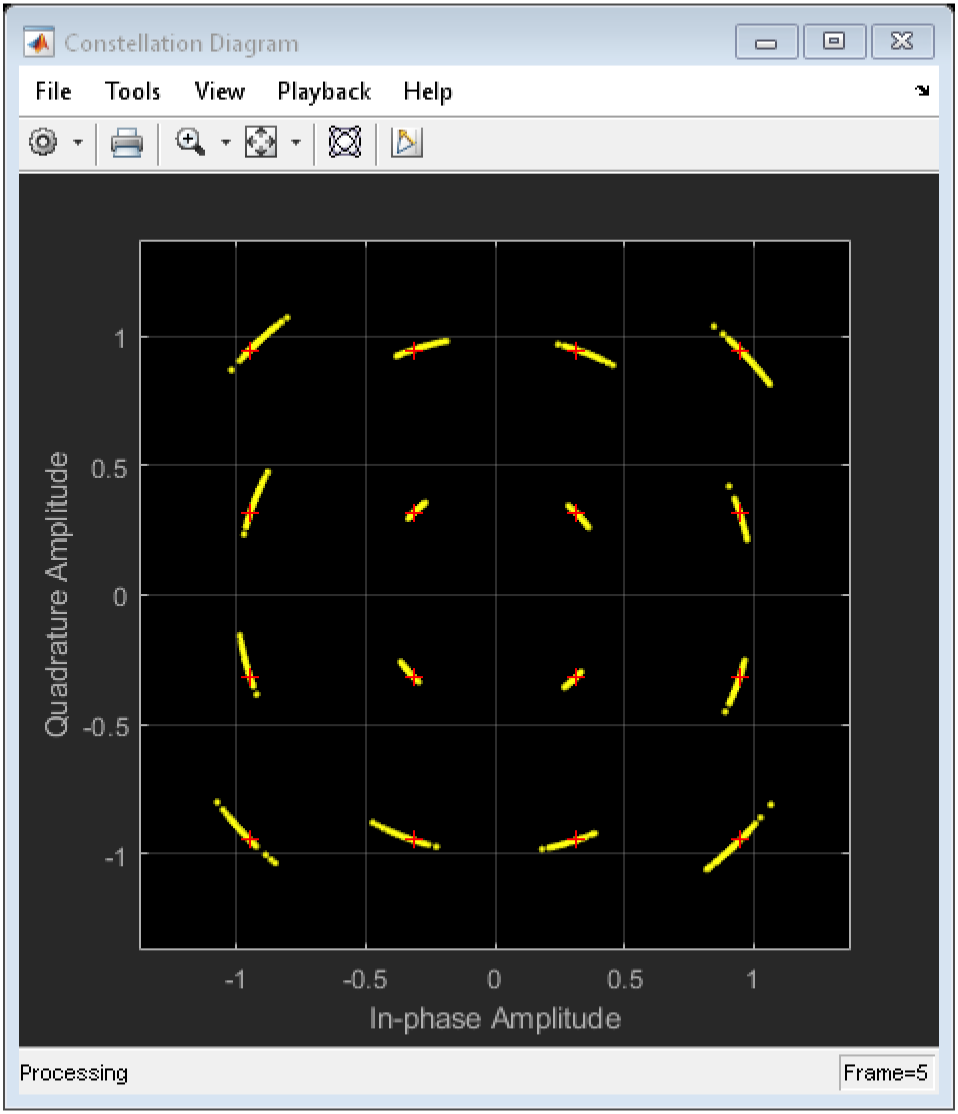


The I/Q imbalance introduces a second tone at -100 Hz, which is the inverse of the input tone.

Phase Noise

Apply phase noise to the transmitted signal. Plot the resulting constellation diagram.

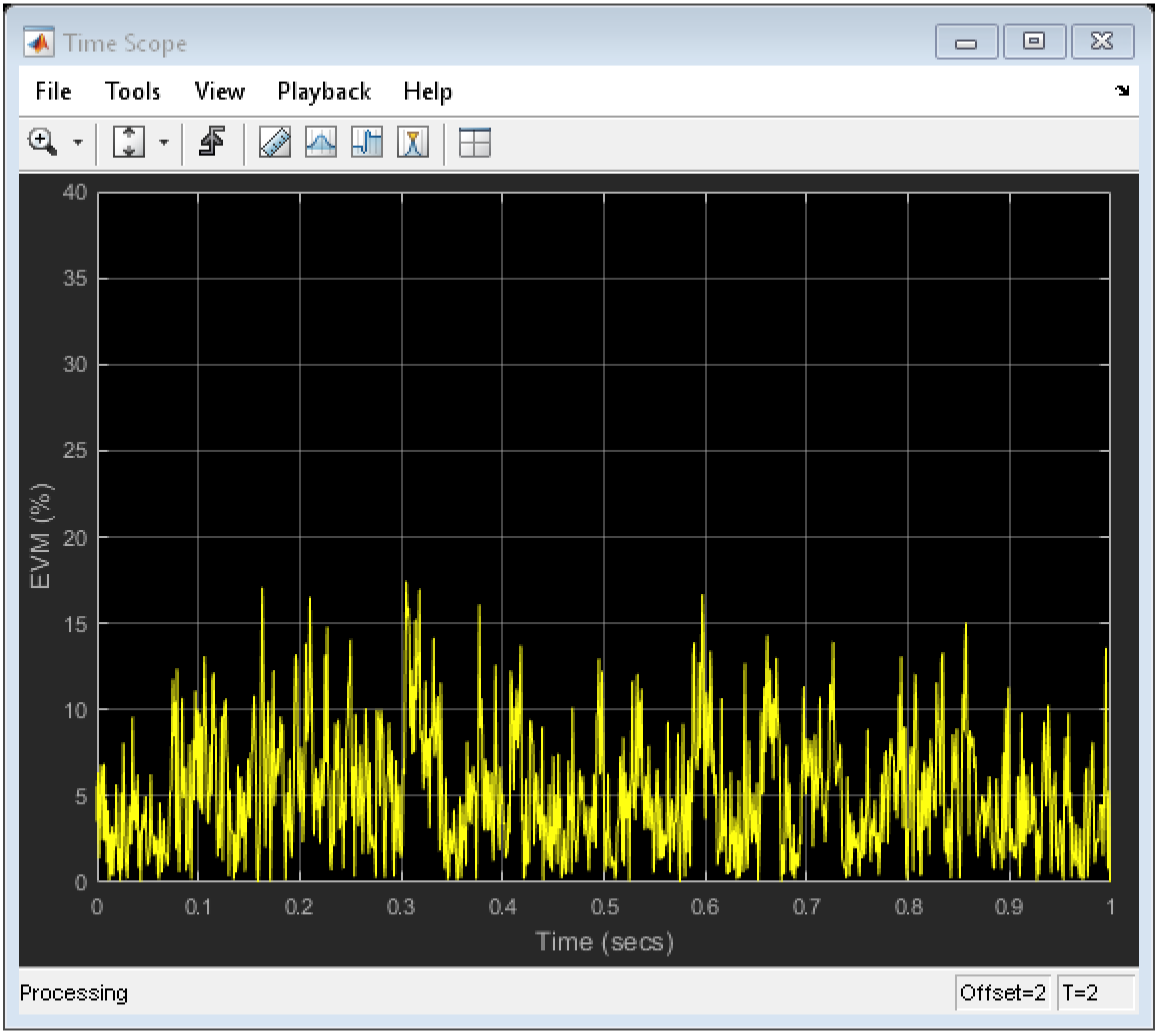
```
pnoise = comm.PhaseNoise('Level',-50,'FrequencyOffset',20,'SampleRate',fs);
pnoiseSig = pnoise(modSig);
constDiagram(pnoiseSig)
```



The phase noise introduces a rotational jitter.

Compute and plot the EVM of the received signal.

```
evmTime = evm(modSig,pnoiseSig);
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 6.1989
```

Filter Effects

Specify the samples per symbol parameter. Create a pair of raised cosine matched filters.

```
sps = 4;
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.2,'FilterSpanInSymbols',8
    'OutputSamplesPerSymbol',sps,'Gain',sqrt(sps));

rxfilter = comm.RaisedCosineReceiveFilter('RolloffFactor',0.2,'FilterSpanInSymbols',8,
    'InputSamplesPerSymbol',sps,'Gain',1/sqrt(sps), ...
    'DecimationFactor',sps);
```

Determine the delay through the matched filters.

```
fltDelay = 0.5*(txfilter.FilterSpanInSymbols + rxfilter.FilterSpanInSymbols);
```

Pass the modulated signal through the matched filters.

```
filtSig = txfilter(modSig);
rxSig = rxfilter(filtSig);
```

To account for the delay through the filters, discard the first `fltDelay` samples.

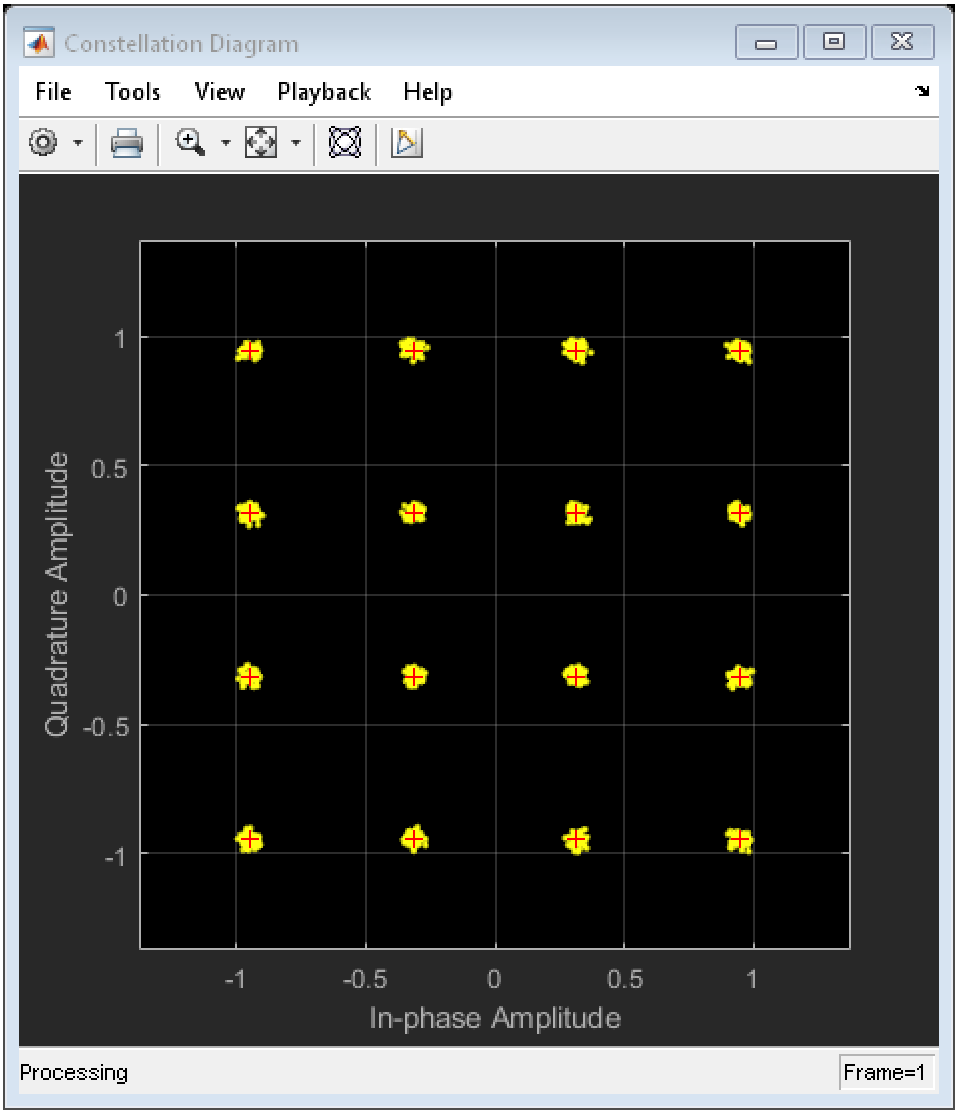
```
rxSig = rxSig(fltDelay+1:end);
```

To accommodate the change in the number of received signal samples, create new constellation diagram and time scope objects.

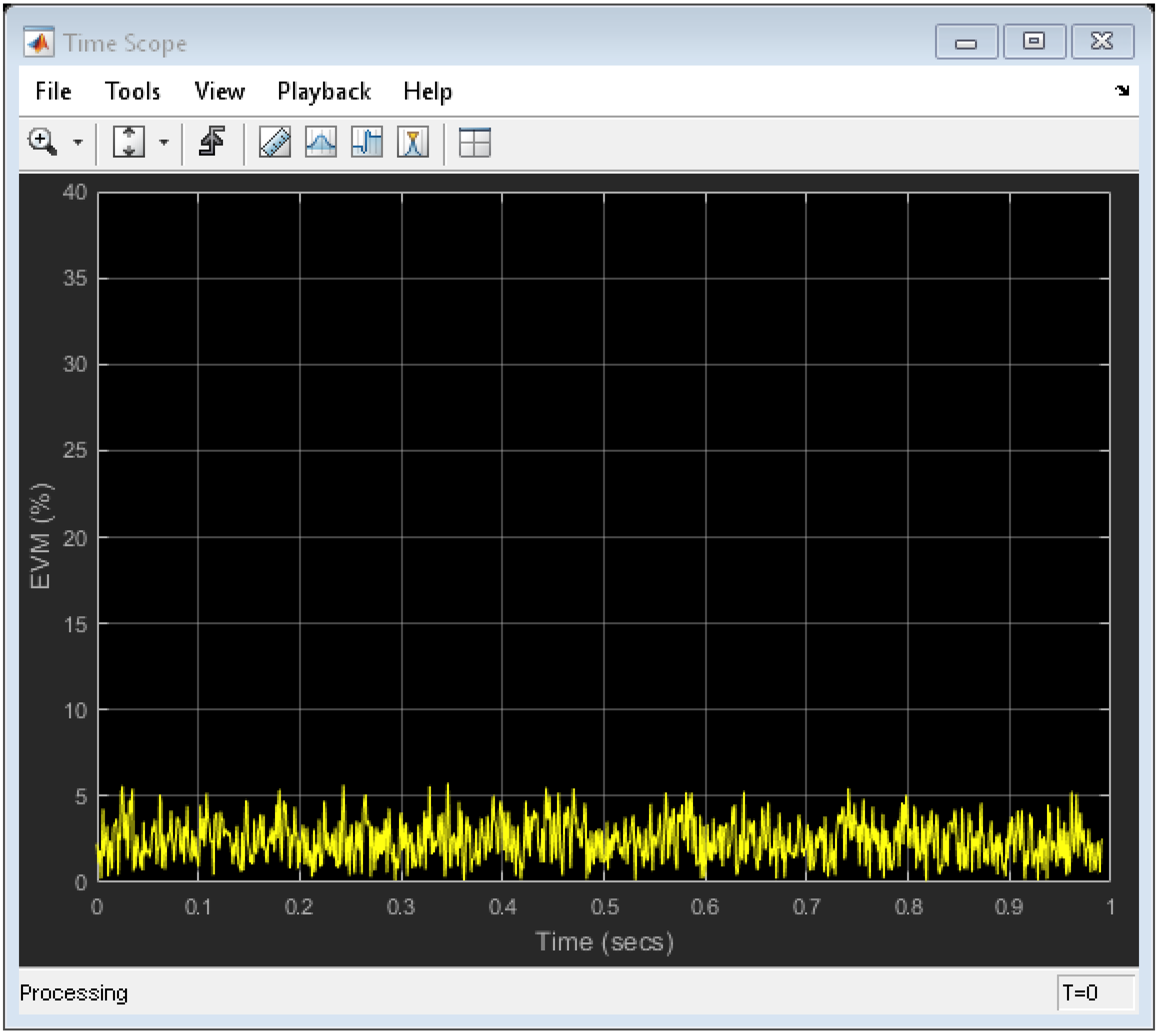
```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
```

Estimate EVM. Plot the received signal constellation diagram and the time-varying EVM.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
evmTime = evm(rxSig);
constDiagram(rxSig)
```



```
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 2.7199
```

Determine the equivalent SNR.

```
mer = comm.MER;
snrEst = mer(modSig(1:end-fltDelay),rxSig)
```

```
snrEst = 31.4603
```

Combined Effects

Combine the effects of the filters, nonlinear amplifier, AWGN, and phase noise. Display the constellation and EVM diagrams.

Create EVM, time scope and constellation diagram objects.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
```

Specify the nonlinear amplifier and phase noise objects.

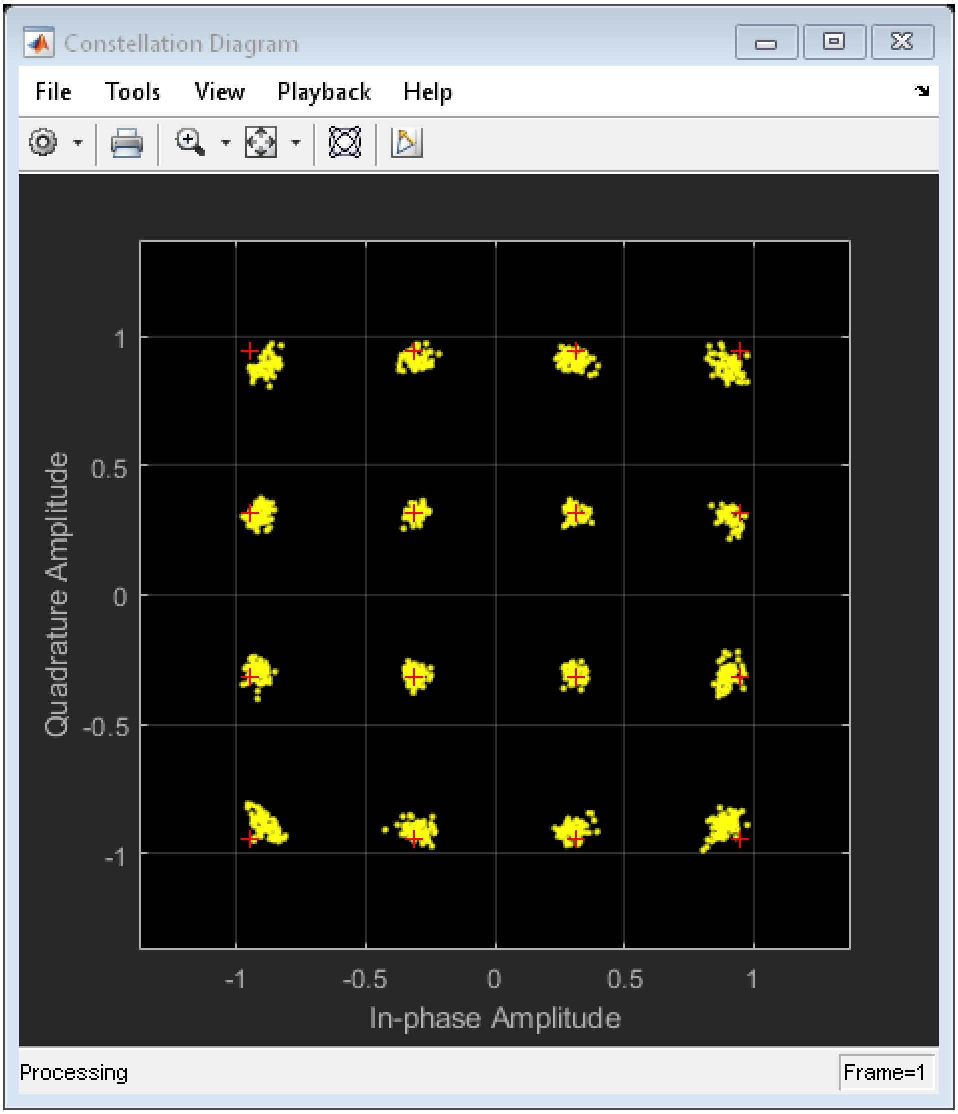
```
amp = comm.MemorylessNonlinearity('IIP3',45,'AMPMConversion',0);
pnoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',fs);
```

Filter and then amplify the modulated signal.

```
txfiltOut = txfilter(modSig);
txSig = amp(txfiltOut);
```

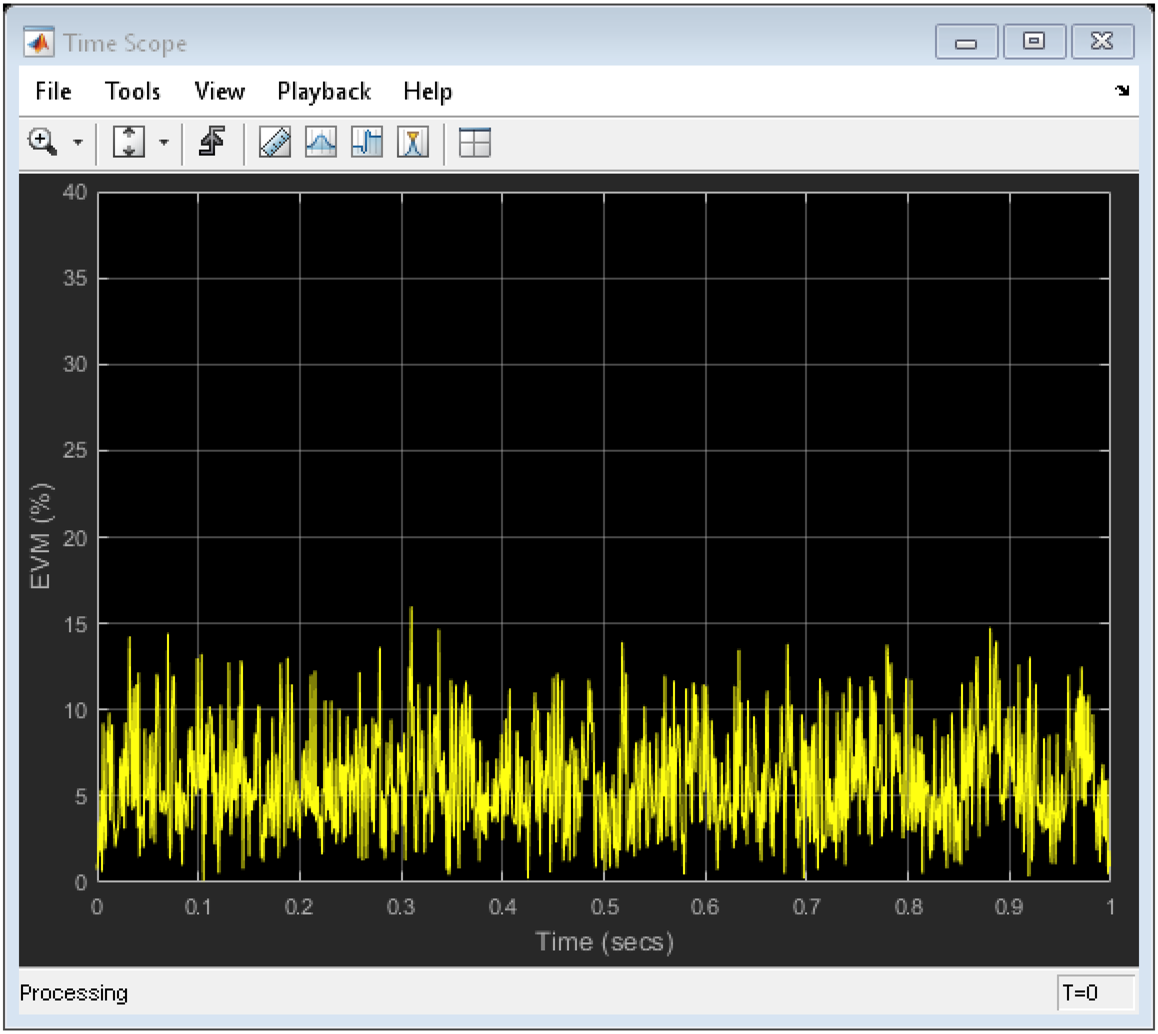
Add phase noise. Pass the impaired signal through the AWGN channel. Plot the constellation diagram.

```
rxSig = awgn(txSig,snrdB);
pnoiseSig = pnoise(rxSig);
rxfiltOut = rxfilter(pnoiseSig);
constDiagram(rxfiltOut)
```



Calculate the time-varying EVM. Plot the result.

```
evmTime = evm(rxfiltOut);
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 6.6444
```

Estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(rxfiltOut)
```

```
snrEst = 23.6978
```

This value is approximately 6 dB worse than the specified value of 30 dB, which means that the effects of the other impairments are significant and will degrade the bit error rate performance.

Definitions

Use the Channel Visualization Tool

Communications System Toolbox software provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See “Fading Channels” for a description of fading channels and objects.

For example, the following code opens the channel visualization tool showing a three-path Rayleigh channel through which a random signal is passed:

```
% Three-Path Rayleigh channel
h = rayleighchan(1/100000, 130, [0 1.5e-5 3.2e-5], [0, -3, -3]);
tx = randi([0 1],500,1);          % Random bit stream
hmod = comm.DBPSKModulator;       % Create DBPSKModulator
dpskSig = step(hmod,tx);          % DPSK signal
h.StoreHistory = true;            % Allow states to be stored
y = filter(h, dpskSig);           % Run signal through channel
plot(h);                          % Call Channel Visualization Tool
```

See Also

`comm.RayleighChannel` | `comm.RicianChannel`

Introduced before R2006a

pmdemod

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

Recover Phase Modulated Signal from AWGN Channel

Set the sample rate. To plot the signals, create a time vector.

```
fs = 50;
t = (0:2*fs+1)'/fs;
```

Create a sinusoidal input signal.

```
x = sin(2*pi*t) + sin(4*pi*t);
```

Set the carrier frequency and phase deviation.

```
fc = 10;
phasedev = pi/2;
```

Modulate the input signal.

```
tx = pmmod(x,fc,fs,phasedev);
```

Pass the signal through an AWGN channel.

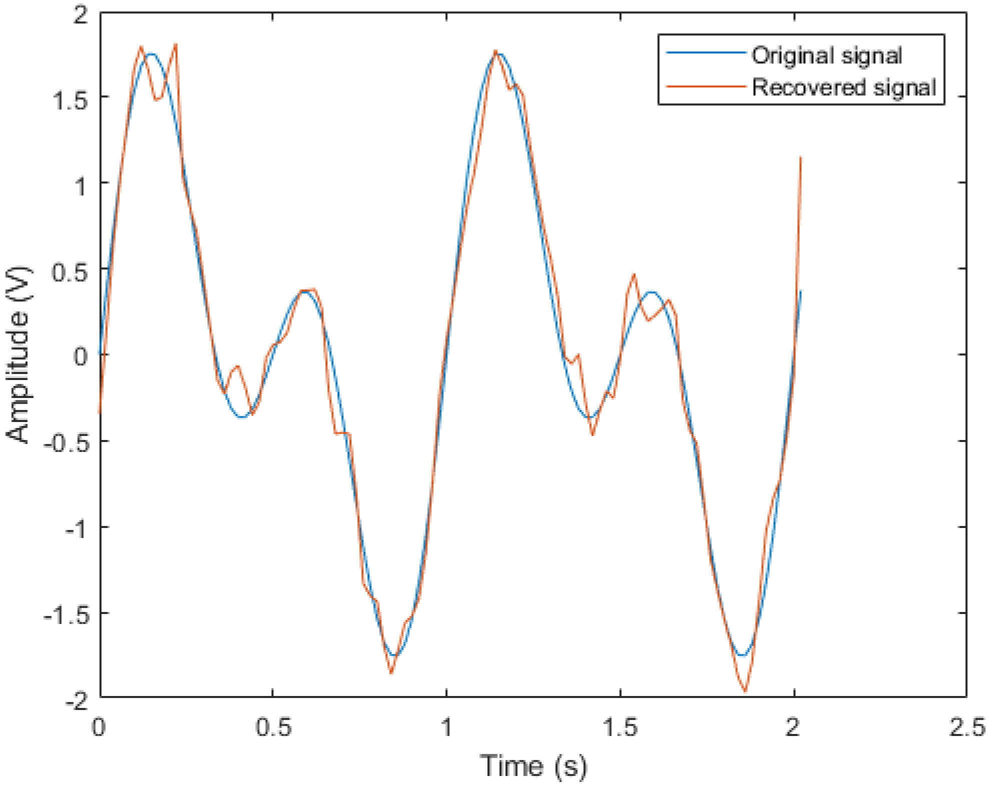
```
rx = awgn(tx,10,'measured');
```

Demodulate the noisy signal.

```
y = pmdemod(rx,fc,fs,phasedev);
```

Plot the original and recovered signals.

```
figure; plot(t,[x y]);
legend('Original signal','Recovered signal');
xlabel('Time (s)')
ylabel('Amplitude (V)')
```



See Also

`fmdemod` | `fmmod` | `pmmod`

Topics

“Digital Modulation”

Introduced before R2006a

pmmod

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

Recover Phase Modulated Signal from AWGN Channel

Set the sample rate. To plot the signals, create a time vector.

```
fs = 50;
t = (0:2*fs+1)'/fs;
```

Create a sinusoidal input signal.

```
x = sin(2*pi*t) + sin(4*pi*t);
```

Set the carrier frequency and phase deviation.

```
fc = 10;
phasedev = pi/2;
```

Modulate the input signal.

```
tx = pmmod(x,fc,fs,phasedev);
```

Pass the signal through an AWGN channel.

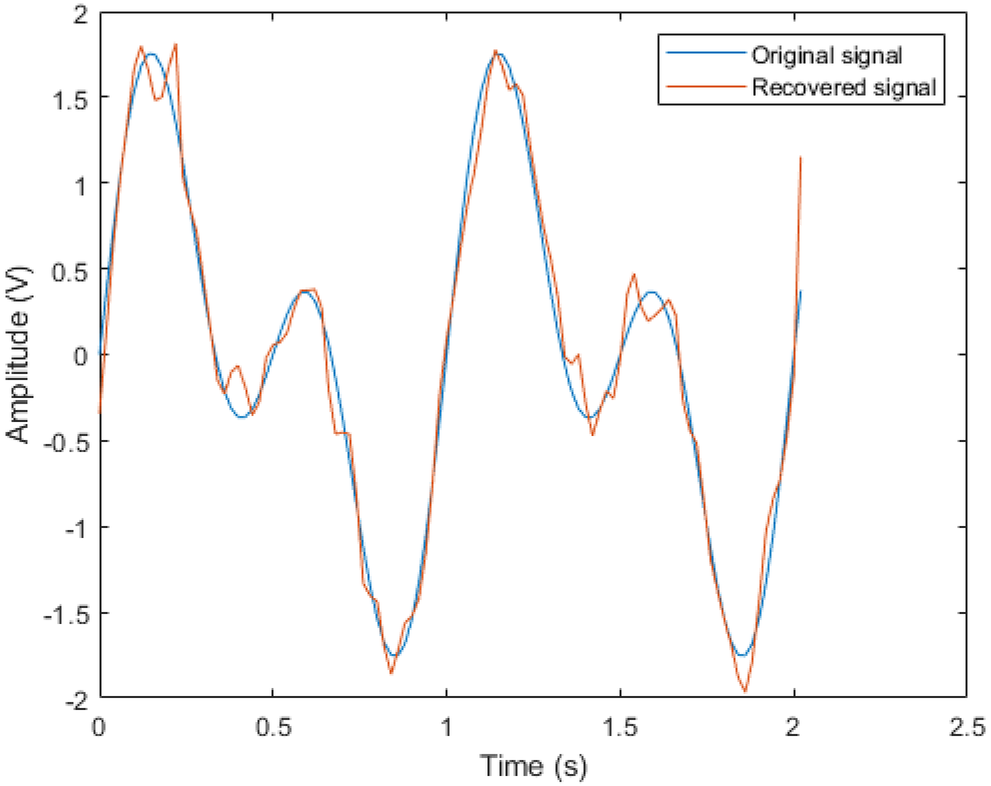
```
rx = awgn(tx,10,'measured');
```

Demodulate the noisy signal.

```
y = pmdemod(rx,fc,fs,phasedev);
```

Plot the original and recovered signals.

```
figure; plot(t,[x y]);
legend('Original signal','Recovered signal');
xlabel('Time (s)')
ylabel('Amplitude (V)')
```



See Also

`fmdemod` | `fmmod` | `pmdemod`

Topics

"Digital Modulation"

Introduced before R2006a

poly2trellis

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 or a *k*-by-*n* cell array of polynomial character vectors 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

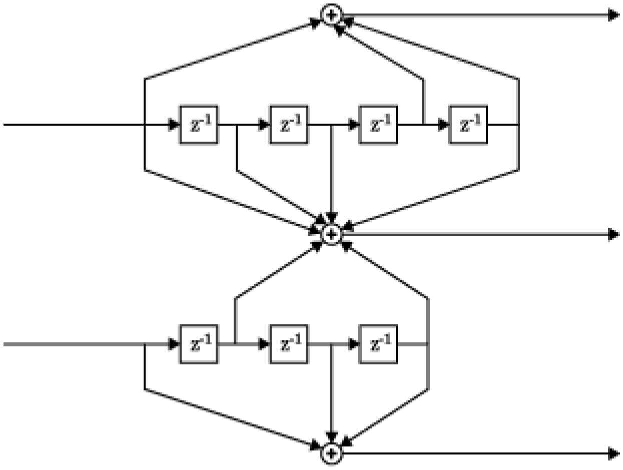
For more about this structure, see the reference page for the `istrellis` function.

Examples

Trellis Structure for a 2/3 Feedforward Convolutional Encoder

Create a trellis structure for a rate 2/3 feedforward convolutional code and display a portion of the next states of the trellis.

The structure of the encoder is depicted. As expected for rate 2/3 encoder, there are two input streams and three output streams.



Create a trellis, where the constraint length of the upper path is 5 and the constraint length of the lower path is 4. The octal representation of the code generator matrix corresponds to the taps from the upper and lower shift registers.

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])
```

```
trellis = struct with fields:
     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

The structure field `numInputSymbols` equals 4 because two bit streams can produce four different input symbols while `numOutputSymbols` equals 8 because three bit streams produce eight different output symbols. As there are seven total shift registers, there are $2^7 = 128$ possible states as shown by `trellis.nextStates`.

Display the first five rows of the 128-by-4 `trellis.nextStates` matrix.

```
trellis.nextStates(1:5,:)
```

```
ans = 5×4
```

```
     0    64     8    72
     0    64     8    72
     1    65     9    73
     1    65     9    73
     2    66    10    74
```

An example of where this encoder is used is found in `convenc`.

Trellis Structure for a 1/2 Feedforward Convolutional Encoder

Create a trellis structure for a rate 1/2 feedforward convolutional code and use it to encode and decode a random bit stream.

Create a trellis in which the constraint length is 7 and the code generator is specified as a cell array of polynomial character vectors.

```
trellis = poly2trellis(7,{'1 + x^3 + x^4 + x^5 + x^6', ...
    '1 + x + x^3 + x^4 + x^6'})
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 64
          nextStates: [64x2 double]
             outputs: [64x2 double]
```

Generate random binary data, convolutionally encode the data, and decode the data using the Viterbi algorithm.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
decodedData = vitdec(codedData,trellis,34,'trunc','hard');
```

Verify that there are no bit errors in the decoded data.

```
biterr(data,decodedData)
```

```
ans = 0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convenc` | `istrellis` | `vitdec`

Topics

"Convolutional Codes"

Introduced before R2006a

primpoly

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

gf | isprimitive

Topics

"Galois Field Computations"

Introduced before R2006a

pskdemod

Phase shift keying demodulation

Syntax

```
z = pskdemod(y,M)
z = pskdemod(y,M,ini_phase)
z = pskdemod(y,M,ini_phase,symorder)
```

Description

`z = pskdemod(y,M)` demodulates the complex envelope, `y`, of a PSK-modulated signal having modulation order `M`.

`z = pskdemod(y,M,ini_phase)` specifies the initial phase of the PSK-modulated signal.

`z = pskdemod(y,M,ini_phase,symorder)` specifies the symbol order of the PSK-modulated signal.

Examples

Compare Phase Noise Effects on PSK and PAM Signals

Compare PSK and PAM modulation schemes to demonstrate that PSK is more sensitive to phase noise. This is the expected result because the PSK constellation is circular while the PAM constellation is linear.

Specify the number of symbols and the modulation order parameters. Generate random data symbols.

```
len = 10000;
M = 16;
msg = randi([0 M-1],len,1);
```

Modulate `msg` using both PSK and PAM to compare the two methods.

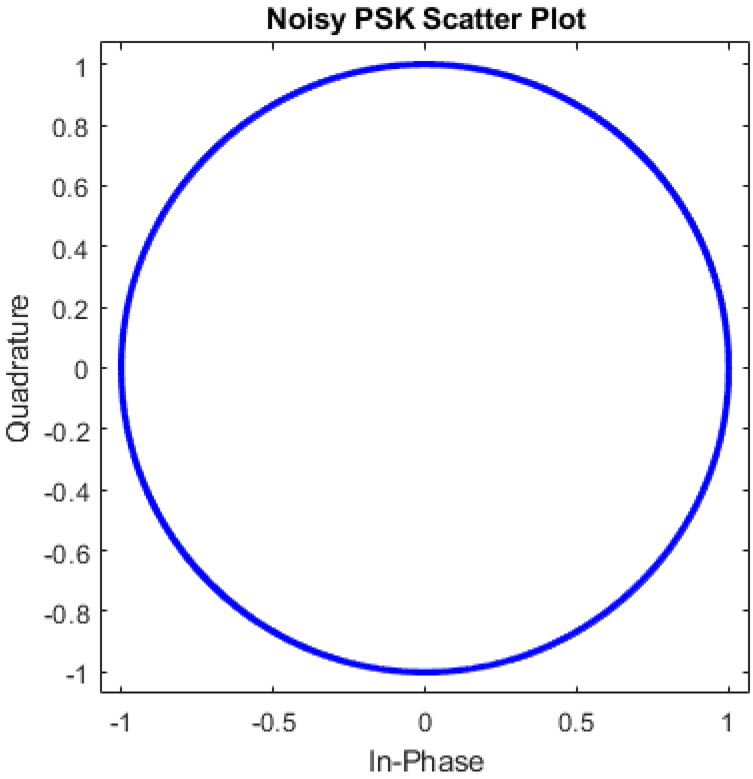
```
txpsk = pskmod(msg,M);
txpam = pammod(msg,M);
```

Perturb the phase of the modulated signals by applying a random phase rotation.

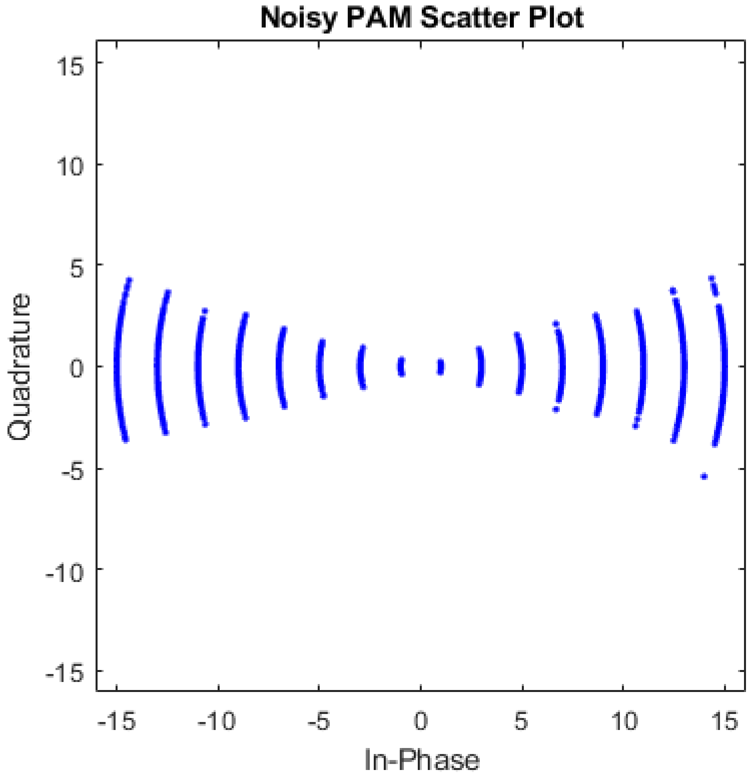
```
phasenoise = randn(len,1)*.015;
rxpsk = txpsk.*exp(2i*pi*phasenoise);
rxpam = txpam.*exp(2i*pi*phasenoise);
```

Create scatter plots 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 the number of symbol errors for each modulation scheme. The PSK signal experiences a much greater number of symbol errors.

```
numerrs_psk = symerr(msg,recovpsk);
numerrs_pam = symerr(msg,recovpam);
[numerrs_psk numerrs_pam]
```

```
ans = 1×2

   343     1
```

Modulate and Demodulate QPSK Signal in AWGN

Generate random symbols.

```
dataIn = randi([0 3],1000,1);
```

QPSK modulate the data.

```
txSig = pskmod(dataIn,4,pi/4);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,10);
```

Demodulate the received signal and compute the number of symbol errors.

```
dataOut = pskdemod(rxSig,4,pi/4);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 2
```

Input Arguments

y — PSK-modulated input signal

vector | matrix

PSK-modulated input signal, specified as a real or complex vector or matrix. If y is a matrix, the function processes the columns independently.

Data Types: single | double
Complex Number Support: Yes

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double | single

ini_phase — Initial phase

0 (default) | scalar | [ ]

Initial phase of the PSK modulation, specified in radians as a real scalar.

If ini_phase is empty, then pskdemod uses an initial phase of 0.

Example: pi/4

Data Types: double | single

symorder — Symbol order

'bin' (default) | 'gray'

Symbol order, specified as 'bin' or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

- If symorder is 'bin', the function uses a natural binary-coded ordering.
- If symorder is 'gray', the function uses a Gray-coded ordering.

Data Types: char

Output Arguments

z — PSK-demodulated output signal

vector | matrix

PSK-demodulated output signal, returned as a vector or matrix having the same number of columns as input signal y.

Data Types: double | single

See Also

comm.PSKDemodulator | modnorm | pskmod

Topics

"Phase Modulation"

Introduced before R2006a

pskmod

Phase shift keying modulation

Syntax

```
y = pskmod(x,M)
y = pskmod(x,M,ini_phase)
y = pskmod(x,M,ini_phase,symorder)
```

Description

`y = pskmod(x,M)` modulates the input signal, `x`, using phase shift keying (PSK) with modulation order `M`.

`y = pskmod(x,M,ini_phase)` specifies the initial phase of the PSK-modulated signal.

`y = pskmod(x,M,ini_phase,symorder)` specifies the symbol order of the PSK-modulated signal.

Examples

Modulate PSK Signal

Modulate and plot the constellations of QPSK and 16-PSK signals.

QPSK

Set the modulation order to 4.

```
M = 4;
```

Generate random data symbols.

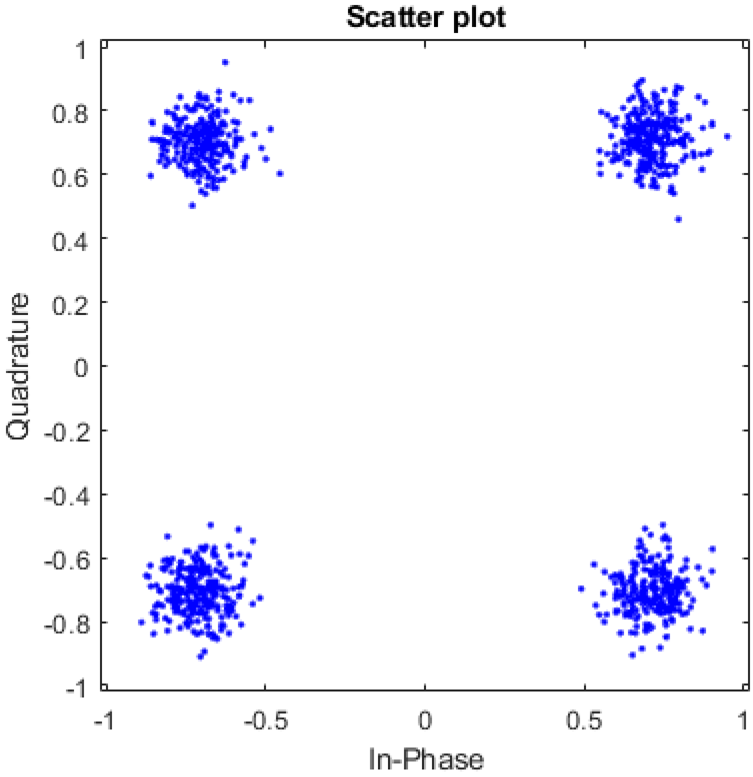
```
data = randi([0 M-1],1000,1);
```

Modulate the data symbols.

```
txSig = pskmod(data,M,pi/M);
```

Pass the signal through white noise and plot its constellation.

```
rxSig = awgn(txSig,20);
scatterplot(rxSig)
```



16-PSK

Change the modulation order from 4 to 16.

```
M = 16;
```

Generate random data symbols.

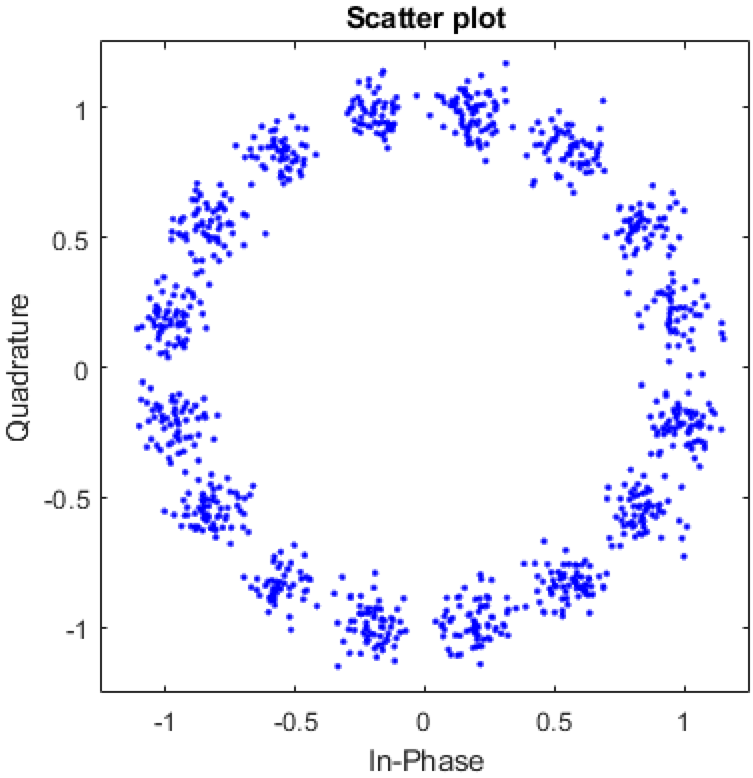
```
data = randi([0 M-1],1000,1);
```

Modulate the data symbols.

```
txSig = pskmod(data,M,pi/M);
```

Pass the signal through white noise and plot its constellation.

```
rxSig = awgn(txSig,20);
scatterplot(rxSig)
```



Modulate and Demodulate QPSK Signal in AWGN

Generate random symbols.

```
dataIn = randi([0 3],1000,1);
```

QPSK modulate the data.

```
txSig = pskmod(dataIn,4,pi/4);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,10);
```

Demodulate the received signal and compute the number of symbol errors.

```
dataOut = pskdemod(rxSig,4,pi/4);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 2
```

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of x must have values in the range of [0, M – 1].

Example: randi([0 3],100,1)

Data Types: double | single

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double | single

ini_phase — Initial phase

0 (default) | scalar | [ ]

Initial phase of the PSK modulation, specified in radians as a real scalar.

If you specify `ini_phase` as empty, then `pskmod` uses an initial phase of 0.

Example: `pi/4`

Data Types: `double` | `single`

`symorder` — Symbol order
`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

Output Arguments

y — PSK-modulated output signal
vector | matrix

Complex baseband representation of a PSK-modulated signal, returned as vector or matrix. The columns of `y` represent independent channels.

Data Types: `double` | `single`

See Also

`comm.PSKModulator` | `modnorm` | `pskdemod`

Topics

"Phase Modulation"

Introduced before R2006a

qamdemod

Quadrature amplitude demodulation

Syntax

```
z = qamdemod(y,M)
z = qamdemod(y,M,symOrder)
z = qamdemod( ___ ,Name,Value)

z = qamdemod(y,M,iniPhase)
```

Description

`z = qamdemod(y,M)` returns a demodulated signal, `z`, given quadrature amplitude modulated (QAM) signal `y` of modulation order `M`.

`z = qamdemod(y,M,symOrder)` returns a demodulated signal and specifies the symbol order.

`z = qamdemod( ___ ,Name,Value)` specifies demodulation behavior using `Name,Value` pairs and any of the previous syntaxes.

`z = qamdemod(y,M,iniPhase)` specifies the initial phase of the QAM constellation. `qamdemod` will not accept `iniPhase` in a future release. Use `z = qamdemod(y,M)` instead.

Input Arguments

y — Input signal
scalar | vector | matrix | 3-D array

Input signal resulting from quadrature amplitude modulation, specified as a complex scalar, vector, matrix, or 3-D array. Each column is treated as an independent channel.

Data Types: `double` | `single` | `fi`

M — Modulation order

scalar integer

Modulation order, specified as a power-of-two scalar integer. The modulation order specifies the number of points in the signal constellation.

Example: 16

Data Types: double

symOrder — Symbol order

'gray' (default) | 'bin' | vector

Symbol order, specified as 'gray', 'bin', or a vector.

- 'gray' — Use “Gray Code” on page 1-849 ordering
- 'bin' — Use natural binary-coded ordering
- Vector — Use custom symbol ordering

Vectors must use unique elements whose values range from 0 to M – 1. The first element corresponds to the upper-left point of the constellation, with subsequent elements running down column-wise from left to right.

Example: [0 3 1 2]

Data Types: char | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

UnitAveragePower — Unit average power flag

false (default) | true

Unit average power flag, specified as the comma-separated pair consisting of UnitAveragePower and a logical scalar. When this flag is true, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is false, the functon scales the constellation so that the QAM constellation points are separated by a minimum distance of 2.

Data Types: `logical`

`OutputType` — Output type

`'integer'` (default) | `'bit'` | `'llr'` | `'approxllr'`

Output type, specified as the comma-separated pair consisting of `OutputType` and one of the following: `'integer'`, `'bit'`, `'llr'`, or `'approxllr'`.

Data Types: `char`

`NoiseVariance` — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of elements in the last dimension of the input signal. Each element of the vector specifies noise variance for all the elements of the input along the corresponding last dimension.

Tip When `OutputType` = `‘llr’`, if the demodulation computation outputs `Inf` or `-Inf` values, it is likely because the specified noise variance values are significantly smaller than the SNR. Since the LLR algorithm computes exponentials using finite precision arithmetic, computation of exponentials with very large or very small numbers can yield positive or negative infinity.

Try using `OutputType` = `‘approxllr’` instead, because the approximate LLR algorithm does not compute exponentials.

Dependencies

This input argument applies only when `'OutputType'` is set to `'llr'` or `'approxllr'`.

Data Types: `double`

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the QAM constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

z — Demodulated output signal

scalar | vector | matrix | 3-D array

Demodulated output signal, returned as a scalar, vector, matrix, or 3-D array. The data type is the same as that of the input signal, `y`. The dimensions of the output vary depending on the specified `OutputType` value.

'OutputType'	Return Value of qamdemod	Dimensions of Output
'integer'	Demodulated integer values from 0 to `M` – 1	`z` has the same dimensions as input `y`.
'bit'	Demodulated bits	The number of rows in `z` is log$_2$(`M`) times the number of rows in `y`. Each demodulated symbol is mapped to a group of log$_2$(`M`) bits, where the first bit represents the MSB and the last bit represents the LSB.
'llr'	Log-likelihood ratio value for each bit	
'approxllr'	Approximate log-likelihood ratio value for each bit	

Examples

Demodulate 8-QAM Signal

Demodulate an 8-QAM signal and plot the points corresponding to symbols 0 and 3.

Generate random 8-ary data symbols.

```
data = randi([0 7],1000,1);
```

Apply 8-QAM.

```
txSig = qammod(data,8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,18,'measured');
```

Demodulate the received signal using an initial phase of π/8.

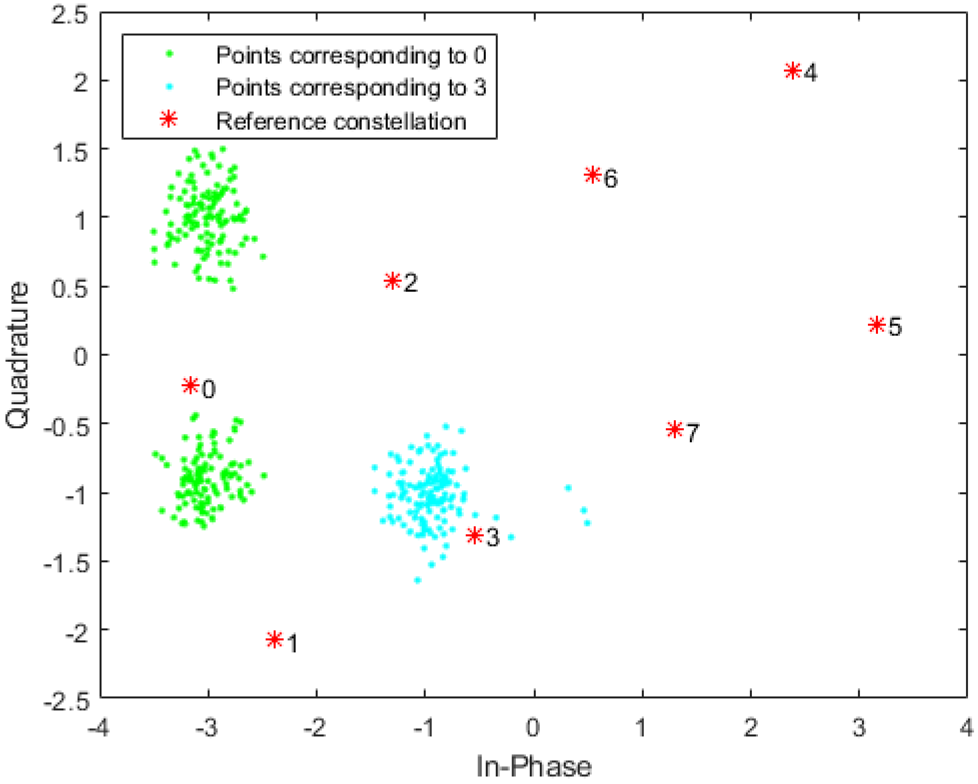
```
rxData = qamdemod(rxSig.*exp(-1i*pi/8),8);
```

Generate the reference constellation points.

```
refpts = qammod((0:7)',8) .* exp(1i*pi/8);
```

Plot the received signal points corresponding to symbols 0 and 3 and overlay the reference constellation. Only the received data corresponding to those symbols is displayed.

```
plot(rxSig(rxData==0),'g.');
hold on
plot(rxSig(rxData==3),'c.');
plot(refpts,'r*')
text(real(refpts)+0.1,imag(refpts),num2str((0:7)'))
xlabel('In-Phase')
ylabel('Quadrature')
legend('Points corresponding to 0','Points corresponding to 3', ...
    'Reference constellation','location','nw');
```



QAM Demodulation with WLAN Symbol Mapping

Modulate and demodulate random data by using 16-QAM with WLAN symbol mapping. Verify that the input data symbols match the demodulated symbols.

Generate a 3-D array of random symbols.

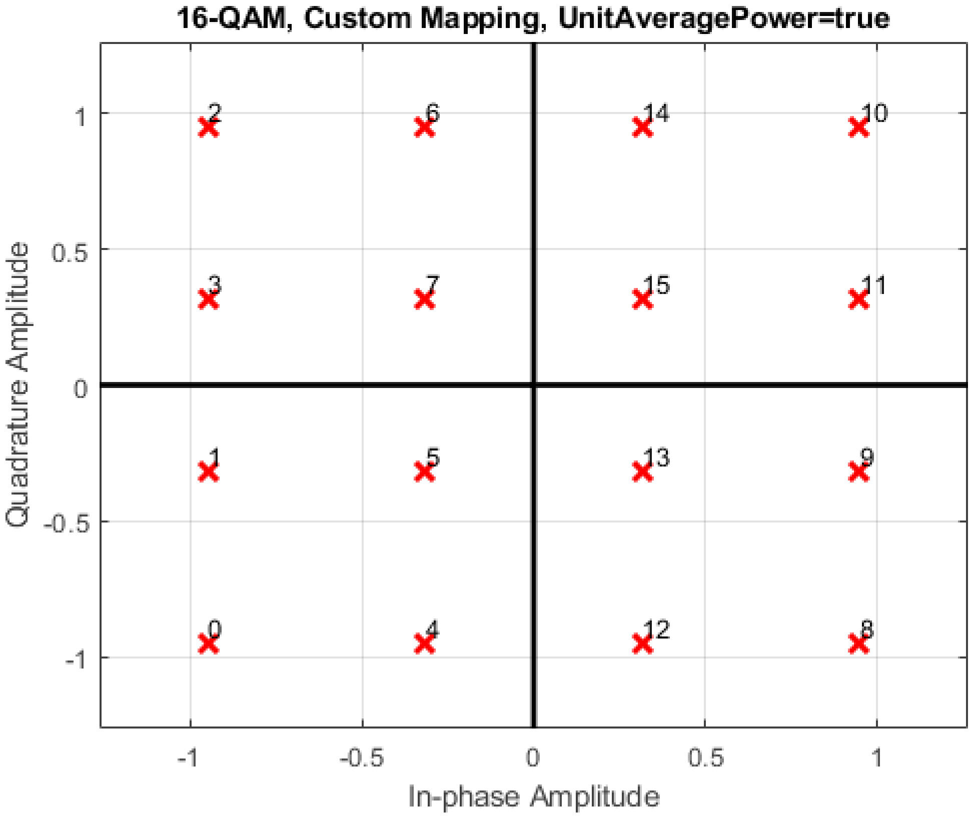
```
x = randi([0,15],20,4,2);
```

Create a custom symbol mapping for the 16-QAM constellation based on WLAN standards.

```
wlanSymMap = [2 3 1 0 6 7 5 4 14 15 13 12 10 11 9 8];
```

Modulate the data, and set the constellation to have unit average signal power. Plot the constellation.

```
y = qammod(x,16,wlanSymMap,'UnitAveragePower', true,'PlotConstellation',true);
```



Demodulate the received signal.

```
z = qamdemod(y,16,wlanSymMap,'UnitAveragePower',true);
```

Verify that the demodulated signal is equal to the original data.

```
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate QAM Fixed-Point Signal

Demodulate a fixed-point QAM signal and verify that the data is recovered correctly.

Set the modulation order, and determine the number of bits per symbol.

```
M = 64;
bitsPerSym = log2(M);
```

Generate random bits. When operating in bit mode, the length of the input data must be an integer multiple of the number of bits per symbol.

```
x = randi([0 1],10*bitsPerSym,1);
```

Modulate the input data using a binary symbol mapping. Set the modulator to output fixed-point data. The numeric data type is signed with a 16-bit word length and a 10-bit fraction length.

```
y = qammod(x,M,'bin','InputType','bit','OutputDataType', ...
    numerictype(1,16,10));
```

Demodulate the 64-QAM signal. Verify that the demodulated data matches the input data.

```
z = qamdemod(y,M,'bin','OutputType','bit');
s = isequal(x,double(z))
```

```
s = logical
   1
```

Estimate BER for Hard and Soft Decision Viterbi Decoding

Estimate bit error rate (BER) performance for hard-decision and soft-decision Viterbi decoders in AWGN. Compare the performance to that of an uncoded 64-QAM link.

Set the simulation parameters.

```
clear; close all
rng default
M = 64;                 % Modulation order
k = log2(M);            % Bits per symbol
EbNoVec = (4:10)';       % Eb/No values (dB)
numSymPerFrame = 1000;   % Number of QAM symbols per frame
```

Initialize the BER results vectors.

```
berEstSoft = zeros(size(EbNoVec));
berEstHard = zeros(size(EbNoVec));
```

Set the trellis structure and traceback length for a rate 1/2, constraint length 7, convolutional code.

```
trellis = poly2trellis(7,[171 133]);
tbl = 32;
rate = 1/2;
```

The main processing loops performs these steps:

- Generate binary data.
- Convolutionally encode the data.
- Apply QAM modulation to the data symbols. Specify unit average power for the transmitted signal.
- Pass the modulated signal through an AWGN channel.
- Demodulate the received signal using hard decision and approximate LLR methods. Specify unit average power for the received signal.
- Viterbi decode the signals using hard and unquantized methods.
- Calculate the number of bit errors.

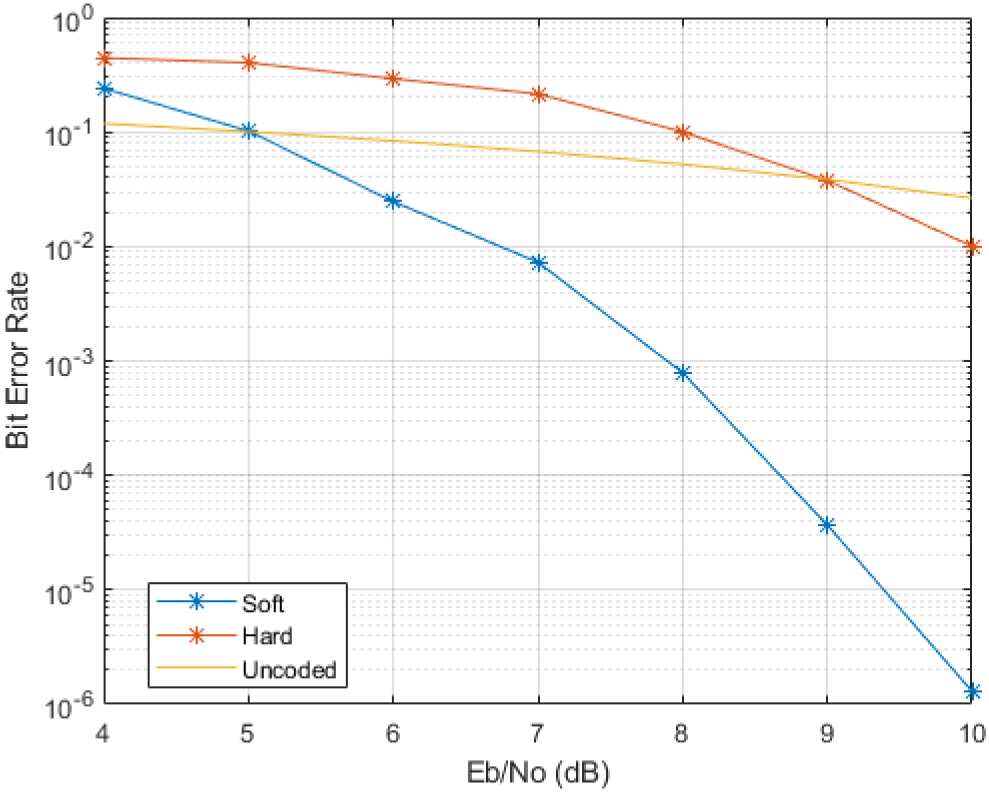
The `while` loop continues to process data until either 100 errors are encountered or 1e7 bits are transmitted.

```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k*rate);
    % Noise variance calculation for unity average signal power.
    noiseVar = 10.^(-snrdB/10);
    % Reset the error and bit counters
    [numErrsSoft,numErrsHard,numBits] = deal(0);
    while numErrsSoft < 100 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame*k,1);

        % Convolutionally encode the data
        dataEnc = convenc(dataIn,trellis);

        % QAM modulate
        txSig = qammod(dataEnc,M,'InputType','bit','UnitAveragePower',true);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal using hard decision (bit) and
        % soft decision (approximate LLR) approaches.
        rxDataHard = qamdemod(rxSig,M,'OutputType','bit','UnitAveragePower',true);
        rxDataSoft = qamdemod(rxSig,M,'OutputType','approxllr', ...
            'UnitAveragePower',true,'NoiseVariance',noiseVar);

        % Viterbi decode the demodulated data
        dataHard = vitdec(rxDataHard,trellis,tbl,'cont','hard');
        dataSoft = vitdec(rxDataSoft,trellis,tbl,'cont','unquant');

        % Calculate the number of bit errors in the frame. Adjust for the
        % decoding delay, which is equal to the traceback depth.
        numErrsInFrameHard = biterr(dataIn(1:end-tbl),dataHard(tbl+1:end));
        numErrsInFrameSoft = biterr(dataIn(1:end-tbl),dataSoft(tbl+1:end));

        % Increment the error and bit counters
        numErrsHard = numErrsHard + numErrsInFrameHard;
        numErrsSoft = numErrsSoft + numErrsInFrameSoft;
        numBits = numBits + numSymPerFrame*k;

    end

    % Estimate the BER for both methods
    berEstSoft(n) = numErrsSoft/numBits;
    berEstHard(n) = numErrsHard/numBits;
end
```

Plot the estimated hard and soft BER data. Plot the theoretical performance for an uncoded 64-QAM channel.

```
semilogy(EbNoVec,[berEstSoft berEstHard],'-*')
hold on
```

```
semilogy(EbNoVec,berawgn(EbNoVec,'qam',M))
legend('Soft','Hard','Uncoded','location','best')
grid
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



As expected, the soft decision decoding produces the best results.

Definitions

Gray Code

A Gray code, also known as a reflected binary code, is a system where the bit patterns in adjacent constellation points differ by only one bit.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`genqamdemod` | `genqammod` | `modnorm` | `pamdemod` | `qammod`

Topics

“Digital Modulation”
“Compute the Symbol Error Rate”
“Exact LLR Algorithm”

Introduced before R2006a

qammod

Quadrature amplitude modulation

Syntax

```
y = qammod(x,M)
y = qammod(x,M,symOrder)
y = qammod( ___ ,Name,Value)

y = qammod(x,M,iniPhase)
```

Description

`y = qammod(x,M)` returns a baseband quadrature amplitude modulated (QAM) signal given input signal `x` and modulation order `M`.

`y = qammod(x,M,symOrder)` returns a modulated signal and specifies the symbol order.

`y = qammod( ___ ,Name,Value)` specifies modulation behavior using `Name,Value` pairs and any of the previous syntaxes.

`y = qammod(x,M,iniPhase)` specifies the initial phase of the QAM constellation. `qammod` will not accept `iniPhase` in a future release. Use `y = qammod(x,M)` instead.

Input Arguments

x — Input signal

scalar | vector | matrix | 3-D array

Input signal, specified as a scalar, vector, matrix, or 3-D array. The elements of x must be binary values or integers that range from 0 to (`M` – 1), where `M` is the modulation order.

Note To process input signal as binary elements, set the `'InputType'` name-value pair to `'bit'`. For binary inputs, the number of rows must be an integer multiple of $\log_2$(`M`).

Groups of $\log_2$(M) bits are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: double | single | fi | int8 | int16 | uint8 | uint16

M — Modulation order
scalar integer

Modulation order, specified as a power-of-two scalar integer. The modulation order specifies the number of points in the signal constellation.

Example: 16

Data Types: double

symOrder — Symbol order
'gray' (default) | 'bin' | vector

Symbol order, specified as 'gray', 'bin', or a vector.

- 'gray' — Use “Gray Code” on page 1-862 ordering
- 'bin' — Use natural binary-coded ordering
- Vector — Use custom symbol ordering

Vectors must use unique elements whose values range from 0 to M – 1. The first element corresponds to the upper-left point of the constellation, with subsequent elements running down column-wise from left to right.

Example: [0 3 1 2]

Data Types: char | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

InputType — Input type
'integer' (default) | 'bit'

Input type, specified as the comma-separated pair consisting of `'InputType'` and either `'integer'` or `'bit'`. If you specify `'integer'`, the input signal must consist of integers from 0 to `M` – 1. If you specify `'bit'`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2$(`M`).

Data Types: `char`

`UnitAveragePower` — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `UnitAveragePower` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the functon scales the constellation so that the QAM constellation points are separated by a minimum distance of 2.

Data Types: `logical`

`OutputDataType` — Output data type

`numerictype` object

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and a numeric type object. See `numerictype` for more information on constructing these objects. If `OutputDataType` is omitted, the output data type is `double` for `double` or built-in integer inputs, and `single` for `single` inputs.

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the QAM constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

y — Modulated signal

scalar | vector | matrix | 3-D array

Modulated signal, returned as a complex scalar, vector, matrix, or 3-D array. For integer inputs, output y has the same dimensions as input signal x. For bit inputs, the number of rows in y is the number of rows in x divided by $\log_2$(M).

Data Types: double | single

Examples

Modulate Data Using QAM

Modulate data using QAM and display the result in a scatter plot.

Set the modulation order to 16 and create a data vector containing each of the possible symbols.

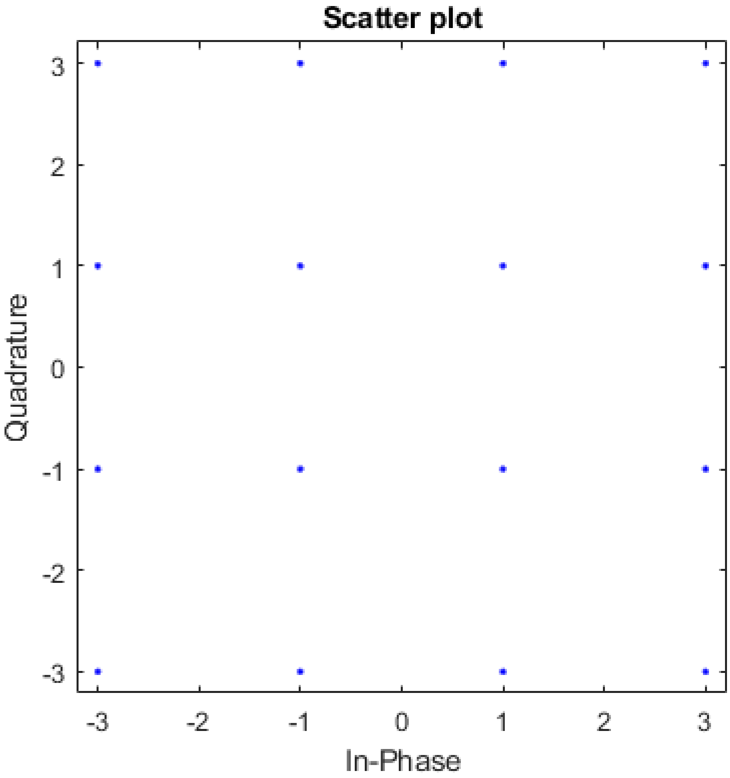
```
M = 16;
x = (0:M-1)';
```

Modulate the data using the qammod function.

```
y = qammod(x,M);
```

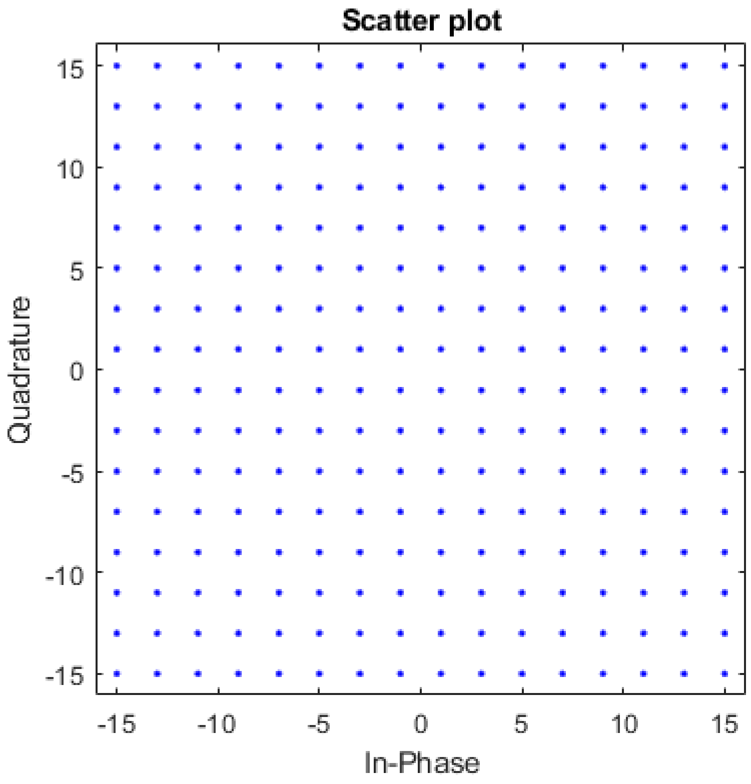
Display the modulated signal constellation using the scatterplot function.

```
scatterplot(y)
```



Set the modulation order to 256, and display the scatter plot of the modulated signal.

```
M = 256;
x = (0:M-1)';
y = qammod(x,M);
scatterplot(y)
```



Normalize QAM Signal by Average Power

Modulate random data symbols using QAM. Normalize the modulator output so that it has an average signal power of 1 W.

Set the modulation order and generate random data.

```
M = 64;
x = randi([0 M-1],1000,1);
```

Modulate the data. Use the `'UnitAveragePower'` name-value pair to set the output signal to have an average power of 1 W.

```
y = qammod(x,M,'UnitAveragePower',true);
```

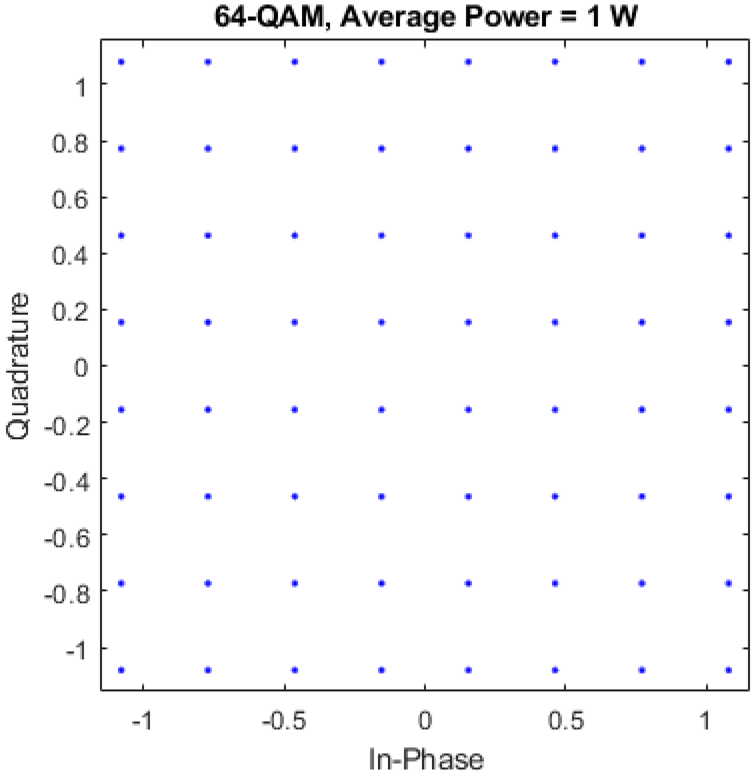
Confirm that the signal has unit average power.

```
avgPower = mean(abs(y).^2)
```

```
avgPower = 1.0070
```

Plot the resulting constellation.

```
scatterplot(y)
title('64-QAM, Average Power = 1 W')
```



QAM Symbol Ordering

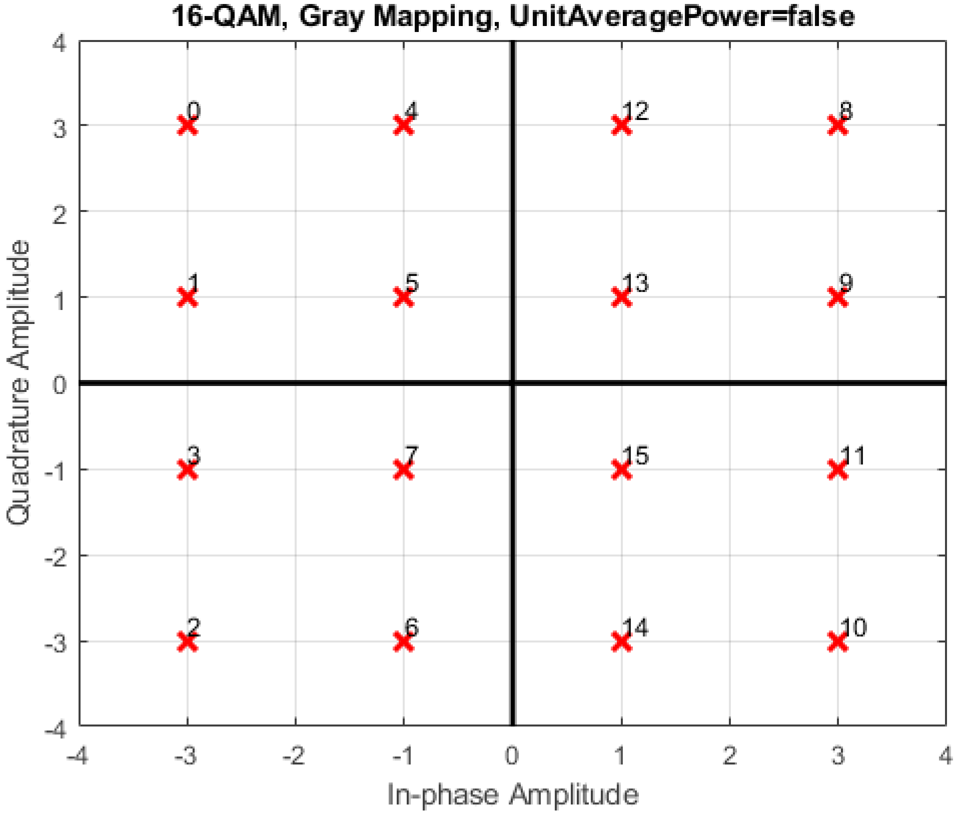
Plot QAM constellations for Gray, binary, and custom symbol mappings.

Set the modulation order, and create a random data sequence.

```
M = 16;
d = randi([0 M-1],1000,1);
```

Modulate the data, and plot its constellation.

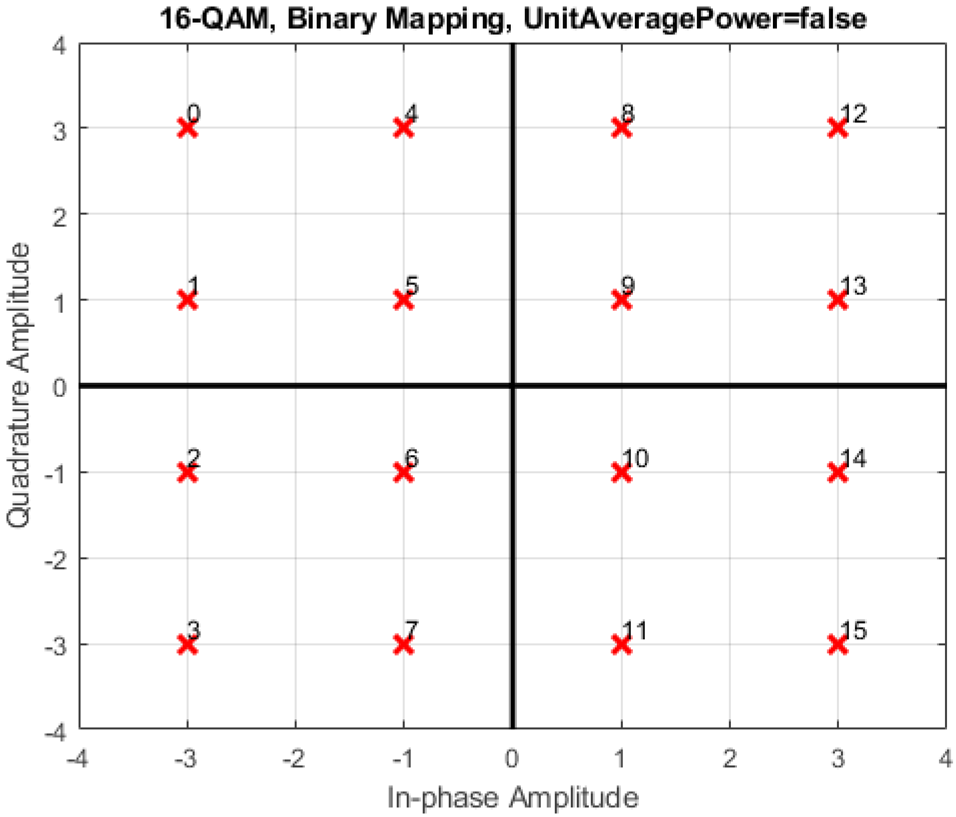
```
y = qammod(d,M,'PlotConstellation',true);
```



The default symbol mapping uses Gray ordering. The ordering of the points is not sequential.

Repeat the modulation process with binary symbol mapping.

```
z = qammod(d,M,'bin','PlotConstellation',true);
```



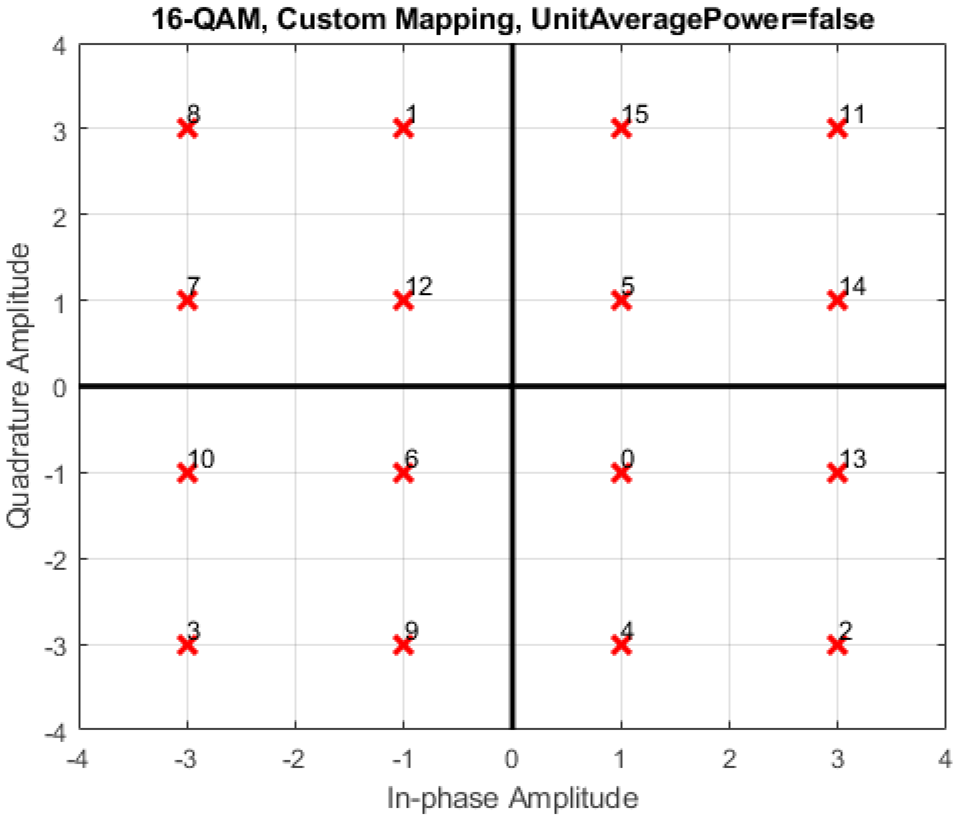
The symbol mapping follows a natural binary order and is sequential.

Create a custom symbol mapping.

```
smap = randperm(16)-1;
```

Modulate and plot the constellation.

```
w = qammod(d,M,smap,'PlotConstellation',true);
```



Quadrature Amplitude Modulation with Bit Inputs

Modulate a sequence of bits using 64-QAM. Pass the signal through a noisy channel. Display the resultant constellation diagram.

Set the modulation order, and determine the number of bits per symbol.

```
M = 64;
k = log2(M);
```

Create a binary data sequence. When using binary inputs, the number of rows in the input must be an integer multiple of the number of bits per symbol.

```
data = randi([0 1],1000*k,1);
```

Modulate the signal using bit inputs, and set it to have unit average power.

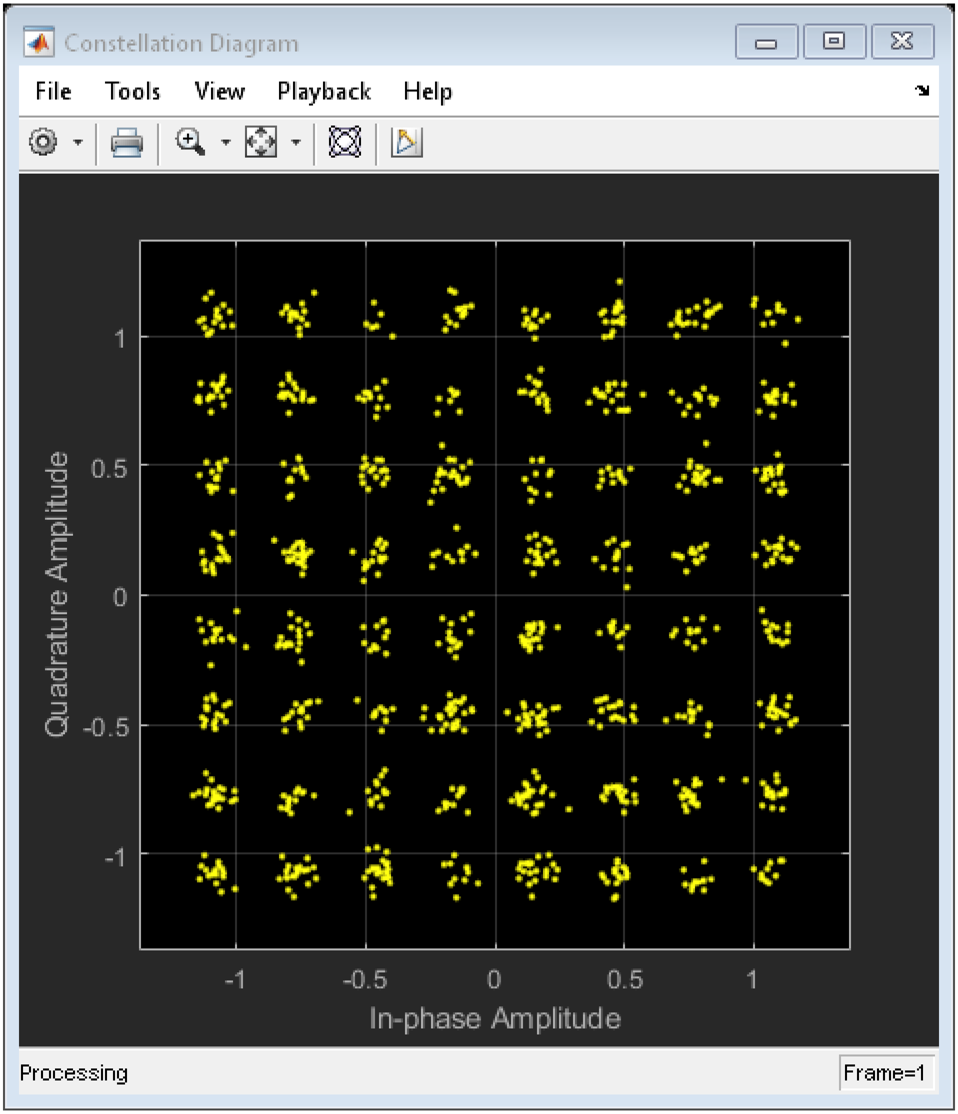
```
txSig = qammod(data,M,'InputType','bit','UnitAveragePower',true);
```

Pass the signal through a noisy channel.

```
rxSig = awgn(txSig,25);
```

Plot the constellation diagram.

```
cd = comm.ConstellationDiagram('ShowReferenceConstellation',false);
step(cd,rxSig)
```


Constellation Diagram
File
Tools
View
Playback
Help
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=1

Demodulate QAM Fixed-Point Signal

Demodulate a fixed-point QAM signal and verify that the data is recovered correctly.

Set the modulation order, and determine the number of bits per symbol.

```
M = 64;
bitsPerSym = log2(M);
```

Generate random bits. When operating in bit mode, the length of the input data must be an integer multiple of the number of bits per symbol.

```
x = randi([0 1],10*bitsPerSym,1);
```

Modulate the input data using a binary symbol mapping. Set the modulator to output fixed-point data. The numeric data type is signed with a 16-bit word length and a 10-bit fraction length.

```
y = qammod(x,M,'bin','InputType','bit','OutputDataType', ...
    numerictype(1,16,10));
```

Demodulate the 64-QAM signal. Verify that the demodulated data matches the input data.

```
z = qamdemod(y,M,'bin','OutputType','bit');
s = isequal(x,double(z))
```

```
s = logical
   1
```

Definitions

Gray Code

A Gray code, also known as a reflected binary code, is a system where the bit patterns in adjacent constellation points differ by only one bit.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`genqamdemod` | `genqammod` | `modnorm` | `pamdemod` | `pammod` | `qamdemod`

Topics

"Digital Modulation"

Introduced before R2006a

qfunc

Q function

Syntax

```
y = qfunc(x)
```

Description

`y = qfunc(x)` returns the output of the Q function for each element of the real array `x`. The Q function is one minus the cumulative distribution function of the standardized normal random variable.

Input Arguments

x — Input
scalar | vector | matrix | N-D array

Input, specified as a real scalar or array.

Output Arguments

y — Q function output
scalar | vector | matrix | N-D array

Q function output, returned as a real scalar or array having the same dimensions as input `x`.

Examples

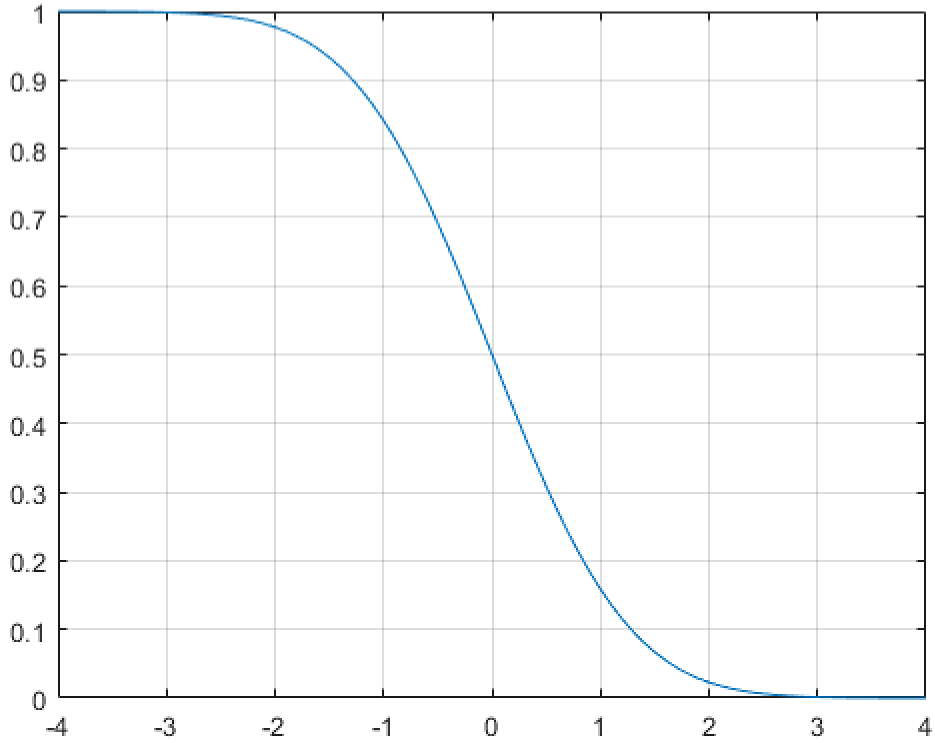
Q Function Results and Plot

Determine the values of the Q function for an input vector.

```
x = -4:0.1:4;
y = qfunc(x);
```

Plot the results.

```
plot(x,y)
grid
```



Calculate QPSK Error Probability Using Q Function

Convert an input Eb/No in dB to its linear equivalent.

```
ebnodB = 7;
ebno = 10^(ebnodB/10);
```

Determine the QPSK error probability, P_b, given that:

$$P_b = Q\left(\sqrt{2\frac{Eb}{No}}\right).$$

```
Pb = qfunc(sqrt(2*ebno))
```

```
Pb = 7.7267e-04
```

Algorithms

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

$$Q(x) = \frac{1}{2}\operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

See Also

erf | erfc | erfcinv | erfcx | erfinv | qfuncinv

Introduced before R2006a

qfuncinv

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

`erf` | `erfc` | `erfcinv` | `erfcx` | `erfinv` | `qfunc`

Introduced before R2006a

quantiz

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

`dpcmdeco` | `dpcmenco` | `lloyds`

Topics

“Quantize a Signal”

Introduced before R2006a

randdeintrlv

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`

Topics

"Interleaving"

Introduced before R2006a

randerr

Generate bit error patterns

Syntax

```
out = randerr(m)
out = randerr(m,n)
out = randerr(m,n,errors)
out = randerr(m,n,errors,seed)
out = randerr(m,n,errors,streamhandle)
```

Description

For all syntaxes, `randerr` treats each row of `out` independently.

`out = randerr(m)` generates an `m`-by-`m` binary matrix, where each row 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, where each row has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

`out = randerr(m,n,errors)` uses the `errors` input to determine the number of nonzero entries in each row of the output `m`-by-`n` binary matrix.

- 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. The elements in the second row of `errors` must sum to one.

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,errors,seed)` accepts a seed value for initializing the uniform random number generator `rand`.

`out = randerr(m,n,errors,streamhandle)` accepts a random stream handle to generate uniform random noise samples by using `rand`. Providing a random stream handle or using the `reset` function on the default random stream object enables you to generate repeatable noise samples. If you want to generate repeatable noise samples, then either reset the random stream input before calling `randerr` or use the same seed input. For more information, see `RandStream`.

Examples

Generate Random Error Matrix

Generate an 8-by-7 binary matrix in which each row is equally likely to have either zero or two nonzero elements.

```
out = randerr(8,7,[0 2])
```

```
out = 8×7

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

Now generate a matrix in which it is three times more likely that a row will have two nonzero elements.

```
out = randerr(8,7,[0 2; 0.25 0.75])
```

```
out = 8×7

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

See Also

Functions

`RandStream` | `rand` | `randi` | `randsrc`

Topics

"Sources and Sinks"

Introduced before R2006a

randintrlv

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`

Topics

"Interleaving"

Introduced before R2006a

randseed

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

`primes` | `rand`

Introduced before R2006a

rainpl

RF signal attenuation due to rainfall

Syntax

```
L = rainpl(range,freq,rainrate)
L = rainpl(range,freq,rainrate,elev)
L = rainpl(range,freq,rainrate,elev,tau)
```

Description

`L = rainpl(range,freq,rainrate)` returns the signal attenuation, `L`, due to rainfall. In this syntax, attenuation is a function of signal path length, `range`, signal frequency, `freq`, and rain rate, `rainrate`. The path elevation angle and polarization tilt angles are assumed to zero.

The `rainpl` function applies the International Telecommunication Union (ITU) rainfall attenuation model to calculate path loss of signals propagating in a region of rainfall [1]. The function applies when the signal path is contained entirely in a uniform rainfall environment. Rain rate does not vary along the signal path. The attenuation model applies only for frequencies at 1–1000 GHz.

`L = rainpl(range,freq,rainrate,elev)` specifies the elevation angle, `elev`, of the propagation path.

`L = rainpl(range,freq,rainrate,elev,tau)` specifies the polarization tilt angle, `tau`, of the signal.

Examples

Signal Attenuation Due to Rainfall

Compute the signal attenuation due to rainfall for a 20 GHz signal over a distance of 10 km in light and heavy rain.

Propagate the signal in a light rainfall of 1 mm/hr.

```
rr = 1.0;
L = rainpl(10000,20.0e9,rr)
```

```
L = 0.7104
```

```
L = 0.7104
```

```
L = 0.7104
```

Propagate the signal in a heavy rainfall of 10 mm/hr.

```
rr = 10.0;
L = rainpl(10000,20.0e9,rr)
```

```
L = 7.8413
```

```
L = 7.8413
```

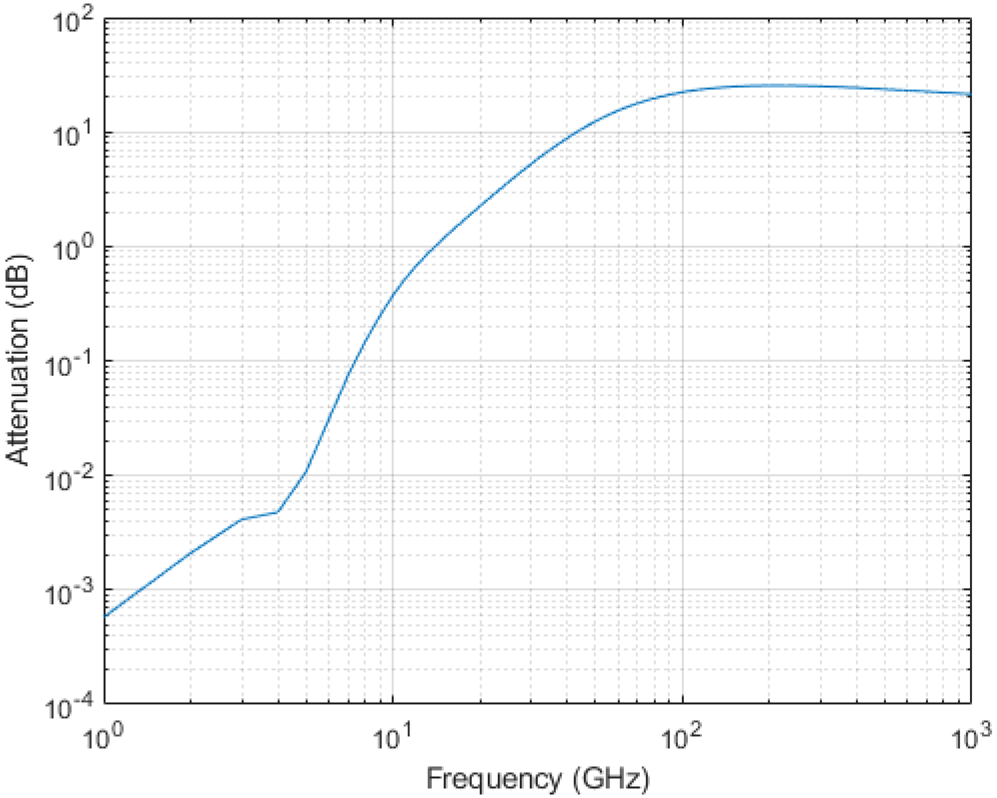
```
L = 7.8413
```

Signal Attenuation Due to Rainfall as Function of Frequency

Plot the signal attenuation due to moderate rainfall for signals in the frequency range from 1 to 1000 GHz. The path distance is 10 km.

Set the rain rate value for moderate rainfall to 3 mm/hr.

```
rr = 3.0;
freq = [1:1000]*1e9;
L = rainpl(10000,freq,rr);
loglog(freq/1e9,L)
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
```



Signal Attenuation Due to Rainfall as Function of Elevation Angle

Compute the signal attenuation due to heavy rain as a function of elevation angle. Elevation angles vary from 0 to 90 degrees. Assume a path distance of 100 km and a signal frequency of 100 GHz.

Set the rain rate to 10 mm/hr.

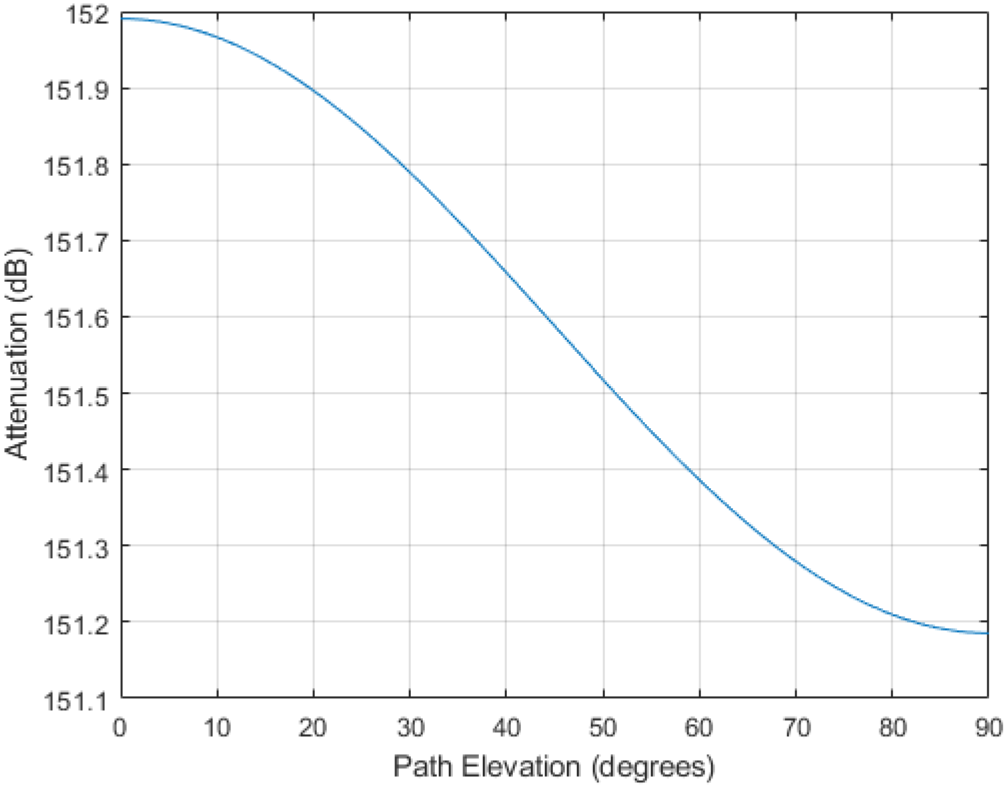
```
rr = 10.0;
```

Set the elevation angles, frequency, range.

```
elev = [0:1:90];
freq = 100.0e9;
rng = 100000.0*ones(size(elev));
```

Compute and plot the loss.

```
L = rainpl(rng,freq,rr,elev);
plot(elev,L)
grid
xlabel('Path Elevation (degrees)')
ylabel('Attenuation (dB)')
```



Signal Attenuation Due to Rainfall as Function of Polarization

Compute the signal attenuation due to heavy rainfall as a function of the polarization tilt angle. Assume a path distance of 100 km, a signal frequency of 100 GHz signal, and a path elevation angle of 0 degrees. Set the rainfall rate to 10 mm/hour. Plot the signal attenuation versus polarization tilt angle.

Set the polarization tilt angle to vary from -90 to 90 degrees.

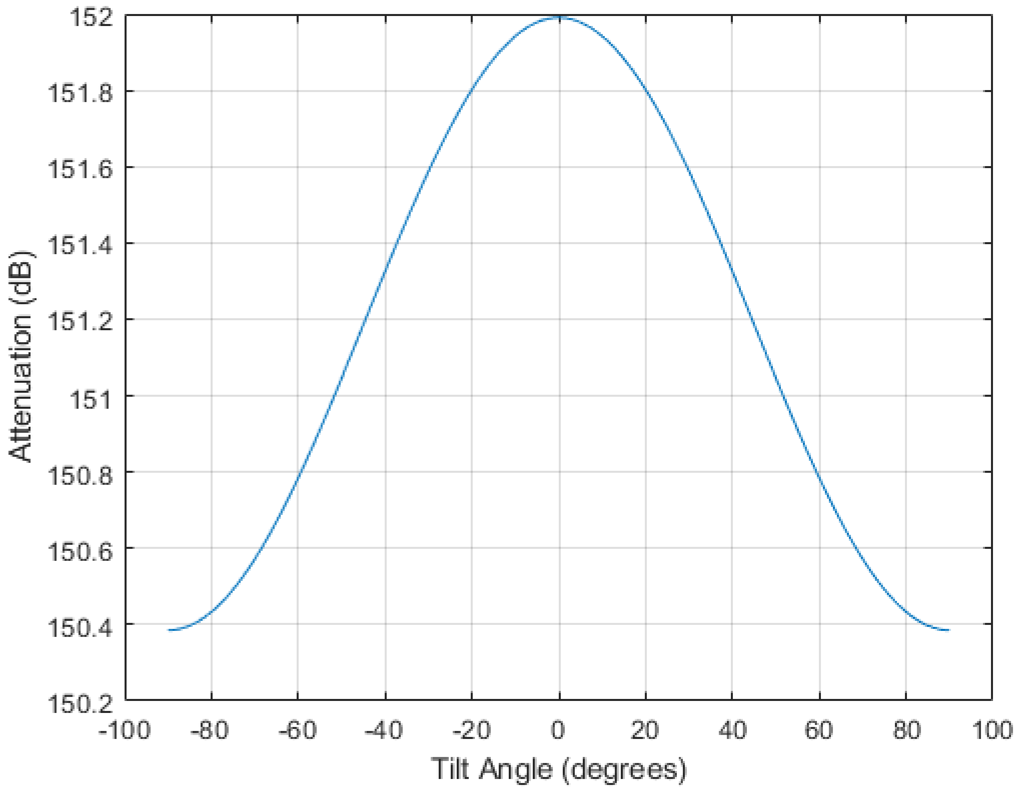
```
tau = -90:90;
```

Set the elevation angle, frequency, path distance, and rain rate.

```
elev = 0;
freq = 100.0e9;
rng = 100e3*ones(size(tau));
rr = 10.0;
```

Compute and plot the attenuation.

```
L = rainpl(rng,freq,rr,elev,tau);
plot(tau,L)
grid
xlabel('Tilt Angle (degrees)')
ylabel('Attenuation (dB)')
```



Input Arguments

`range` — Signal path length

nonnegative real-valued scalar | nonnegative real-valued *M*-by-1 column vector | nonnegative real-valued 1-by-*M* row vector

Signal path length, specified as a nonnegative real-valued scalar, or as a *M*-by-1 or 1-by-*M* vector. Units are in meters.

Example: `[13000.0,14000.0]`

`freq` — Signal frequency
positive real-valued scalar | nonnegative real-valued *N*-by-1 column vector | nonnegative real-valued 1-by-*N* row vector

Signal frequency, specified as a positive real-valued scalar, or as a nonnegative *N*-by-1 or 1-by-*N* vector. Frequencies must lie in the range 1–1000 GHz.

Example: `[1400.0e6,2.0e9]`

`rainrate` — Rain rate
nonnegative real-valued scalar

Rain rate, specified as a nonnegative real-valued scalar. Units are in mm/hr.

Example: `1.5`

`elev` — Signal path elevation angle
0.0 (default) | real-valued scalar | real-valued *M*-by-1 column vector | real-valued 1-by-*M* row vector

Signal path elevation angle, specified as a real-valued scalar, or as an *M*-by-1 or 1-by- *M* vector. Units are in degrees between –90° and 90°. If `elev` is a scalar, all propagation paths have the same elevation angle. If `elev` is a vector, its length must match the dimension of `range` and each element in `elev` corresponds to a propagation range in `range`.

Example: `[0,45]`

`tau` — Tilt angle of polarization ellipse
0.0 (default) | real-valued scalar | real-valued *M*-by-1 column vector | real-valued 1-by-*M* row vector

Tilt angle of the signal polarization ellipse, specified as a real-valued scalar, or as an *M*-by-1 or 1-by- *M* vector. Units are in degrees between –90° and 90°. If `tau` is a scalar, all signals have the same tilt angle. If `tau` is a vector, its length must match the dimension of `range`. In that case, each element in `tau` corresponds to a propagation path in `range`.

The tilt angle is defined as the angle between the semimajor axis of the polarization ellipse and the *x*-axis. Because the ellipse is symmetrical, a tilt angle of 100° corresponds to the same polarization state as a tilt angle of -80°. Thus, the tilt angle need only be specified between ±90°.

Example: `[45,30]`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

Definitions

Rainfall Attenuation Model

This model calculates the attenuation of signals that propagate through regions of rainfall.

Electromagnetic signals are attenuate when propagating through a region of rainfall. Rainfall attenuation is computed according to the ITU rainfall model *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The model computes the specific attenuation (attenuation per kilometer) of a signal as a function of rainfall rate, signal frequency, polarization, and path elevation angle. To compute the attenuation, this model uses

$$\gamma_r = kr^{\alpha},$$

where r is the rain rate in mm/hr. The parameter k and exponent α depend on the frequency, the polarization state, and the elevation angle of the signal path. The specific attenuation model is valid for frequencies from 1–1000 GHz.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by a propagation distance, R. Then, total attenuation is $L_r = R\gamma_r$. Instead of using geometric range as the propagation distance, the toolbox uses a modified range. The modified range is the geometric range multiplied by a range factor

$$\frac{1}{1+\dfrac{R}{R_0}}$$

where

$$R_0 = 35e^{-0.015r}$$

is the effective path length in kilometers (see Seybold, J. *Introduction to RF Propagation*.) When there is no rain, the effective path length is 35 km. When the rain rate is, for example, 10 mm/hr, the effective path length is 30.1 km. At short range, the propagation distance is approximately the geometric range. For longer ranges, the propagation distance asymptotically approaches the effective path length.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. 2005.

[2] Seybold, J. *Introduction to RF Propagation*. New York: Wiley & Sons, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fogpl` | `fspl` | `gaspl`

Introduced in R2016a

randsrc

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,___,seed)
out = randsrc(m,n,___,streamhandle)
```

Description

`out = randsrc` generates a random scalar that is either `-1` or `1`, with equal probability.

`out = randsrc(m)` generates an `m`-by-`m` random bipolar matrix. Each entry independently takes the value `-1` or `1` with equal probability.

`out = randsrc(m,n)` generates an `m`-by-`n` random bipolar matrix. Each entry independently takes the value `-1` or `1` with equal probability.

`out = randsrc(m,n,alphabet)` generates an `m`-by-`n` matrix, with each entry 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, with each entry 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,___,seed)` accepts input combinations from prior syntaxes and a seed value, for initializing the uniform random number generator, `rand`.

`out = randsrc(m,n,___,streamhandle)` accepts input combinations from prior syntaxes and a random stream handle to generate uniform random noise samples by using `rand`. Providing a random stream handle or using the `reset` function on the default random stream object enables you to generate repeatable noise samples. If you want to generate repeatable noise samples, then either reset the random stream input before calling `randsrc` or use the same seed input. For more information, see `RandStream`.

Examples

Generate Random Matrix from Prescribed Alphabet

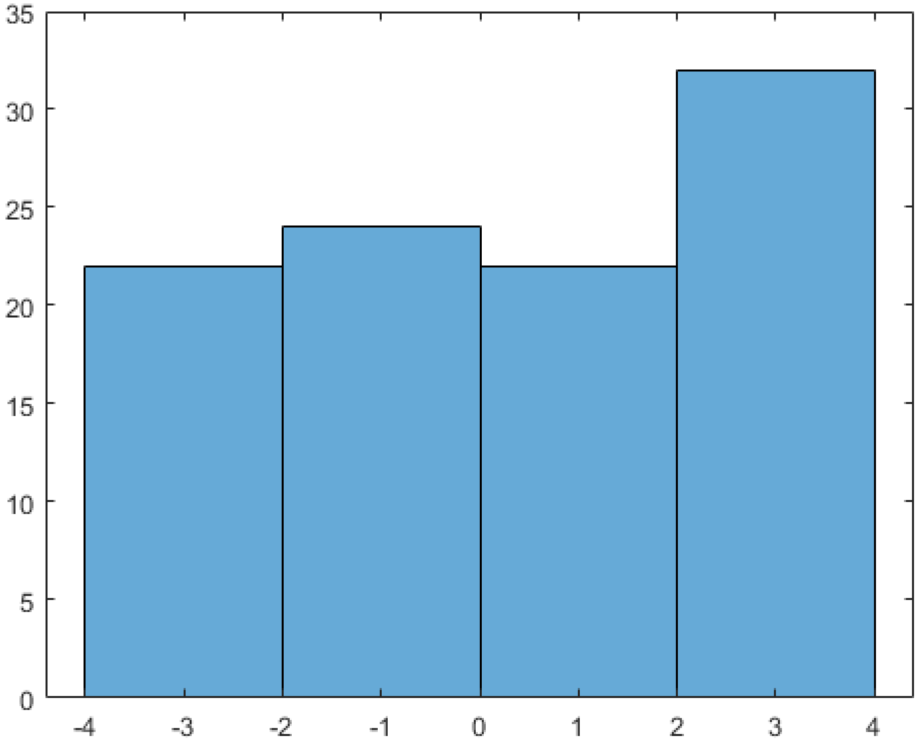
Generate a 10-by-10 matrix from the set of {-3,-1,1,3}.

```
out = randsrc(10,10,[-3 -1 1 3])
```

```
out = 10×10

     3    -3     1     1    -1    -1     3     3    -1    -3
     3     3    -3    -3    -1     1    -1    -1     3    -3
    -3     3     3    -1     3     1     1     3     1     1
     3    -1     3    -3     3    -3     1    -3     1     3
     1     3     1    -3    -3    -3     3     3     3     3
    -3    -3     3     3    -1    -1     3    -1    -1    -3
    -1    -1     1     1    -1     3     1    -3     3     1
     1     3    -1    -1     1    -1    -3    -1     3    -1
     3     3     1     3     1     1    -3     1    -1    -3
     3     3    -3    -3     3    -3    -1    -1     1    -1
```

Plot the histogram. Each of the four possible element values occur with equal probability. Your values might differ.

```
histogram(out,[-4 -2 0 2 4])
```



Generate a matrix in which the likelihood of a -1 or 1 is four times higher than the likelihood of a -3 or 3.

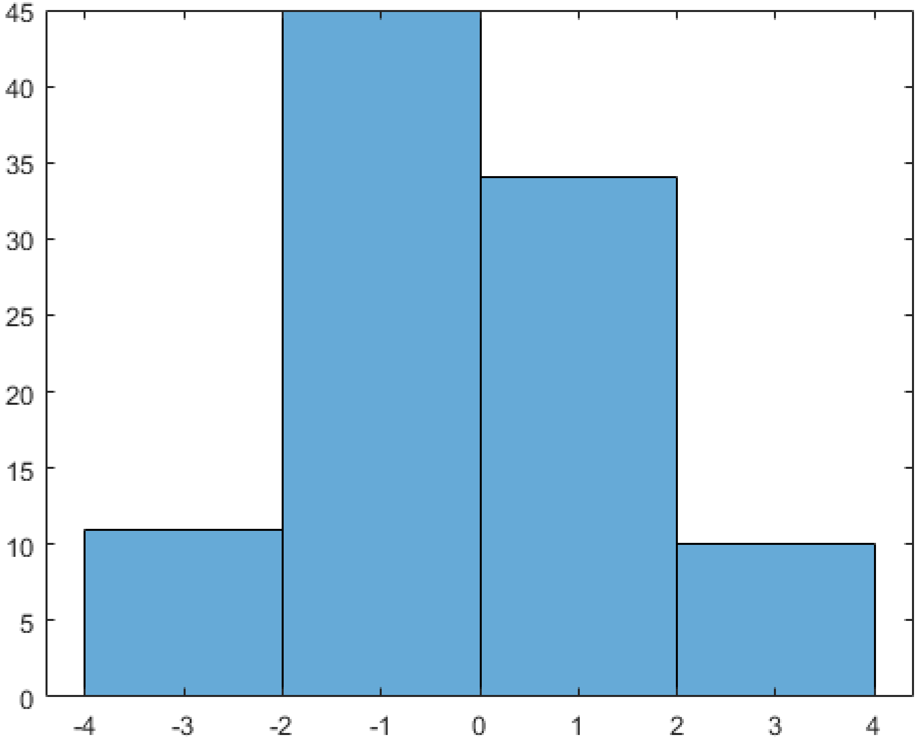
```
out = randsrc(10,10,[-3 -1 1 3; 0.1 0.4 0.4 0.1])
```

```
out = 10×10

    -1    -1    -1    -1     1    -1     1    -1     1     3
     1    -3     3     3     1    -3    -1    -1    -1     1
    -1    -1    -3    -1    -1     3    -1     1     1    -1
     1     3     1    -1     1     3    -1    -3    -1    -1
    -1    -1     1    -1    -1    -1    -3    -3     1    -1
     1     1     1    -1    -3    -1    -1    -1    -1    -1
    -1     1    -3     1    -1    -1     3     1    -1     1
```

```
     1      3     -1      1     -1      3      3      1      1      1
     1     -3     -1      1     -1     -1      1      1      1      1
     1     -1      1     -1     -1     -1     -3     -1     -3      1
```

Plot the histogram. Values of -1 and 1 are more likely.

```
histogram(out,[-4 -2 0 2 4])
```



See Also

Functions
`RandStream` | `rand` | `randerr` | `randi`

Introduced before R2006a

rayleighchan

(To be removed) Construct Rayleigh fading channel object

Syntax

```
chan = rayleighchan(ts,fd)
chan = rayleighchan(ts,fd,tau,pdb)
chan = rayleighchan
```

Note `rayleighchan` will be removed in a future release. Use `comm.RayleighChannel` instead.

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 “Displaying and Changing 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`.
`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
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, plot (channel). You can use the channel visualization tool in Normal mode and Accelerator mode.

Examples

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

`comm.RayleighChannel`

Topics

“Fading Channels”

Introduced before R2006a

rectpulse

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

Introduced before R2006a

reset (channel)

(To be removed) Reset channel object

Syntax

```
reset(chan)
reset(chan,randstate)
```

Note This function will be removed in a future release. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

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.

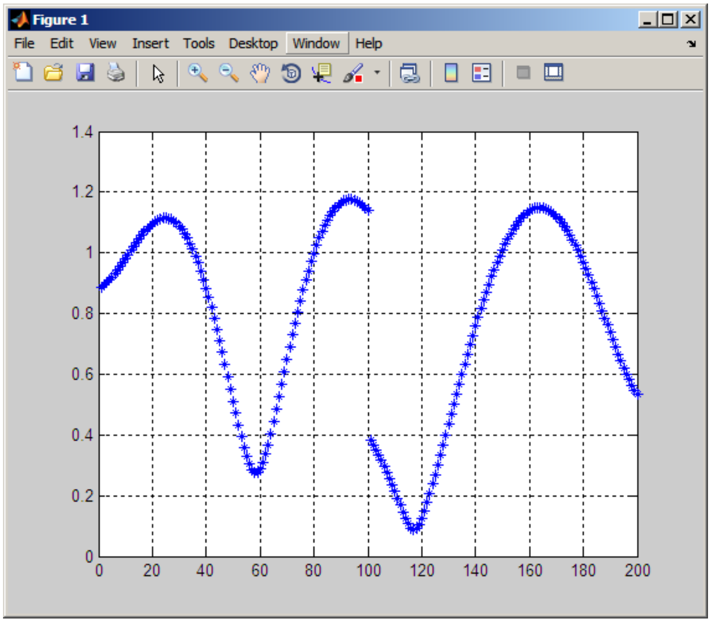
Examples

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 a plot similar to this figure.



See Also

`comm.RayleighChannel` | `comm.RicianChannel`

Topics

"Fading Channels"

Introduced in R2007a

reset (equalizer)

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`

Topics

"Equalization"

Introduced before R2006a

ricianchan

(To be removed) 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
```

Note ricianchan will be removed in a future release. Use comm.RicianChannel instead.

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).

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 “Displaying and Changing 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.
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.

Property	Description
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:

- The Kfactor property contains the value of K_r (if it’s a scalar) or $\left\{K_{r,k}\right\}$, $1 \leq k \leq K$ (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).

The `rayleighchan` reference page includes descriptions for properties common to both Rayleigh and Rician channel objects.

Channel Visualization

The characteristics of a channel can be plotted using the channel visualization tool, `plot (channel)`. You can use the channel visualization tool in Normal mode and Accelerator mode.

References

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

See Also

`comm.RicianChannel`

Topics

"Fading Channels"

Introduced before R2006a

rls

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

Property	Description
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

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

Topics

"Equalization"

Introduced before R2006a

rsdec

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. *`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

Reed-Solomon Decoding

Set the RS code parameters.

```
m = 3;                        % Number of bits per symbol
n = 2^m-1;                    % Codeword length
k = 3;                        % Message length
```

Generate three codewords composed of 3-bit symbols. Encode the message with a (7,3) RS code.

```
msg = gf([2 7 3; 4 0 6; 5 1 1],m);
code = rsenc(msg,n,k);
```

Introduce one error on the first codeword, two errors on the second codeword, and three errors on the third codeword.

```
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;
```

Decode the corrupted codeword.

```
[rxcode,cnumerr] = rsdec(noisycode,n,k);
```

Observe that the number of corrected errors matches the introduced errors for the first two rows. In row three, the number of corrected errors is `-1` because a (7,3) RS code cannot correct more than two errors.

```
cnumerr
```

```
cnumerr = 3×1
```

```
     1
     2
    -1
```

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 1-918 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

`gf` | `rsenc` | `rsgenpoly`

Topics

“Block Codes”

Introduced before R2006a

rsenc

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`. *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`.

Examples

Reed-Solomon Code Generation

Set the code parameters.

```
m = 3;           % Number of bits per symbol
n = 2^m - 1;     % Codeword length
k = 3;           % Message length
```

Create two messages based on GF(8).

```
msg = gf([2 7 3; 4 0 6],m)
```

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

Array elements = 

   2   7   3
   4   0   6
```

Generate RS (7,3) codewords.

```
code = rsenc(msg,n,k)
```

```
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
```

The codes are systematic so the first three symbols of each row match the rows of `msg`.

Limitations

`n` and `k` must differ by an integer. `n` between 7 and 65535.

See Also

`gf` | `rsdec` | `rsgenpoly`

Topics

“Block Codes”

"Represent Words for Reed-Solomon Codes"
"Create and Decode Reed-Solomon Codes"

Introduced before R2006a

rsgenpoly

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 = rsgenpoly(n,k,prim_poly,b,outputFormat)
[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 = rsgenpoly(n,k,prim_poly,b,outputFormat) is the same as the syntax above, except that outputFormat specifies the 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] = 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

gf | rsdec | rsenc

Topics

"Block Codes"

Introduced before R2006a

rsgenpolycoeffs

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.

Examples

Generate Polynomial Coefficients for a Reed-Solomon Code

This example shows how to generate polynomial coefficients for a (15,11) Reed-Solomon code.

Generate the coefficients using `rsgenpolycoeffs`.

```
genpoly = rsgenpolycoeffs(15,11)
```

```
genpoly = 1x5 uint32 row vector

    1   13   12    8    7
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation, these usage notes and limitations apply:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`gf` | `rsdec` | `rsenc` | `rsgenpoly`

Introduced in R2010b

scatterplot

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 character vector 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

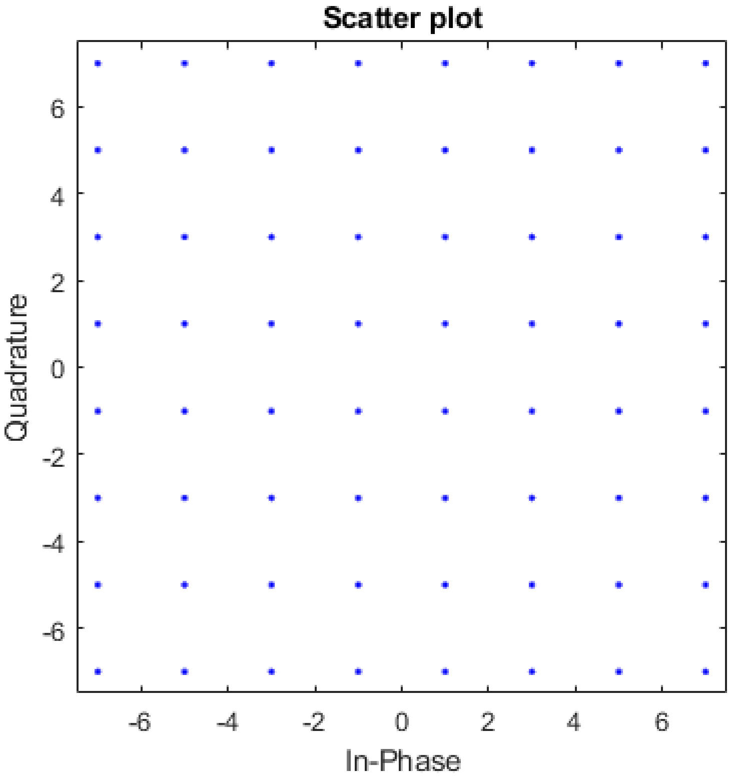
Generate Scatter Plot of 64-QAM Signal

Create a 64-QAM signal in which each constellation point is used.

```
d = (0:63)';
s = qammod(d,64);
```

Display the scatter plot of the constellation.

```
scatterplot(s)
```



- “Scatter Plots and Constellation Diagrams”

Tips

Use `comm.ConstellationDiagram` when these are required:

- Measurements
- Basic reference constellations
- Signal trajectory plots
- Maintaining state between calls

Use `scatterplot` when:

- A simple snapshot of the signal constellation is needed.

See Also

`comm.ConstellationDiagram` | `comm.EyeDiagram` | `plot` | `scatter`

Topics

"Scatter Plots and Constellation Diagrams"

Introduced before R2006a

semianalytic

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(...)
```

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
Phase shift keying (PSK) with differential encoding, where the phase offset of the constellation is 0	`'psk/diff'`	2, 4

Modulation Scheme	Value of *modtype*	Valid Values of *M*
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 quadrature 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.

Alternatives

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

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`

Topics

“Performance Results via the Semianalytic Technique”

Introduced before R2006a

shift2mask

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 polynomial character vector
- 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.

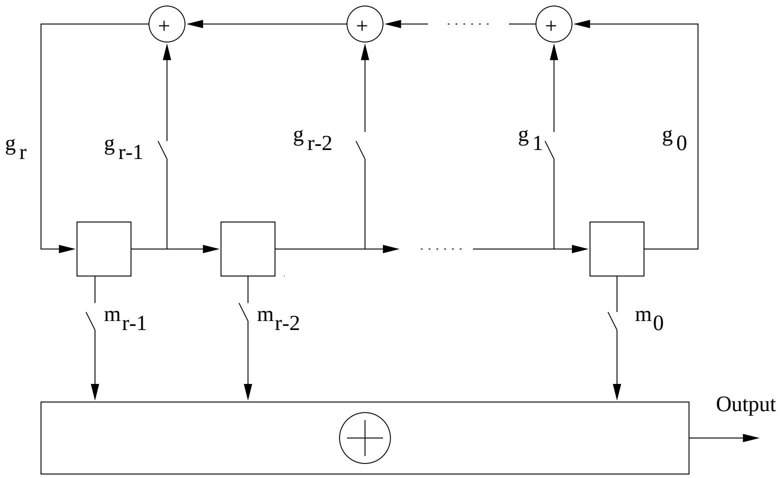
Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

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

Convert Shift to Mask

Convert a shift in a linear feedback shift register into an equivalent mask.

Convert 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 $x^4 + x^3 + 1$. The length of the mask is equal to the degree of the primitive polynomial, 4.

```
mk = shift2mask([1 1 0 0 1],5)
```

```
mk = 1×4

     1     0     1     1
```

Convert a shift of 7 to a mask of $x^4 + x^2$ for the primitive polynomial $x^5 + x^2 + 1$.

```
mk2 = shift2mask('x5+x2+1',7)
```

```
mk2 = 1×5

     1     0     1     0     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

`deconv` | `isprimitive` | `mask2shift` | `primpoly`

Introduced before R2006a

signlms

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.

Examples

Create a Linear Equalizer using Signed LMS Algorithm

This example shows to use a signed least mean square (LMS) algorithm to create an adaptive equalizer object.

Set the number of weights and the step size for the equalizer.

```
nWeights = 2;
stepSize = 0.05;
```

Create the adaptive algorithm object using the signed regressor LMS algorithm type.

```
alg = signlms(stepSize,'Signed Regressor LMS');
```

Construct a linear equalizer using the algorithm object.

```
eqObj = lineareq(nWeights,alg)
```

```
eqObj =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Signed Regressor LMS'
                nWeights: 2
             nSampPerSym: 1
```

```
               RefTap: 1
             SigConst: [-1 1]
             StepSize: 0.0500
        LeakageFactor: 1
              Weights: [0 0]
         WeightInputs: [0 0]
 ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

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

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

Topics

"Equalization"

Introduced before R2006a

ssbdemod

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

Demodulate Sideband Signal

Define the sampling frequency and original signal.

```
fs = 270000;
t = (0:1/fs:0.01)';
signal = sin(2*pi*300.*t)+2*sin(2*pi*600.*t);
```

Convert the original signal to upper-sideband and lower-sideband modulated signals using `ssbmod`. Use a cutoff frequency of `12000` and an initial phase of `0.`

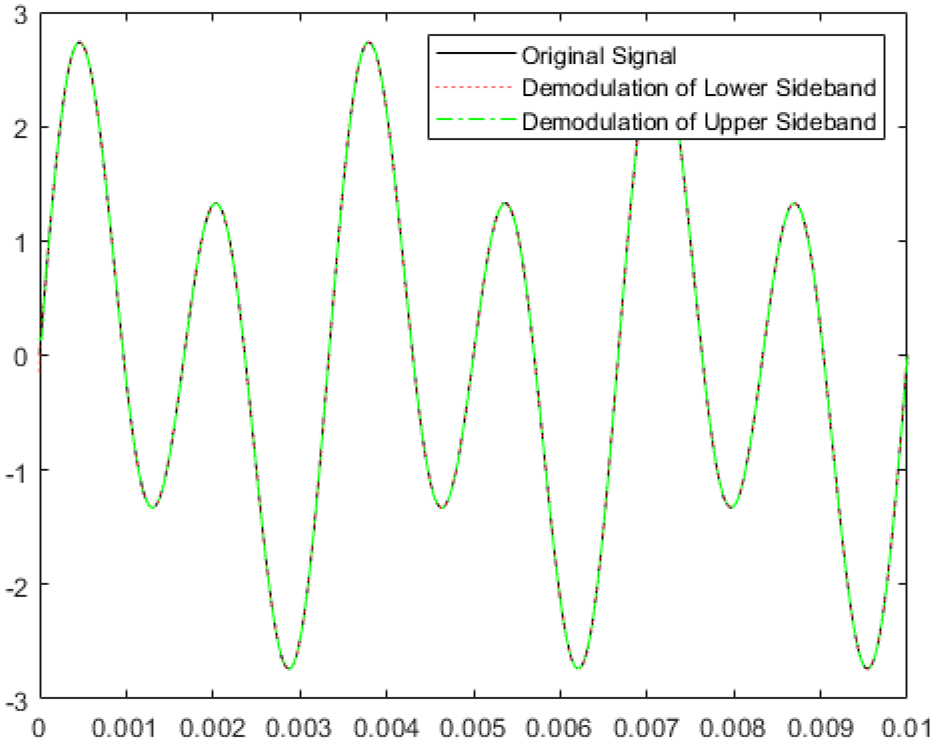
```
fc = 12000;
initialPhase = 0;
lowerSidebandSignal = ssbmod(signal,fc,fs,initialPhase);
upperSidebandSignal = ssbmod(signal,fc,fs,initialPhase,'upper');
```

Demodulate the lower and upper sideband signals.

```
s1 = ssbdemod(lowerSidebandSignal,fc,fs);
s2 = ssbdemod(upperSidebandSignal,fc,fs);
```

Compare processed signals with original and verify reconstruction.

```
plot(t,signal,'k',t,s1,'r:',t,s2,'g-.');
legend('Original Signal','Demodulation of Lower Sideband','Demodulation of Upper Sideba
```



See Also

`amdemod` | `ssbmod`

Topics

"Digital Modulation"

Introduced before R2006a

ssbmod

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

`ammod` | `ssbdemod`

Topics

"Digital Modulation"

Introduced before R2006a

stdchan

Construct channel System object from set of standardized channel models

Syntax

```
chan = stdchan(chantype,rs,fd)
```

Description

`chan = stdchan(chantype,rs,fd)` constructs a fading channel object `chan` according to the specified `chantype`. `chantype` is chosen from the channel models listed in “Supported Standards” on page 1-959. `rs` is the sampling rate of the input signal and `fd` is the maximum Doppler shift.

Examples

Filter Signal Through CDMA Channel

Set the sample rate and the maximum Doppler shift.

```
rs = 20e6;
fd = 3;
```

Create a CDMA Typical Urban channel model (TUx) channel object and turn on frequency response visualization.

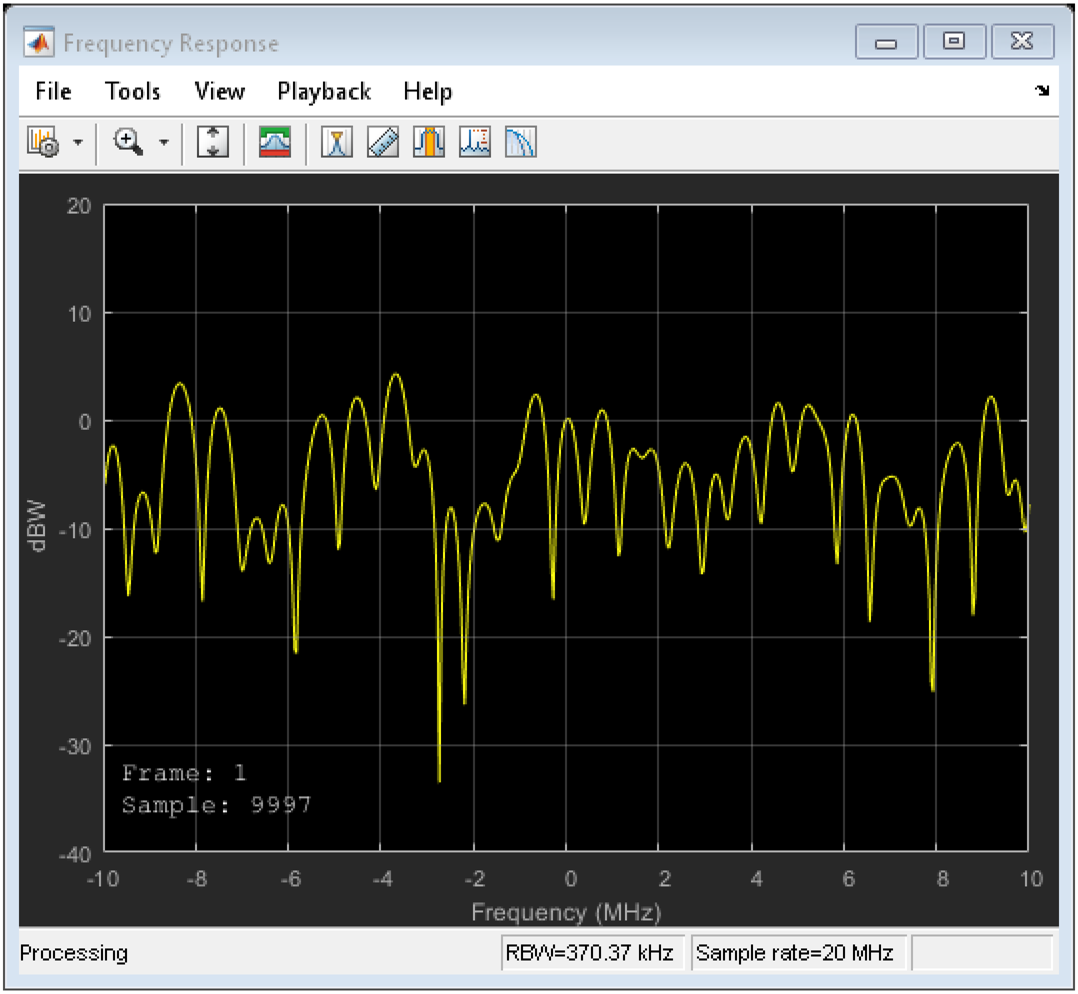
```
chan = stdchan('cdmaTUx',rs,fd);
chan.Visualization = 'Frequency response';
```

Generate random data and apply QPSK modulation.

```
data = randi([0 3],10000,1);
txSig = pskmod(data,4,pi/4);
```

Filter the QPSK signal through the CDMA channel.

```
y = chan(txSig);
```



Frame: 1
Sample: 9997

Processing

RBW=370.37 kHz

Sample rate=20 MHz

GSM and EDGE Channel Model

Create a channel model useful for GSM and EDGE simulations. Experiment with low speed and high speed conditions.

Configure parameters and System objects

Frame configuration.

```
M = 8; % Modulation order, 8-PSK
Rbit = 9600; % Input bit rate
Rs = Rbit / log2(M); % Symbol rate
Nsamples = 5e2; % Number of samples per frame
Nframes = 10; % Number of frames
```

Speed and channel configuration.

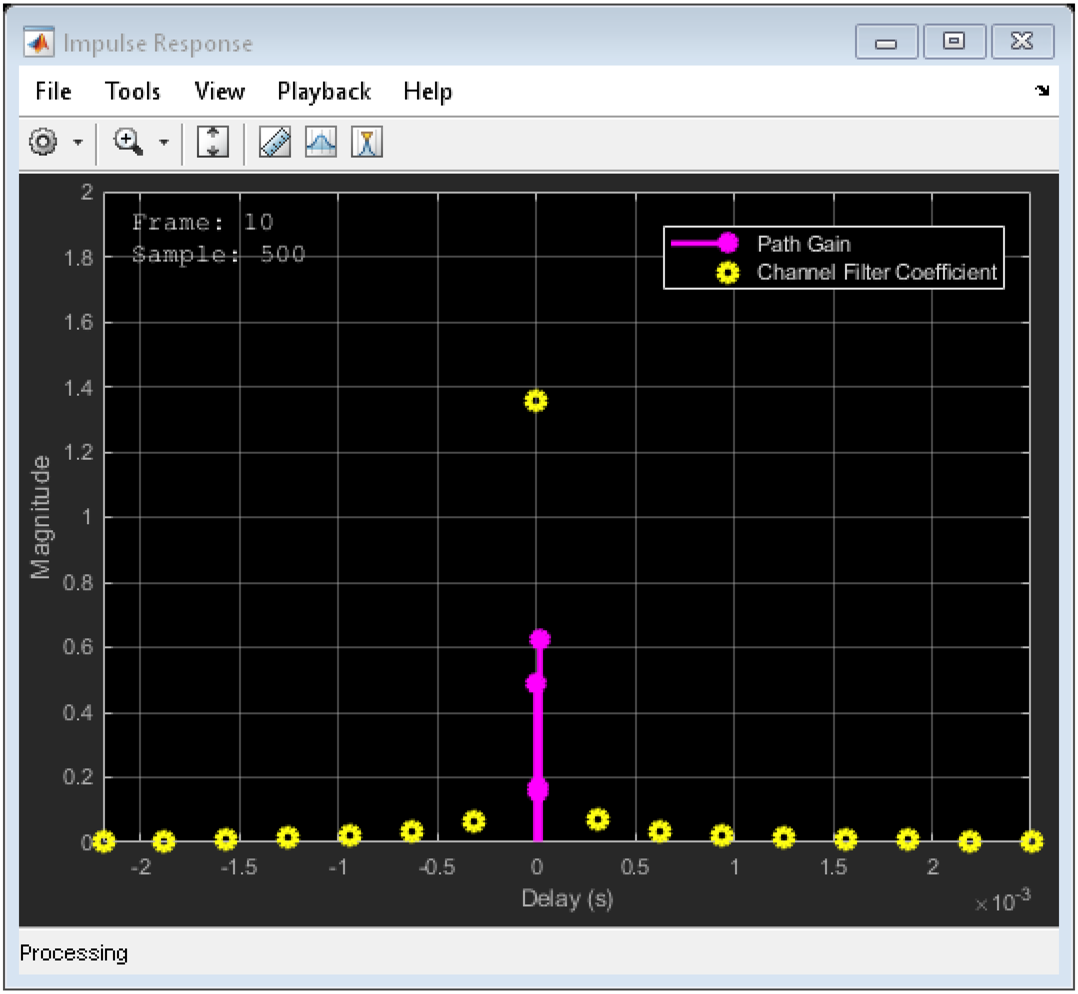
```
v = 10 * 1e3/3600; % Mobile speed (m/s)
fc = 1800e6; % Carrier frequency
c = physconst('LightSpeed'); % Speed of light in free space
fd = v*fc/c; % Maximum Doppler shift of diffuse component
```

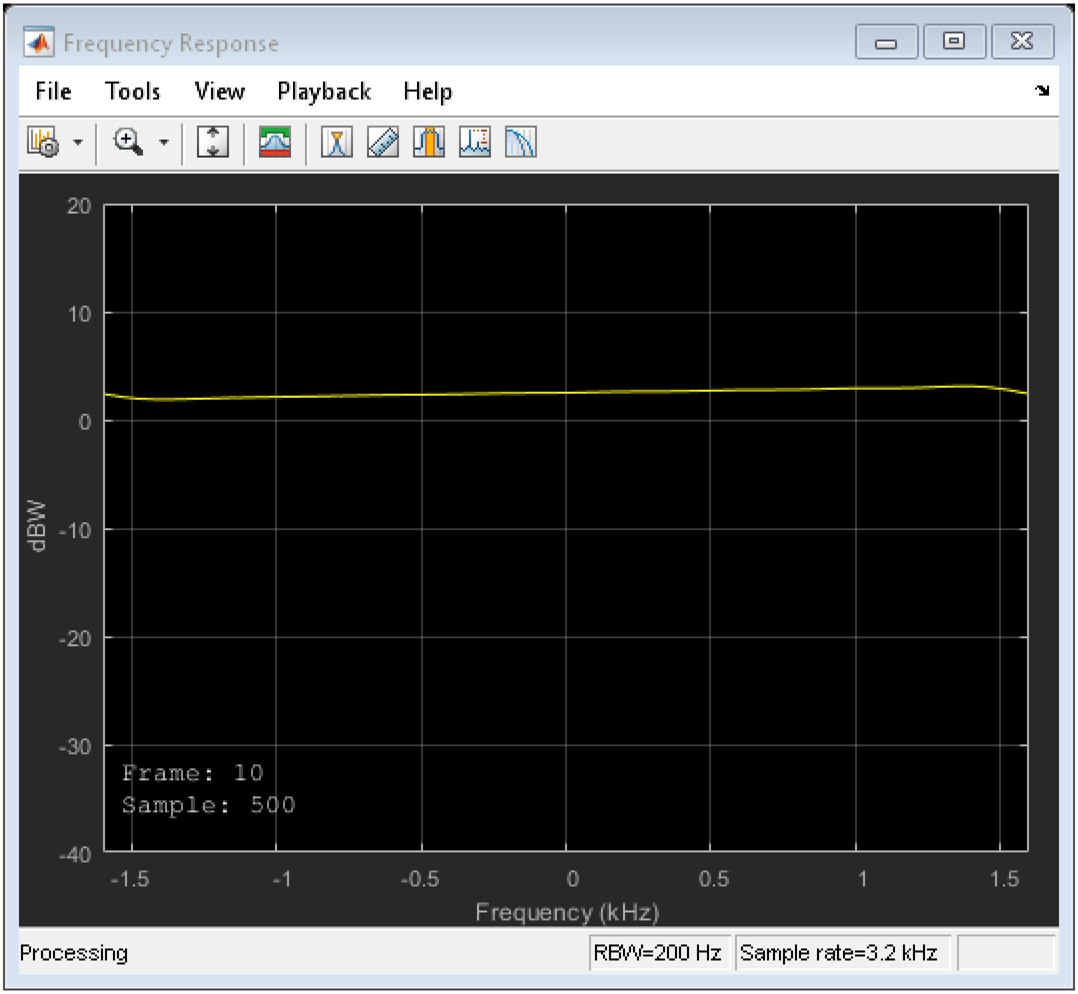
Create System objects for modulator and channel.

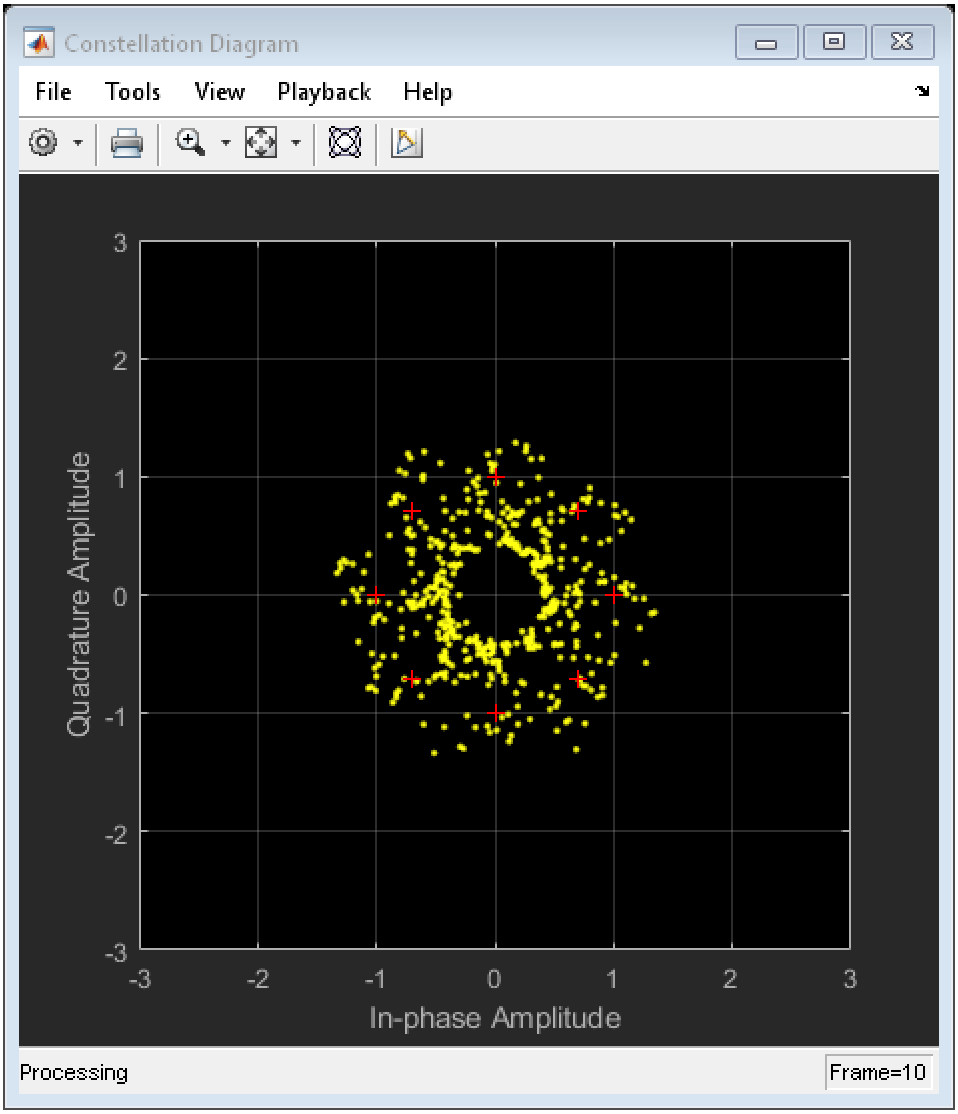
```
modulator  = comm.PSKModulator(M,'PhaseOffset',0);

channel = stdchan('gsmeqx6',Rs,fd);
channel.RandomStream = 'mt19937ar with seed'; % set for reproducability
channel.Visualization = 'Impulse and frequency responses';
channel.SamplesToDisplay = '100%';

refC = constellation(modulator);
constDiagram = comm.ConstellationDiagram( ...
    'ReferenceConstellation',refC, ...
    'XLimits',[-3 3],'YLimits',[-3 3]);
```

Simulate at low speed

```
for iFrames = 1:Nframes
    msg = randi([0 M-1], Nsamples, 1);
    modSignal = modulator(msg);
    chanOut = channel(modSignal);
    constDiagram(chanOut);
end
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 10
Sample: 500
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Delay (s)
$\times 10^{-3}$
Processing

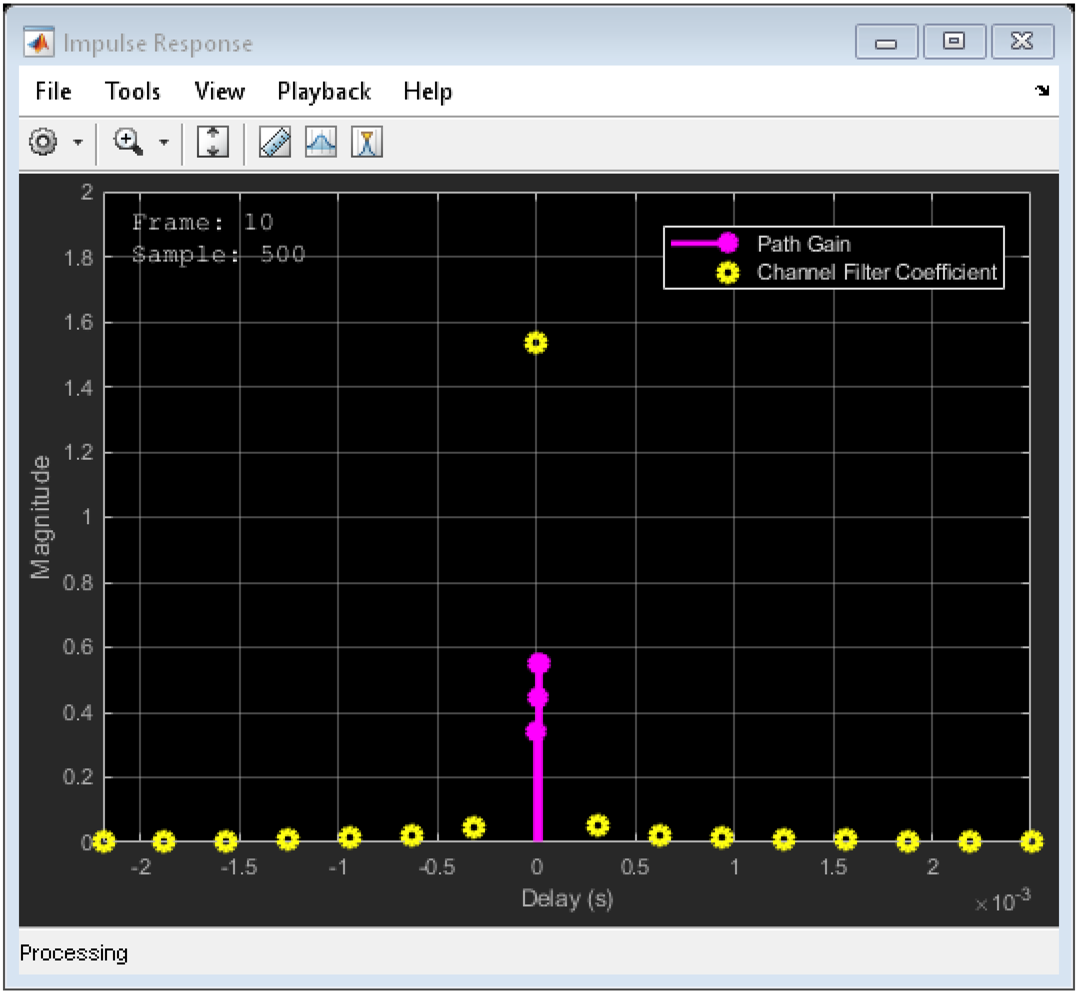

Frequency Response
File
Tools
View
Playback
Help
20
10
0
-10
-20
-30
-40
dBW
-1.5
-1
-0.5
0
0.5
1
1.5
Frequency (kHz)
Frame: 10
Sample: 500
Processing
RBW=200 Hz
Sample rate=3.2 kHz

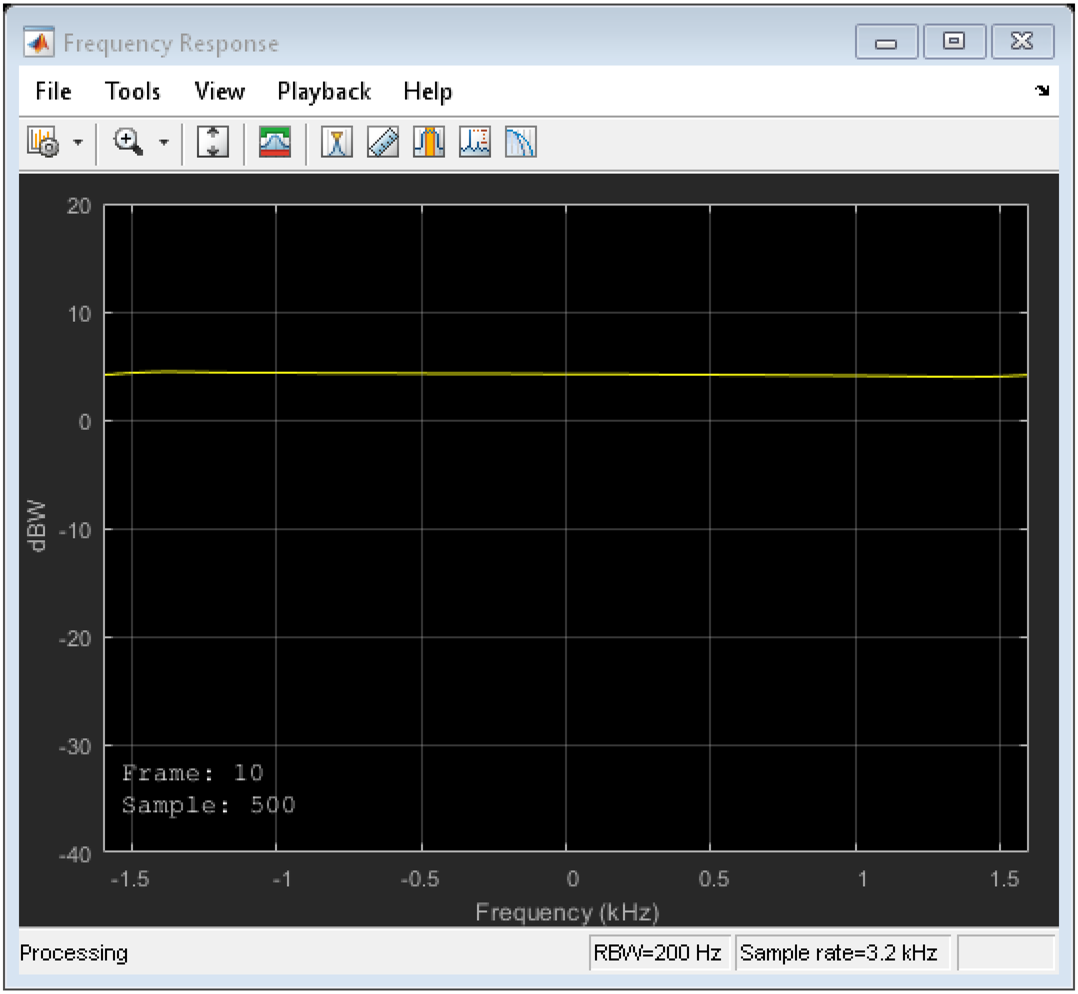


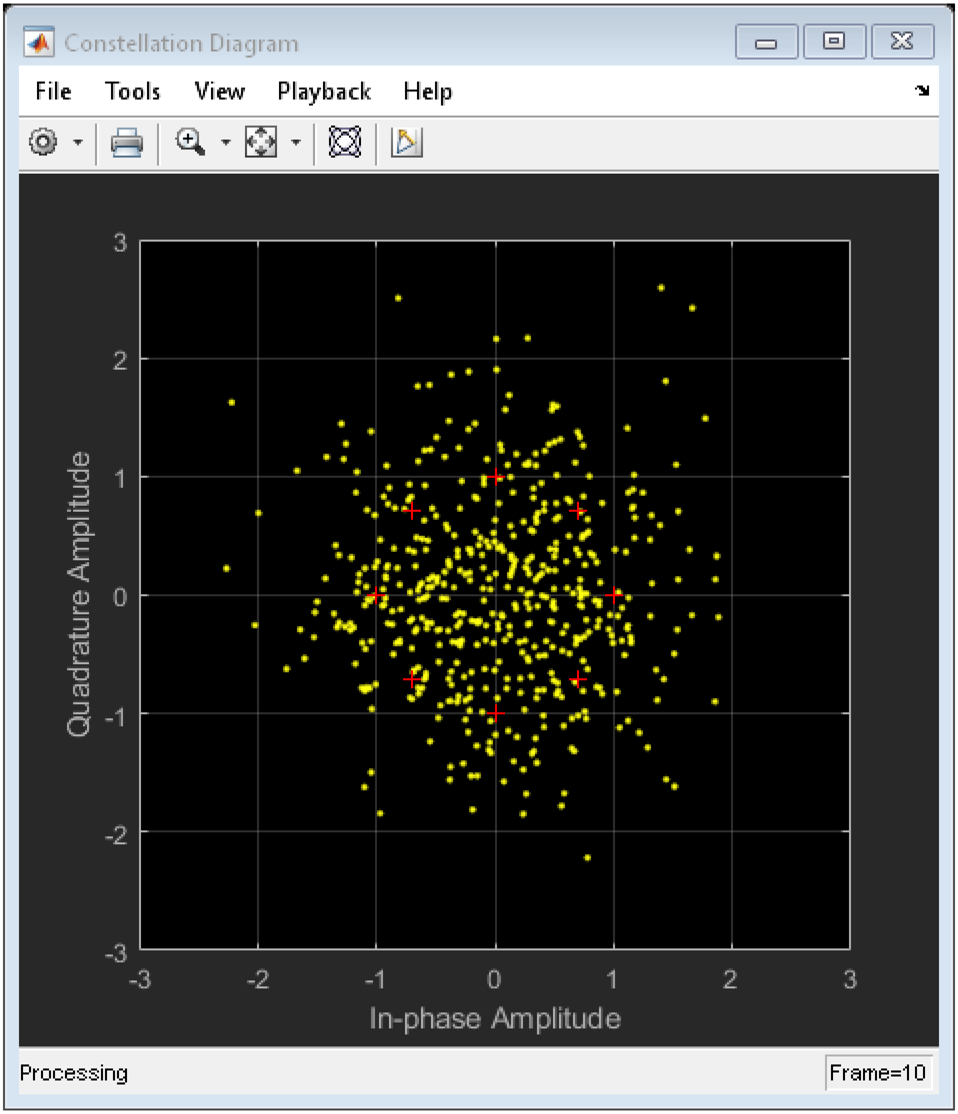
Simulate at high speed

Release and reconfigure objects.

```
release(constDiagram);
release(channel);

v = 120 * 1e3 / 3600; % Mobile speed (m/s)
fd = v*fc/c; % Maximum Doppler shift of diffuse component

channel.MaximumDopplerShift = fd; % Adjust maximum doppler shift

for iFrames = 1:Nframes
    msg = randi([0 M-1], Nsamples, 1);
    modSignal = modulator(msg);
    chanOut = channel(modSignal);
    constDiagram(chanOut);
end
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 10
Sample: 500
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Delay (s)
$\times 10^{-3}$
Processing


Frequency Response
File
Tools
View
Playback
Help
20
10
0
-10
-20
-30
-40
dBW
-1.5
-1
-0.5
0
0.5
1
1.5
Frequency (kHz)
Frame: 10
Sample: 500
Processing
RBW=200 Hz
Sample rate=3.2 kHz



Input Arguments

chantype — Channel type
string | character vector

Channel type, specified as a string or character vector. Valid options are listed in “Supported Standards” on page 1-959.

Example: `stdchan('gsmRAx6c1',rs,fd)`, configures a channel model for the GSM typical case for rural area (RAx), 6 taps, case 1, with a sample rate `rs`, and maximum Doppler shift `fd`

Data Types: `char` | `string`

`rs` — Sample rate
scalar

Sample rate in Hertz, specified as a scalar.

Data Types: `double`

`fd` — Maximum Doppler shift
scalar

Maximum Doppler shift in Hertz, specified as a scalar.

Data Types: `double`

Output Arguments

chan — Channel object
System object

Channel object, returned as a `comm.RayleighChannel` or `comm.RicianChannel` System object.

Definitions

Supported Standards

For GSM, CDMA, and ITU-R HF standards, call `stdchan` to return a `comm.RayleighChannel` or `comm.RicianChannel` System object modeling one of these profiles.

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
`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

CDMA channel models for deployment evaluation (3GPP TR 25.943 V6.0.0 (2004-12)):

Channel model	Profile
`cdmaTUx`	Typical Urban channel model (TUx)
`cdmaRAx`	Rural Area channel model (RAx)
`cdmaHTx`	Hilly Terrain channel model (HTx)

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
`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

See Also

Functions
`doppler`

System Objects
`comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2007b

symerr

Compute number of symbol errors and symbol error rate

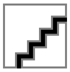
Syntax

```
[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.



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
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`.

Compare Elements of Matrix

Compare Elements of Matrix with Another Matrix

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4

     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aMatrix = [1,1,1,1;2,2,2,2;3,3,3,3]
```

```
aMatrix = 3×4

     1     1     1     1
     2     2     2     2
     3     3     3     3
```

```
[number1,ratio1] = symerr(x,aMatrix)
```

```
number1 = 3
```

```
ratio1 = 0.2500
```

Compare Elements of Matrix with Row Vector

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4
```

```
     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aRowVector = [1,2,3,1]
```

```
aRowVector = 1×4

     1     2     3     1
```

```
[number2,ratio2] = symerr(x,aRowVector)
```

```
number2 = 3×1

     1
     3
     4

ratio2 = 3×1

    0.2500
    0.7500
    1.0000
```

Compare Elements of Matrix with Column Vector

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4

     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aColumnVector = [1;2;3]
```

```
aColumnVector = 3×1

     1
     2
     3
```

```
[number3,ratio3] = symerr(x,aColumnVector)
```

```
number3 = 1×4

     1     0     2     0


ratio3 = 1×4

    0.3333         0    0.6667         0
```

Use Alternative Type of Comparison

You can specify alternative comparison methods used by `symerr`. In this example, you use a flag 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')
```

```
number = 
       3


ratio = 
       3/4


loc = 2×2

       0              1       
       1              1       
```

See Also

alignsignals | biterr | finddelay

Introduced before R2006a

syndtable

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` | `gfcosets` | `hammgen`

Topics

“Block Codes”

Introduced before R2006a

testconsole.Results

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`

Introduced in R2009b

tpcdec

Turbo product code (TPC) decoder

Syntax

```
decoded = tpcdec(llr,N,K)
decoded = tpcdec(llr,N,K,S)
decoded = tpcdec(llr,N,K,S,iternum)
```

Description

`decoded = tpcdec(llr,N,K)` performs 2-D TPC decoding on input log likelihood ratios, `llr`, using two linear block codes specified by codeword length `N` and message length `K`. For a description of 2-D TPC decoding, see "Algorithms" on page 1-981.

`decoded = tpcdec(llr,N,K,S)` performs 2-D TPC decoding on the shortened `llr` using a 2-D TPC decoder specified by codeword length (`N–K+S`) and message length `S`.

`decoded = tpcdec(llr,N,K,S,iternum)` performs 2-D TPC decoding for `iternum` iterations. To use `iternum` with full-length messages, specify `S` as empty, [ ].

Examples

Decode Using Full-Length TPC Codes

Decode an approximate log-likelihood ratio output signal from 16-QAM demodulation.

Begin by encoding a random bit vector using 2-D turbo product coding (TPC) with extended Hamming and BCH codes.

Specify the (N,K) code pairs to use for TPC encoding.

```
N = [32;16];
K = [21;11];
```

Generate a column vector of random message bits and TPC-encode the message. The desired length for the message bits is the product of the elements in K.

```
msg = randi([0 1],prod(K),1);
code = tpcenc(msg,N,K);
```

Apply 16-QAM modulation. Add AWGN to the signal. Demodulate the signal, outputting approximate LLRs.

```
M = 16;
snr = 10;

txsig = qammod(code,M,'InputType','bit', ...
    'UnitAveragePower',true);

rxsig = awgn(txsig,snr,'measured');

llr = qamdemod(rxsig,M,'OutputType','approxllr', ...
    'UnitAveragePower',true,'NoiseVariance',10.^(-snr/10));
```

Perform TPC decoding using three iterations. Because the demodulator output is negative bipolar mapped and TPC decoder expects positive bipolar mapped input, the demodulated signal output must be negated at the decoder input. Check the number of bit errors in the decoded signal.

```
iterations = 3;
decoded = tpcdec(-llr,N,K,[],iterations);

numerr = biterr(msg,decoded)
```

```
numerr = 0
```

Decode Using Shortened TPC Codes

Decode a shortened TPC code. Apply QPSK modulation and output the approximate log-likelihood ratio signal obtained from QPSK demodulation.

Begin by encoding a random bit vector using 2-D turbo product coding (TPC) with extended Hamming and BCH codes.

Specify (N,K) code pairs and S codes for TPC encoding.

```
N = [32;32];
K = [21;26];
S = [19;24];
```

Generate a column vector of random message bits and TPC-encode the message. The desired length for the shortened message bits is the product of the elements in S.

```
msg = randi([0 1],prod(S),1);
code = tpcenc(msg,N,K,S);
```

Apply QPSK modulation. Add AWGN to the signal. Demodulate the signal and output approximate LLRs.

```
M = 4;
snr = 3;

txsig = qammod(code,M,'InputType','bit', ...
    'UnitAveragePower',true);

rxsig = awgn(txsig,snr,'measured');

llr = qamdemod(rxsig,M,'OutputType','approxllr', ...
    'UnitAveragePower',true,'NoiseVariance',10.^(-snr/10));
```

Perform TPC decoding using two iterations. Because the demodulator output is negative bipolar mapped and TPC decoder expects positive bipolar mapped input, the demodulated signal output must be negated at the decoder input. Check the bit error rate of the decoded signal.

```
iterations = 2;
decoded = tpcdec(-llr,N,K,S,iterations);

[~,ber] = biterr(msg,decoded)
```

```
ber = 0.0066
```

Input Arguments

`llr` — Log likelihood ratios

column vector

Log likelihood ratios, specified as a column vector.

- For full-length codes, the length of the input column vector is the product of the elements in N.
- For shortened codes, the length of the input column vector is the product of the elements in (N–K+S).

Data Types: double | single

N — Codeword length

two-element integer column vector

Codeword length, specified as a two-element integer column vector, [N_R; N_C]. N_R represents the number of rows in the product code matrix. N_C represents the number of columns in the product code matrix. For more information about N_R and N_C, see “Algorithms” on page 1-981. For a list of valid ($N(i)$,$K(i)$) code pairs, see “Definitions” on page 1-979.

Data Types: double

K — Message length

two-element integer column vector

Message length, specified as a two-element integer column vector, [K_R; K_C]. For a full-length message, the input column vector containing the input LLRs is arranged into an K_R-by-K_C matrix. K_R represents the number of rows in the message matrix. K_C represents the number of columns in the message matrix. For more information about K_R and K_C, see “Algorithms” on page 1-981. For a list of valid ($N(i)$,$K(i)$) code pairs, see “Definitions” on page 1-979.

Data Types: double

S — Shortened message length

two-element integer column vector

Shortened message length, specified as a two-element integer column vector, [S_R; S_C]. For a shortened message, the input column vector containing the input LLRs is arranged into an S_R-by-S_C matrix.S_R represents the number of rows in the matrix. S_C represents the number of columns in the matrix. For more information about S_R and S_C, see “Algorithms” on page 1-981.

When you specify this parameter, specify N and K vectors for the full-length TPC codes that are shortened to ($N(i)$ – $K(i)$ + $S(i)$, $S(i)$) codes.

Data Types: double

`iternum` — Number of decoding iterations

4 (default) | positive integer

Number of decoding iterations, specified as a positive integer.

Data Types: `double`

Output Arguments

decoded — TPC decoded message

column vector

TPC decoded message, returned as a column vector.

- For full-length codes, the length of the returned column vector is the product of the elements in `K`.
- For shortened codes, the length of the returned column vector is the product of the elements in `S`.

Definitions

Component Codes

This table lists the supported component codes. $N(i)$,$K(i)$ represent the elements of the two-element column vector parameters, `N` and `K`, that specify the individual the code pairs (N_R, K_R) and (N_C, K_C). Any two pairs of component codes listed in the table can form a 2-D TPC code. The last column in the table lists the error-correction capability for each code pair.

Code type	**Component Code** ($N(i)$,$K(i)$)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1

	(15,11)	1
	(7,4)	1
Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2
	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-
	(3,2)	-

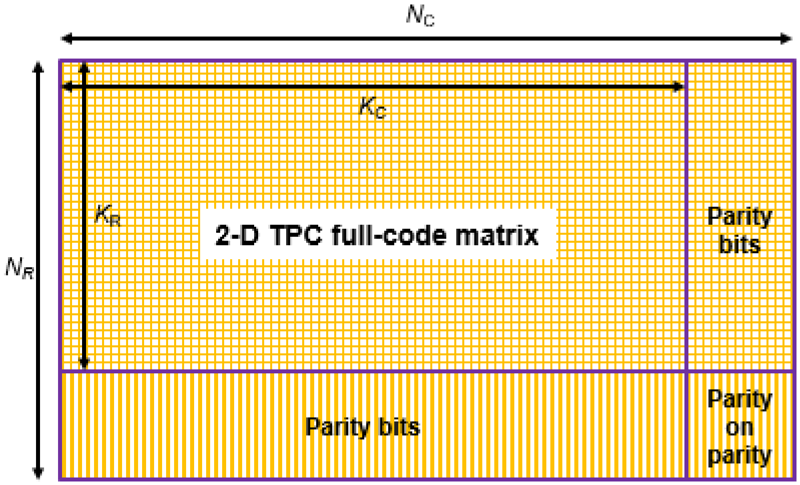
Algorithms

Turbo product codes (TPC) are a form of concatenated codes used as forward error correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. This function implements an iterative soft input, soft output 2-D product code decoding, as described in [2], using two “Linear Block Codes”. The function expects the soft bit log likelihood ratios (LLRs) obtained from digital demodulation as the input signal.

Note The TPC decoder expects a positive bipolar mapped input, specifically –1 mapped to 0 and +1 mapped to 1. The output from demodulators in the Communications System Toolbox is negative bipolar mapping, specifically +1 mapped to 0 and –1 mapped to +1. Therefore, the LLR output from demodulators must be negated to provide the positive bipolar mapped input expected by the TPC decoder.

TPC Decoding Full-Length Messages

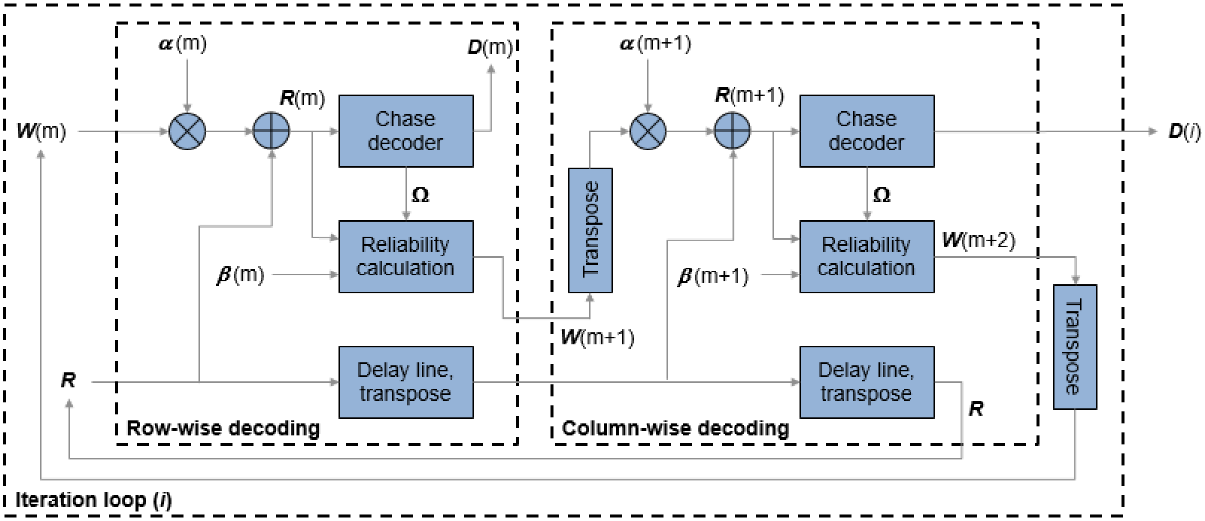
The 2-D TPC decoding of the full-length input message uses the specified two-element column vector parameters, `N` and `K`. Row-wise decoding uses the (N_C, K_C) code pair and column-wise decoding uses the (N_R, K_R) code pair. The input vector length must be N_R × N_C. To perform the 2-D TPC decoding, the column vector of the input LLRs, composed of the message and parity bits, is arranged into an N_R-by-N_C matrix.



The TPC decoder achieves near-optimum decoding of product codes using Chase decoding and the Pyndiah algorithm to perform iterative soft input, soft output decoding. Chase decoding forms a set of possible codewords for each row or column. the Pyndiah algorithm calculates soft information required for the next decoding step. For a detailed description, see [1] and [2].

Iterative Soft Input, Soft Output Decoder

The iterative soft input, soft output decoding, as shown in the block diagram, carries out two decoding steps for each iteration.



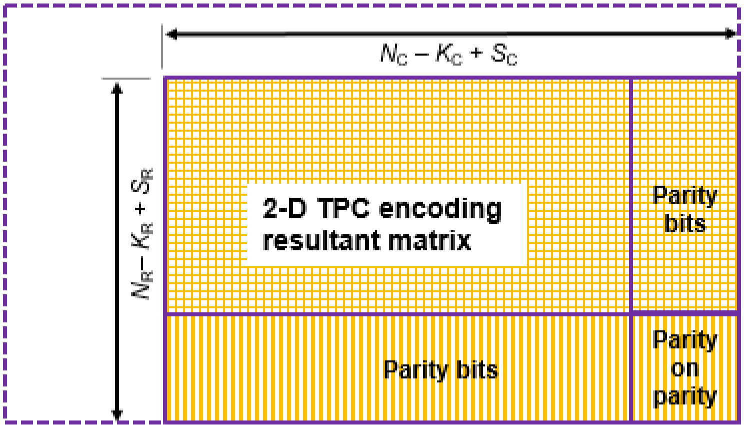
The soft inputs for decoding are $\boldsymbol{R}(m) = \boldsymbol{R} + \alpha(m)\boldsymbol{W}(m)$.

- Iteration loop counter i increments from $i = 1$ to the specified number of iterations.
- $m = 2i - 1$ is the decoding step index.
- $\boldsymbol{R}$ is the received LLR matrix.
- $\boldsymbol{R}(m)$ is the soft input for the mth decoding step.
- $\boldsymbol{W}(m)$ is the input extrinsic information for the mth decoding step.
- $\alpha(m)$ = [0,0.2,0.3,0.5,0.7,0.9,1,1, ...], where α is a weighting factor applied based on the decoding step index. For higher decoding steps, $\alpha = 1$.

- $\beta(m)$ = [0.2,0.4,0.6,0.8,1,1, ...], where β is a reliability factor applied based on the decoding step index. For higher decoding steps, β = 1.
- ***D*** contains the decoded message bits. After iterating through the specified number of iterations, the output message bits are formed from ***D*** by mapping –1 to 0 and +1 to 1, then reshaping the message block into a column vector.

TPC Decoding Shortened Messages

The 2-D TPC decoding of the shortened input message uses the specified two-element column vector parameters: N, K, and S. Row-wise decoding uses the (N_C – K_C + S_C, S_C) code pair and column-wise decoding uses the (N_R – K_R + S_R, S_R) code pair. The input vector length must be (N_R – K_R + S_R) × (N_C– K_C + S_C). To perform the 2-D TPC decoding of shortened messages, the column vector of the input LLRs, composed of the shortened message and parity bits, is arranged into an (N_R – K_R + S_R)-by-(N_C – K_C + S_C) matrix.



The TPC decoder processes the received shortened message LLRs similar to full length codes, with these exceptions:

- The shortened bit positions in the received codeword are set to –1.
- The Chase algorithm does not consider the shortened bit positions while choosing the least reliable bits.

References

[1] Chase, D. "Class of Algorithms for Decoding Block Codes with Channel Measurement Information." *IEEE Transactions on Information Theory*, Volume 18, Number 1, January 1972, pp. 170–182.

[2] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Volume 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- TPC parameters N, K, and S must be constant values. If the value used for each of these parameters does not change, then you can assign them by expression or variable.

See Also

Functions
bchdec | tpcenc

System Objects
comm.BCHDecoder

Introduced in R2018a

tpcenc

Turbo product code (TPC) encoder

Syntax

```
code = tpcenc(msg,N,K)
code = tpcenc(msg,N,K,S)
```

Description

`code = tpcenc(msg,N,K)` performs 2-D TPC encoding of the input message, `msg`, using two linear block codes specified by codeword length `N` and message length `K`. For a description of 2-D TPC encoding, see “Algorithms” on page 1-990.

`code = tpcenc(msg,N,K,S)` performs 2-D TPC encoding on the shortened input message of length `S`, using a 2-D TPC encoder specified by codeword length (`N–K+S`) and message length `S`.

Examples

Encode Using Full-Length TPC Codes

Encode a random bit vector using 2-D turbo product coding (TPC) with extended Hamming and BCH codes.

Specify (N,K) code pairs for TPC encoding.

```
N = [32;64];
K = [21;57];
```

Generate a column vector of random message bits. The desired length for the message bits is the product of elements in `K`.

```
msg = randi([0 1],prod(K),1);
```

TPC-encode the message.

```
code = tpcenc(msg,N,K);
```

Verify that the length of the encoded codeword is the product of elements in `N`.

```
size(code)
```

```
ans = 1×2

        2048           1

```

```
prod(N)
```

```
ans = 2048
```

Encode Shortened Message Using Turbo Product Coding

Encode a random bit vector using 2-D turbo product coding (TPC), applying message shortening.

Specify (N,K) code pairs and S codes for TPC encoding.

```
N = [32;64];
K = [21;57];
S = [19;24];
```

Generate a column vector of random message bits. The desired length for the shortened message bits is the product of the elements in `S`.

```
msg = randi([0 1],prod(S),1);
```

TPC-encode the shortened message.

```
code = tpcenc(msg,N,K,S);
```

Verify that the length of the encoded codeword is the product of elements in (`N-K+S`).

```
size(code)
```

```
ans = 1×2
```

```
   930     1
```

```
prod(N-K+S)
```

```
ans = 930
```

Input Arguments

`msg` — Input message bits to encode
column vector

Input message bits to encode, specified as a column vector.

- For a full-length input messages, the length of the column vector must be the product of the elements in K.
- For a shortened input messages, the length of the column vector must be the product of the elements in S.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`N` — Codeword length
two-element integer column vector

Codeword length, specified as a two-element integer column vector, [N_R; N_C]. N_R represents the number of rows in the product code matrix. N_C represents the number of columns in the product code matrix. For more information about N_R and N_C, see "Algorithms" on page 1-990. For a list of valid ($N(i)$,$K(i)$) code pairs, see "Component Codes" on page 1-988.

Data Types: `double`

`K` — Message length
two-element integer column vector

Message length, specified as a two-element integer column vector, [K_R; K_C]. For a full-length message, the input column vector containing the message bits to enocde is arranged into an K_R-by-K_C matrix. K_R represents the number of rows in the message matrix. K_C represents the number of columns in the message matrix. For more information about K_R and K_C, see “Algorithms” on page 1-990. For a list of valid ($N(i)$,$K(i)$) code pairs, see “Component Codes” on page 1-988.

Data Types: `double`

S — Shortened message length
two-element integer column vector

Shortened message length, specified as a two-element integer column vector, [S_R; S_C]. For a shortened message, the input column vector containing the message bits to enocde is arranged into an S_R-by-S_C matrix. S_R represents the number of rows in the matrix. S_C represents the number of columns in the matrix. For more information about S_R and S_C, see “Algorithms” on page 1-990.

When you specify this parameter, specify N and K vectors for the full-length TPC codes that are shortened to ($N(i)$–$K(i+S(i)$, $S(i)$)) codes.

Data Types: `double`

Output Arguments

code — TPC-encoded message
column vector

TPC-encoded message, returned as a column vector.

- For full-length input messages, the length of the returned column vector is the product of the elements in `N`.
- For shortened input messages, the length of the returned column vector is the product of the elements in (`N–K+S`).

Definitions

Component Codes

This table lists the supported component codes. $N(i)$,$K(i)$ represent the elements of the two-element column vector parameters, `N` and `K`, that specify the individual the code pairs (N_R, K_R) and (N_C, K_C). Any two pairs of component codes listed in the table can form a 2-D

TPC code. The last column in the table lists the error-correction capability for each code pair.

Code type	**Component Code** (*N(i)*,*K(i)*)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1
	(15,11)	1
	(7,4)	1
Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2
	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-

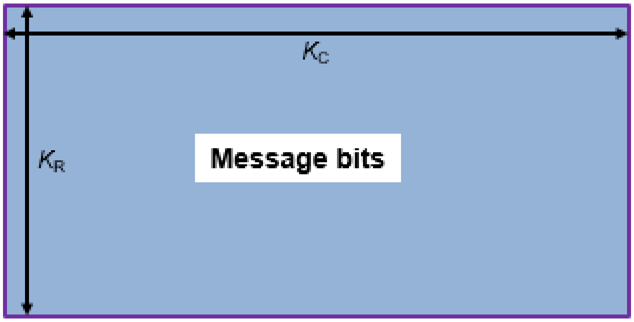
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-
	(3,2)	-

Algorithms

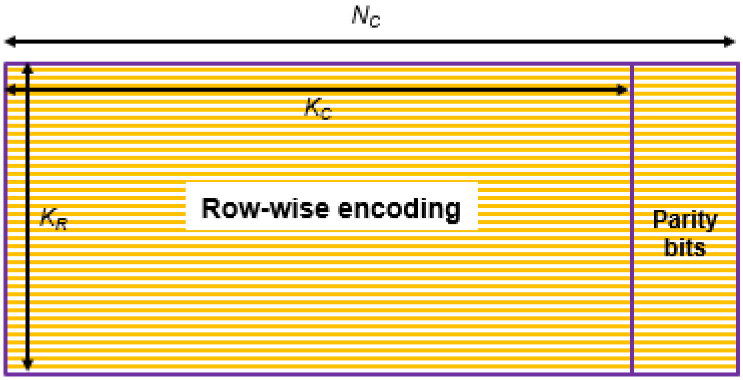
Turbo product codes (TPC) are a form of concatenated codes used as forward error-correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. This function implements 2-D product code encoding, as described in [1], using two “Linear Block Codes”.

Construction of Product Codes

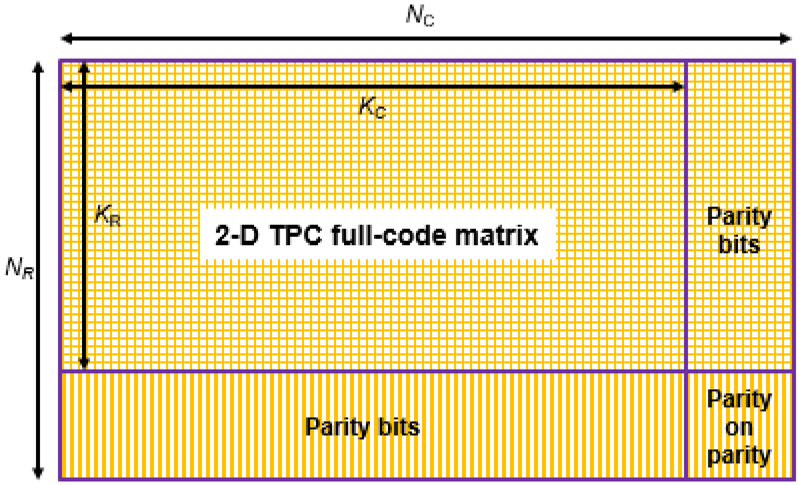
The full-length input message is encoded using specified 2-D TPC code pairs. Row-wise encoding uses the (N_C, K_C) code pair and column-wise encoding uses the (N_R, K_R) code pair. The input column vector containing the input message bits is arranged into a K_R-by-K_C matrix. The input vector length must be $K_R \times K_C$.



Row-wise encoding uses an (N_C, K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in an K_R-by-N_C matrix that includes parity bits added to each row.



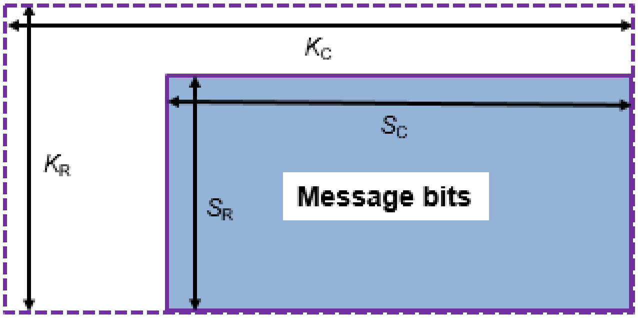
The next processing operation performs column-wise encoding using an (N_R, K_R) systematic linear block encoder on each of the N_C columns. After the 2-D TPC encoding, the initial K_R-by-K_C matrix results in a N_R-by-N_C matrix that includes parity bits added to each row and column.



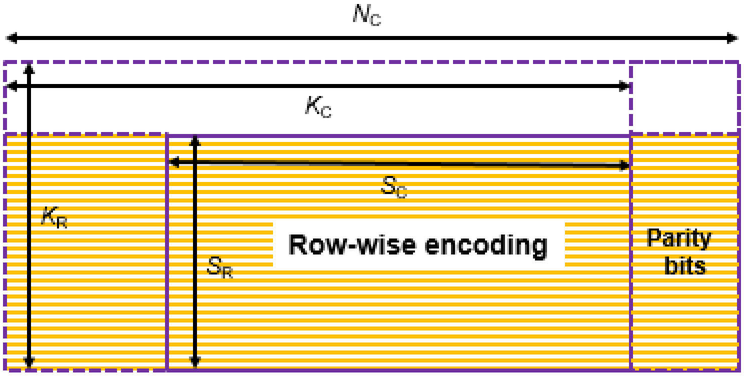
The 2-D TPC full-code matrix is reshaped into a column vector and returned. The length of the TPC-encoded output is $N_R \times N_C$.

Construction of Product Codes with Shortening

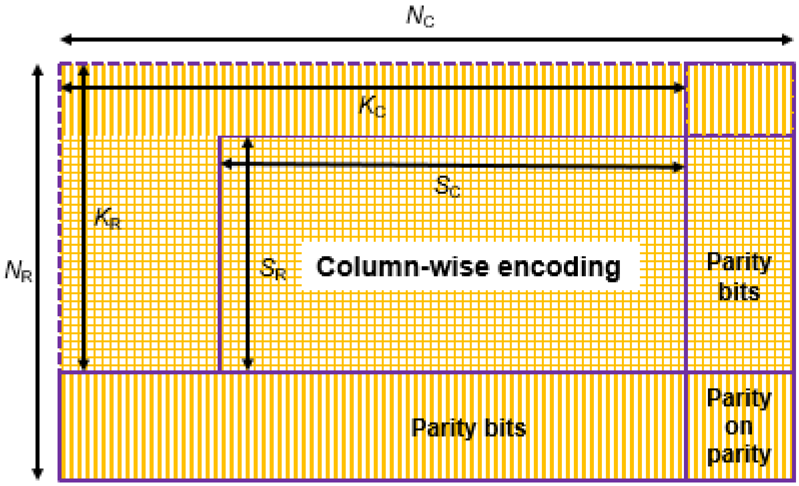
The shortened input message is encoded using specified 2-D TPC code pairs. Row-wise encoding uses an (N_C, K_C) code pair and column-wise encoding uses an (N_R, K_R) code pair. The input vector length must be S_R × S_C. The input column vector, containing the shortened message bits, is arranged into a K_R-by-K_C matrix. The shortened message bits matrix prepends two dimensions by padding the beginning of the message matrix with zeros.



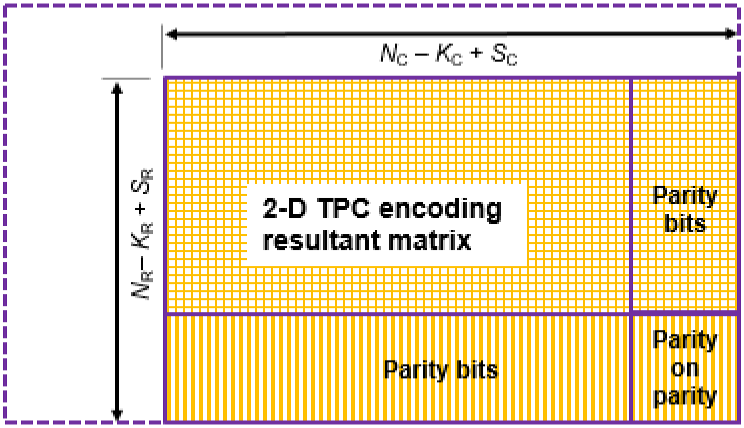
Row-wise encoding uses an (N_C, K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in a K_R-by-N_C matrix that includes parity bits added to each row.



The next processing operation performs column-wise encoding using an (N_R, K_R) systematic linear block encoder over N_C columns. After the 2-D TPC encoding, the zero-padded bits are excluded from the output to yield a (N_R-K_R+S_R)-by-(N_C-K_C+S_C) matrix. This matrix includes parity bits added to each row and column.



The 2-D TPC shortened-code matrix is reshaped into a column vector and returned.



For a shortened input message, the length of the TPC-encoded message is (N_R–K_R+S_R) × (N_C–K_C+S_C).

References

[1] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Volume 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- TPC parameters N, K, and S must be constant values. If the value used for each of these parameters does not change, then you can assign them by expression or variable.

See Also

Functions
bchenc | tpcdec

System Objects
comm.BCHEncoder

Introduced in R2018a

varlms

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'`

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.
`InitStep`	Initial value of step size when the algorithm starts
`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

$\mu_0 = \mu + (\texttt{IncStep})\ \mathrm{Re}(gg_{\mathrm{prev}})$

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

(LeakageFactor) w + 2 μ g^*

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

cma | dfe | equalize | lineareq | lms | normlms | rls | signlms

Topics

“Equalization”

Introduced before R2006a

vec2mat

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

Convert Vector to Matrix Using vec2mat

Create a five-element vector.

```
vec = [1,2,3,4,5];
```

Convert the vector to matrices with two, three, and four columns.

```
twoColumnMatrix = vec2mat(vec,2)
```

```
twoColumnMatrix = 3×2

     1     2
     3     4
     5     0
```

```
threeColumnMatrix = vec2mat(vec,3)
```

```
threeColumnMatrix = 2×3

     1     2     3
     4     5     0
```

```
fourColumnMatrix = vec2mat(vec,4)
```

```
fourColumnMatrix = 2×4

     1     2     3     4
     5     0     0     0
```

Specify Nonzero Padding

Create a five-element vector.

```
vec = 1:5;
```

Specify a padding value for the matrix output by `vec2Mat`. Convert the vector to a four-column matrix using the nonzero padding value.

```
paddingValue = NaN;
mat = vec2mat(vec,4,paddingValue)
```

```
mat = 2×4

     1     2     3     4
     5   NaN   NaN   NaN
```

You can also specify the padding value as a vector or matrix. `vec2mat` pads the output matrix with values taken from `paddingValue` in order.

```
paddingValue = [10,8,6;9,7,5]
```

```
paddingValue = 2×3

    10     8     6
     9     7     5
```

```
mat2 = vec2mat(vec,3,paddingValue)
```

```
mat2 = 2×3

     1     2     3
     4     5    10
```

```
mat3 = vec2mat(vec,4,paddingValue)
```

```
mat3 = 2×4

     1     2     3     4
     5    10     9     8
```

Return Number of Padded Elements

You can optionally return the number of elements padded by `vec2mat`.

Create a five-element vector and a matrix of padding values.

```
vec = [1;2;3;4;5];
padding = [2,4;6,4];
```

Convert `vec` to matrices with two, three, and four columns. Specify nonzero padding and return the number padded in each scenario.

```
[mat2,numPadded2] = vec2mat(vec,2,padding)
```

```
mat2 = 3×2

     1     2
     3     4
     5     2


numPadded2 = 1
```

```
[mat3,numPadded3] = vec2mat(vec,3,padding)
```

```
mat3 = 2×3

     1     2     3
     4     5     2


numPadded3 = 1
```

```
[mat4,numPadded4] = vec2mat(vec,4,padding)
```

```
mat4 = 2×4

     1     2     3     4
     5     2     6     4


numPadded4 = 3
```

See Also

`reshape`

Introduced before R2006a

vitdec

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.

`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 on page 1-1004 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.

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     0     1
     2     3     3     2     2     3     3     2     2     3     3     2     2     3
     0     1     1     0     0     1     1     0     0     1     1     0     0     1
     2     3     3     2     2     3     3     2     2     3     3     2     2     3

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     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     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 (0th 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

Create Convolutional Code

This example shows how to create a convolutional code using the `convenc` function and how to decode it using `vitdec`.

Encoding

Define a trellis.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]);
```

Encode a vector of ones.

```
x = ones(100,1);
code = convenc(x,t);
```

Decoding

Define a trellis.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]);
```

Encode a vector of ones.

```
code = convenc(ones(100,1),t);
```

Set the traceback length for decoding and decode using `vitdec`.

```
tb = 2;
decoded = vitdec(code,t,tb,'trunc','hard');
```

Verify that the decoded data is a vector of 100 ones.

```
isequal(decoded,ones(100,1))
```

```
ans = logical
   1
```

Estimate BER for Rate 2/3 Convolutional Code

This example performs a bit error rate simulation for a link that uses 16-QAM modulation a rate 2/3 convolutional code.

Set the modulation order, and compute the number of bits per symbol.

```
M = 16;
k = log2(M);
```

Generate random binary data.

```
dataIn = randi([0 1],100000,1);
```

Define a convolutional coding trellis for a rate 2/3 code.

```
tPoly = poly2trellis([5 4],[23 35 0; 0 5 13]);
codeRate = 2/3;
```

Convolutionally encode the input data.

```
codeword = convenc(dataIn,tPoly);
```

Reshape the encoded column vector into a matrix having `k` columns. Then, convert the binary matrix into an integer column vector.

```
codewordMat = reshape(codeword,length(codeword)/k,k);
txSym = bi2de(codewordMat);
```

Apply 16-QAM modulation to the encoded symbols.

```
txSig = qammod(txSym,M);
```

Convert a 10 dB Eb/No to an equivalent signal-to-noise ratio. Pass the signal through an AWGN channel.

```
EbNo = 10;
snr = EbNo + 10*log10(k*codeRate);
rxSig = awgn(txSig,snr,'measured');
```

Demodulate the received signal.

```
demodSig = qamdemod(rxSig,M);
```

Convert the output of the demodulator into a binary column vector.

```
demodSigMat = de2bi(demodSig,k);
demodSigBinary = demodSigMat(:);
```

Set the traceback depth of the Viterbi decoder.

```
traceBack = 16;
```

Decode the binary demodulated signal by using a Viterbi decoder operating in a continuous termination mode.

```
dataOut = vitdec(demodSigBinary,tPoly,traceBack,'cont','hard');
```

Calculate the delay through the decoder, and compute the bit error statistics.

```
decDelay = 2*traceBack;
[numErrors,ber] = biterr(dataIn(1:end-decDelay),dataOut(decDelay+1:end))
```

```
numErrors = 26
```

```
ber = 2.6008e-04
```

Compare the BER with the uncoded BER.

```
berUncoded = berawgn(EbNo,'qam',M);
berUncoded/ber
```

```
ans = 6.7446
```

The convolutional code reduces the BER by approximately a factor of 4.

Estimate BER for Hard and Soft Decision Viterbi Decoding

Estimate bit error rate (BER) performance for hard-decision and soft-decision Viterbi decoders in AWGN. Compare the performance to that of an uncoded 64-QAM link.

Set the simulation parameters.

```
clear; close all
rng default
M = 64;                 % Modulation order
k = log2(M);            % Bits per symbol
EbNoVec = (4:10)';       % Eb/No values (dB)
numSymPerFrame = 1000;   % Number of QAM symbols per frame
```

Initialize the BER results vectors.

```
berEstSoft = zeros(size(EbNoVec));
berEstHard = zeros(size(EbNoVec));
```

Set the trellis structure and traceback length for a rate 1/2, constraint length 7, convolutional code.

```
trellis = poly2trellis(7,[171 133]);
tbl = 32;
rate = 1/2;
```

The main processing loops performs these steps:

- Generate binary data.
- Convolutionally encode the data.
- Apply QAM modulation to the data symbols. Specify unit average power for the transmitted signal.
- Pass the modulated signal through an AWGN channel.
- Demodulate the received signal using hard decision and approximate LLR methods. Specify unit average power for the received signal.
- Viterbi decode the signals using hard and unquantized methods.
- Calculate the number of bit errors.

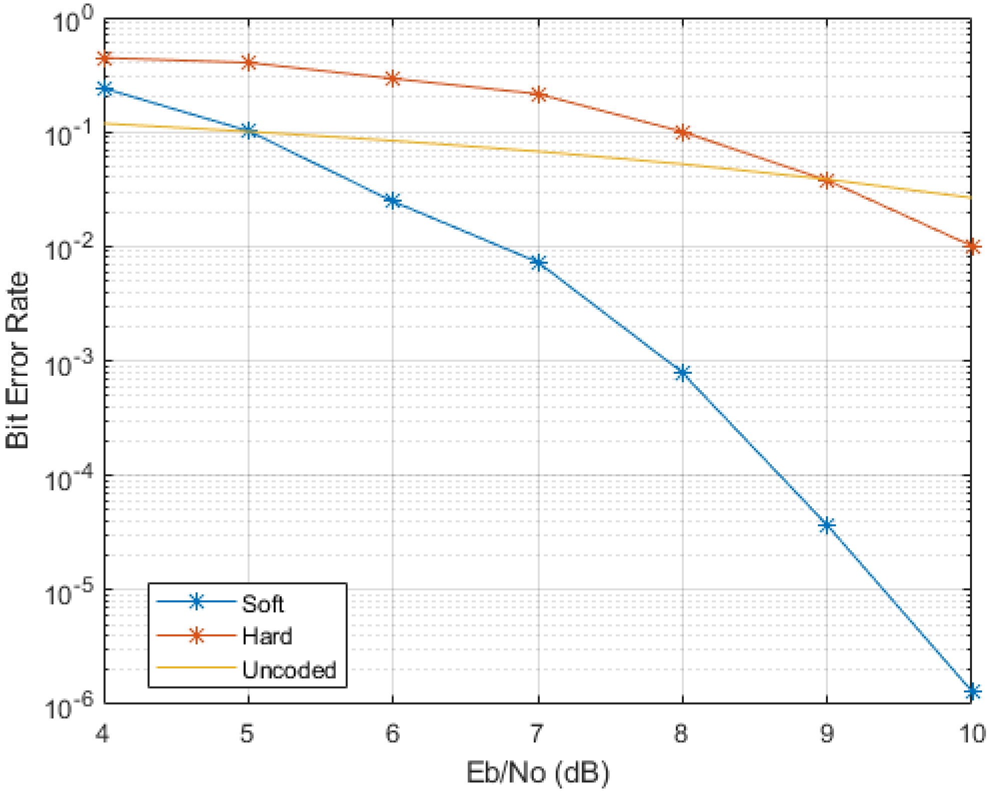
The `while` loop continues to process data until either 100 errors are encountered or 1e7 bits are transmitted.

```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k*rate);
    % Noise variance calculation for unity average signal power.
    noiseVar = 10.^(-snrdB/10);
    % Reset the error and bit counters
    [numErrsSoft,numErrsHard,numBits] = deal(0);
    while numErrsSoft < 100 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame*k,1);

        % Convolutionally encode the data
        dataEnc = convenc(dataIn,trellis);

        % QAM modulate
        txSig = qammod(dataEnc,M,'InputType','bit','UnitAveragePower',true);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal using hard decision (bit) and
        % soft decision (approximate LLR) approaches.
        rxDataHard = qamdemod(rxSig,M,'OutputType','bit','UnitAveragePower',true);
        rxDataSoft = qamdemod(rxSig,M,'OutputType','approxllr', ...
            'UnitAveragePower',true,'NoiseVariance',noiseVar);

        % Viterbi decode the demodulated data
        dataHard = vitdec(rxDataHard,trellis,tbl,'cont','hard');
        dataSoft = vitdec(rxDataSoft,trellis,tbl,'cont','unquant');

        % Calculate the number of bit errors in the frame. Adjust for the
        % decoding delay, which is equal to the traceback depth.
        numErrsInFrameHard = biterr(dataIn(1:end-tbl),dataHard(tbl+1:end));
        numErrsInFrameSoft = biterr(dataIn(1:end-tbl),dataSoft(tbl+1:end));

        % Increment the error and bit counters
        numErrsHard = numErrsHard + numErrsInFrameHard;
        numErrsSoft = numErrsSoft + numErrsInFrameSoft;
        numBits = numBits + numSymPerFrame*k;

    end

    % Estimate the BER for both methods
    berEstSoft(n) = numErrsSoft/numBits;
    berEstHard(n) = numErrsHard/numBits;
end
```

Plot the estimated hard and soft BER data. Plot the theoretical performance for an uncoded 64-QAM channel.

```
semilogy(EbNoVec,[berEstSoft berEstHard],'-*')
hold on
```

```
semilogy(EbNoVec,berawgn(EbNoVec,'qam',M))
legend('Soft','Hard','Uncoded','location','best')
grid
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



As expected, the soft decision decoding produces the best results.

Limitations

In order to improve performance of C/C++ code generated by MATLAB, integrity and responsiveness checks should be disabled before running the `codegen` function. See

“MATLAB Code Design Considerations for Code Generation” (Simulink) for more information.

For example, given the following function:

```
function y = vitdec_hard(x,t,tb)
%# codegen

y = vitdec(x,t,tb,'trunc','hard');
```

Execute these commands for optimal performance.

```
cf = coder.config;
cf.IntegrityChecks = false;
cf.ResponsivenessChecks = false;
codegen('vitdec_hard','-args',{x,coder.Constant(t),tb})
```

The coded data, `x`, the trellis structure, `t`, and the traceback length, `tb`, must be defined in the base workspace.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convenc` | `istrellis` | `poly2trellis`

Topics

“Convolutional Codes”

Introduced before R2006a

wgn

Generate white Gaussian noise

Syntax

```
y = wgn(m,n,power)
y = wgn(m,n,power,imp)
y = wgn(m,n,power,imp,streamhandle)
y = wgn(m,n,power,imp,seed)
y = wgn( ___ ,powertype)
y = wgn( ___ ,outputtype)
```

Description

`y = wgn(m,n,power)` generates an `m`-by-`n` matrix of white Gaussian noise. `power` specifies the power of `y` in decibels relative to a watt. The default load impedance is 1 ohm.

`y = wgn(m,n,power,imp)` accepts an additional input, `imp`, that specifies the load impedance in ohms.

`y = wgn(m,n,power,imp,streamhandle)` accepts a random stream handle to generate the normal random samples by using `randn`, before generating the matrix of white Gaussian noise. For more information, see `RandStream`.

`y = wgn(m,n,power,imp,seed)` accepts a seed value for initializing the normal random number generator, `randn`, before generating the matrix of white Gaussian noise. If you want to generate repeatable noise samples, then either reset the random stream input before calling `wgn` or use the same seed input.

`y = wgn( ___ ,powertype)` accepts an additional input, `powertype`, that specifies the units of `power`. Choices for `powertype` are `'dBW'`, `'dBm'`, and `'linear'`.

`y = wgn( ___ ,outputtype)` accepts an additional input, `outputtype`, specified as `'real'` or `'complex'`. If `outputtype` is `'complex'`, then the real and imaginary parts of `y` each have a noise power of `power`/2.

Note The output of the wgn function is expressed in volts. For power calculations, a load of 1 ohm is assumed.

Examples

Generate White Gaussian Noise

Generate a 1000-element column vector containing real white Gaussian noise of power 0 dBW.

```
y1 = wgn(1000,1,0);
```

Confirm that the power is approximately 0 dBW, that is, 1 W.

```
var(y1)
```

```
ans = 0.9979
```

Generate a vector of complex white Gaussian noise having power -6 dBW.

```
y2 = wgn(1000,1,-6,'complex');
```

Confirm that the power is 0.25 W (-6 dBW).

```
var(y2)
```

```
ans = 0.2522
```

See Also

Functions

RandStream | awgn | randn

Topics

"Sources and Sinks"

Introduced before R2006a

winner2.AntennaArray

Create antenna array

Download Required: To use this function, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get Add-Ons” (MATLAB) and “Manage Your Add-Ons” (MATLAB).

Syntax

```
antArray = winner2.AntennaArray
antArray = winner2.AntennaArray(Name,Value)
```

Description

`antArray = winner2.AntennaArray` returns a structure representing an antenna array with one isotropic antenna element. Both the antenna array and the single element have no rotation and are located at the origin, [0;0;0].

`antArray = winner2.AntennaArray(Name,Value)` returns a structure representing an antenna array defined using one or more `Name,Value` pair arguments.

For more information, see “Antenna Array Model” on page 1-1023.

Examples

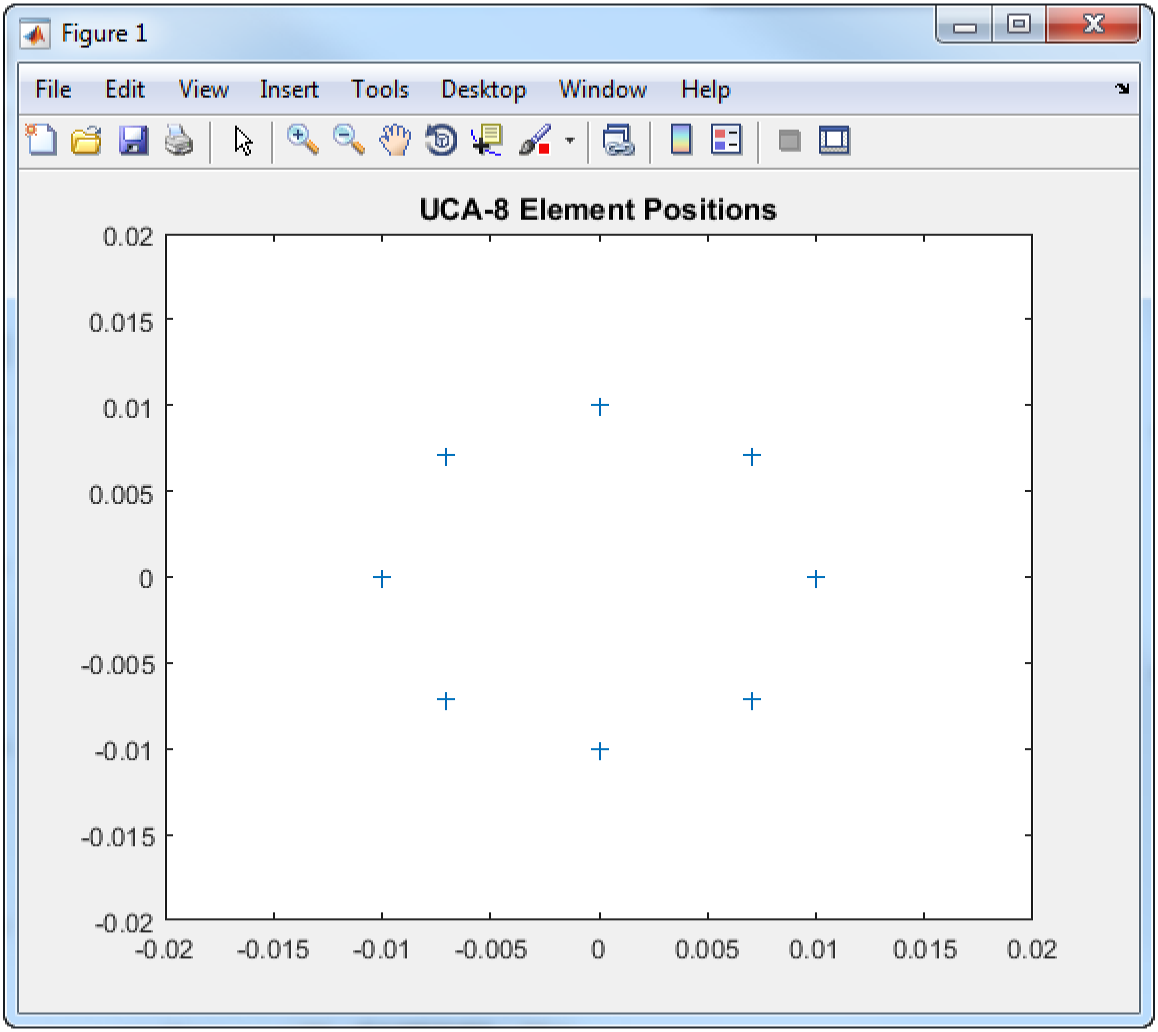
Create WINNER2 Eight Element Uniform Circular Array

Use the `winner2.AntennaArray` function to create an eight element uniform circular array (UCA-8) with a 1 cm radius.

```
UCA8 = winner2.AntennaArray('UCA',8,0.01);
```

Plot element positions

```
pos = {UCA8.Element(:).Pos};
plot(cellfun(@(x) x(1),pos),cellfun(@(x) x(2),pos),'+');
xlim([-0.02 0.02]);
ylim([-0.02 0.02]);
title('UCA-8 Element Positions');
```



Create WINNER2 Two Element Uniform Linear Array

Use the `winner2.AntennaArray` function to create a two element uniform linear array (ULA-2) with 50 cm spacing and the dipole elements slanted at +45 and -45 degrees.

```
az = -180:179; % 1-degree spacing
pattern = cat(1,shiftdim(winner2.dipole(az,45),-1), ...
    shiftdim(winner2.dipole(az,-45),-1));
ULA2 = winner2.AntennaArray('ULA',2,0.5, ...
    'FP-ECS',pattern,'Azimuth',az);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Pos',[1 0 0; 0 1 0],'Rot',[0 0 0; 0 pi() 0]` indicates the coordinates and rotation angles for two antenna elements.

Pos — Position of each antenna element
`0` (default) | column vector | matrix

Position of each antenna element, specified as the comma-separated pair consisting of `'Pos'` and a column vector or an N_E-by-3 matrix. The three columns represent the x-, y-, and z-coordinates in meters from the origin. N_E indicates the number of elements in the antenna array. The elements have no rotation. When there is more than one element, the `'Element'` field of `antArray` is a row vector of structures representing all the elements.

Example: `'Pos',[63.1 10.2 11.5; 62 11 12]` indicates the coordinates for two antenna elements.

Data Types: `double`

Rot — Rotation angle of each antenna element
`0` (default) | column vector | matrix | optional

Rotation angle of each antenna element, specified as the comma-separated pair consisting of `'Rot'` and a column vector or an N_E-by-3 matrix. The three columns represent the

Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians. N_E indicates the number of elements in the antenna array. `Rot` only applies when `Pos` is specified. If not specified with `Pos`, the rotation angle is `0`.

Example: `'Rot',[2 1.5 0; 0 pi() 0]` indicates the rotation angles for two antenna elements.

Data Types: `double`

UCA — Uniform circular antenna array
`N,1` (default) | `N,Rad`

Uniform circular antenna array, specified as the comma-separated pair consisting of `'UCA'` and `N,Rad`. In this argument, `N` indicates the number of elements (N_E) and `Rad` indicates the radius in meters. If `Rad` is not specified, the default radius is 1 meter.

Example: `'UCA',8,0.5` indicates an eight element uniform circular array with 0.5 meter radius.

Data Types: `double`

ULA — Uniform linear antenna array
`N,1/N` (default) | `N,Spacing`

Uniform linear antenna array, specified as the comma-separated pair consisting of `'ULA'` and `N,Spacing`. In this argument, `N` indicates the number of elements (N_E) and `Spacing` indicates the separation between adjacent elements in meters. If `Spacing` is not specified, the default separation is 1/`N` meters.

ULA elements are placed along *x-axis* with the center of the array at [0;0;0]. For an even number of elements, there is no antenna element at [0;0;0].

Example: `'ULA',3,0.25` indicates a three element uniform linear array with 0.25 meter spacing between adjacent elements.

Data Types: `double`

`FP-ECS` — Field pattern of element coordinate system
4-D array

Field pattern of element coordinate system, specified as the comma-separated pair consisting of `'FP-ECS'` and a P-by-2-by1-by-N_{AZ} array.

- The first dimension, P, can be either 1 or any number greater than or equal to the number of elements in the antenna array (N_E). When $P = 1$, the same pattern applies to all elements. When $P > N_E$, the first N_E rows apply.
- The second dimension, `2`, indicates that two polarizations characterize the field pattern. The first dimension in the field pattern stores vertical polarization, and the second one stores horizontal polarization.
- The third dimension, `1`, indicates that one elevation angle characterizes the field pattern.
- The fourth dimension, N_{AZ}, is the number of field pattern samples taken between –180 and 180 degrees. N_{AZ} equals the number of elements specified in `Azimuth` or when `Azimuth` is not present it equals the number of equidistant field pattern samples taken over azimuth angle.

Data Types: `double`

`FP-ACS` — Field pattern array coordinate system

4-D array

Field pattern array coordinate system, specified as the comma-separated pair consisting of `'FP-ACS'` and a P-by-2-by1-by-N_{AZ} array. Array format is the same as the `FP-ECS` syntax, except that the field pattern is specified in the array-coordinate-system (ACS).

- The first dimension, P, can be either 1 or any number greater than or equal to the number of elements in the antenna array (N_E). When $P = 1$, the same pattern applies to all elements. When $P > N_E$, the first N_E rows apply.
- The second dimension, `2`, indicates that two polarizations characterize the field pattern. The first dimension in the field pattern stores vertical polarization, and the second one stores horizontal polarization. Missing polarization dimensions of the field pattern are substituted with zeros.
- The third dimension, `1`, indicates that one elevation angle characterizes the field pattern.
- The fourth dimension, N_{AZ}, is the number of field pattern samples taken between –180 and 180 degrees. N_{AZ} equals the number of elements specified in `Azimuth` or when `Azimuth` is not present it equals the number of equidistant field pattern samples taken over azimuth angle.

Data Types: `double`

`Azimuth` — Azimuth angles for 'FP-ACS' or 'FP-ECS' field patterns

row vector

Azimuth angles for `FP-ACS` or `FP-ECS` field patterns in degrees , specified as the comma-separated pair consisting of `'Azimuth'` and an 1-by-N_{AZ} row vector. The values in the row vector indicate azimuth angles for elements in the field patterns.

Note `Azimuth` applies only when `FP-ACS` or `FP-ECS` are defined. If `Azimuth` is not specified, uniform spacing is used for elements in the field pattern.

Example: `'Azimuth',[0 10 20 90 180 270 340 350]`

Data Types: `double`

Output Arguments

`antArray` — Antenna array definition

structure

Antenna array definition, returned as a structure containing these fields.

`Name` — Antenna array name

character vector

Antenna array name, returned as a character vector.

`Pos` — Antenna array position

vector

Antenna array position, returned as a 3-by-1 vector, representing the x-, y-, and z-coordinates in meters from the origin.

`Rot` — Antenna array rotation

vector

Antenna array rotation, returned as a 3-by-1 vector, representing the Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians.

`Element` — Element definition

row vector of structures

Element definition, returned as a row vector of structures, with each structure representing one element and containing these fields.

`Pos` — Antenna array position
vector

Antenna array position, returned as a 3-by-1 vector, representing the x-, y-, and z-coordinates in meters from the origin.

`Rot` — Antenna array rotation
vector

Antenna array rotation, returned as a 3-by-1 vector, representing the Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians.

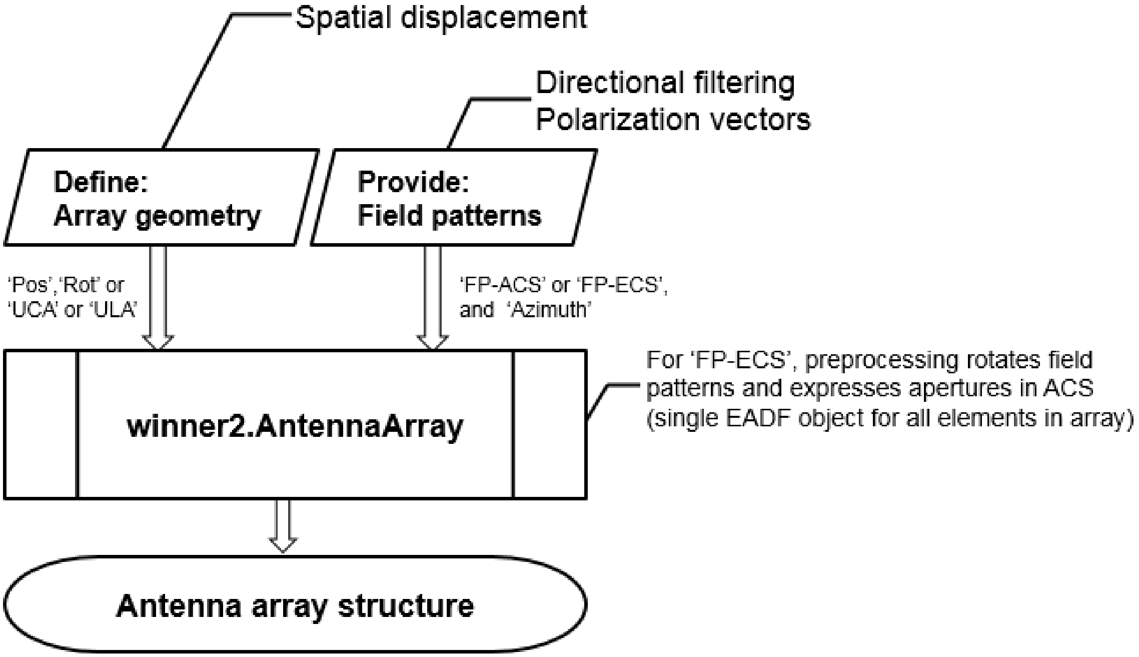
`Aperture` — Aperture definition
structure

Aperture definition, returned as a structure representing the antenna aperture.

Definitions

Antenna Array Model

To create an antenna array model, you must define the geometry of array elements (positions and rotation) and the element field patterns. The arguments provided to `winner2.AntennaArray` are always processed such that the array geometry is created first, and then the field patterns are assigned.



For a detailed description of the antenna array specification for the WINNER channel model, see WINNER II Channel Models [1], Section 4.1.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

`winner2.dipole` | `winner2.layoutparset`

Introduced in R2017a

winner2.dipole

Calculate field pattern of half-wavelength dipole

Download Required: To use this function, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get Add-Ons” (MATLAB) and “Manage Your Add-Ons” (MATLAB).

Syntax

```
pat = winner2.dipole(az)
pat = winner2.dipole(az,slant)
```

Description

`pat = winner2.dipole(az)` returns the azimuth field pattern of a 0-degree slanted dipole at the azimuth angles specified in `az`.

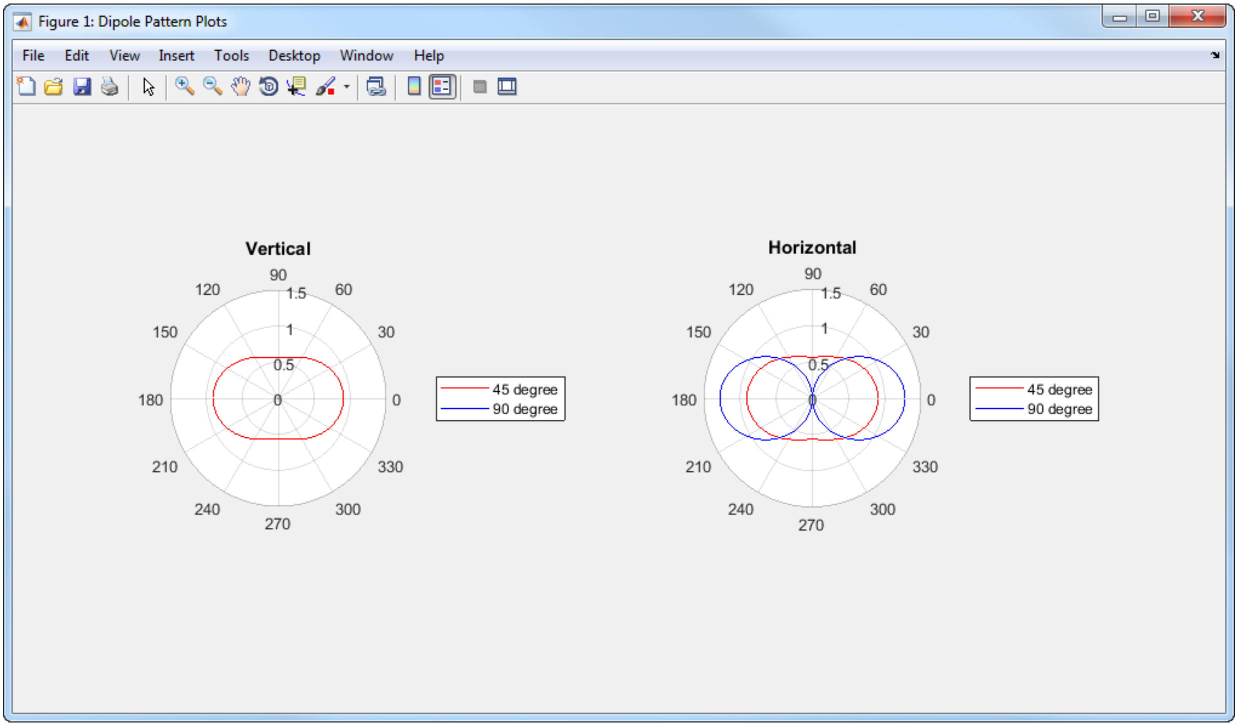
`pat = winner2.dipole(az,slant)` returns the azimuth field pattern of a slanted dipole at the azimuth angles specified in `az`.

Examples

Create 45 and 90 Degree Slanted Dipoles

```
az = -180:179;  % 1 degree spacing
pattern45 = squeeze(winner2.dipole(az, 45));
pattern90 = squeeze(winner2.dipole(az, 90));

fh = figure;
set(fh, 'Position', [100 100 1000 500]);
fh.Name = 'Dipole Pattern Plots';

subplot(1,2,1);
```

```
polarplot(az/180*pi, pattern45(1,:), 'r');
hold on;
polarplot(az/180*pi, pattern90(1,:), 'b');
rlim([0 1.5]);
legend('45 degree', '90 degree');
title('Vertical');

subplot(1,2,2);
polarplot(az/180*pi, pattern45(2,:), 'r');
hold on;
polarplot(az/180*pi, pattern90(2,:), 'b');
rlim([0 1.5]);
legend('45 degree','90 degree');
title('Horizontal');
```



Input Arguments

az — Azimuth angles

vector

Azimuth angles, specified as a vector indicating the azimuth angles to compute the field pattern gain. Units are in degrees.

Data Types: `double`

`slant` — Slant angle

scalar

Slant angle, specified as a scalar representing the counterclockwise angle seen from the front of the dipole. Units are in degrees.

Data Types: `double`

Output Arguments

pat — Field pattern

3-D array

Field pattern, returned as a 2-by-1-by-N_{AZ} array representing the vertical and horizontal field pattern, where N_{AZ} is the number of elements in the `az` input vector.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

`winner2.AntennaArray` | `winner2.layoutparset`

Introduced in R2017a

winner2.layoutparset

WINNER II layout parameter configuration

Download Required: To use this function, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get Add-Ons” (MATLAB) and “Manage Your Add-Ons” (MATLAB).

Syntax

```
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays)
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax)
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax,seed)
```

Description

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays)` returns a structure of randomly generated WINNER II network layout parameters given mobile station (MS) indices, base station (BS) indices, BS to MS links, and antenna array configurations.

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax)` additionally specifies the maximum layout range used when generating MS and BS positions.

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax,seed)` additionally specifies a seed value for repeatability. To assign `seed` when not assigning `rmax`, specify `rmax` as [ ].

Examples

Create Two MS to One BS WINNER 2 System Layout

Create a WINNER 2 system layout with two mobile stations (MS) connecting to the same base station (BS).

Define antenna arrays for one BS and two MS.

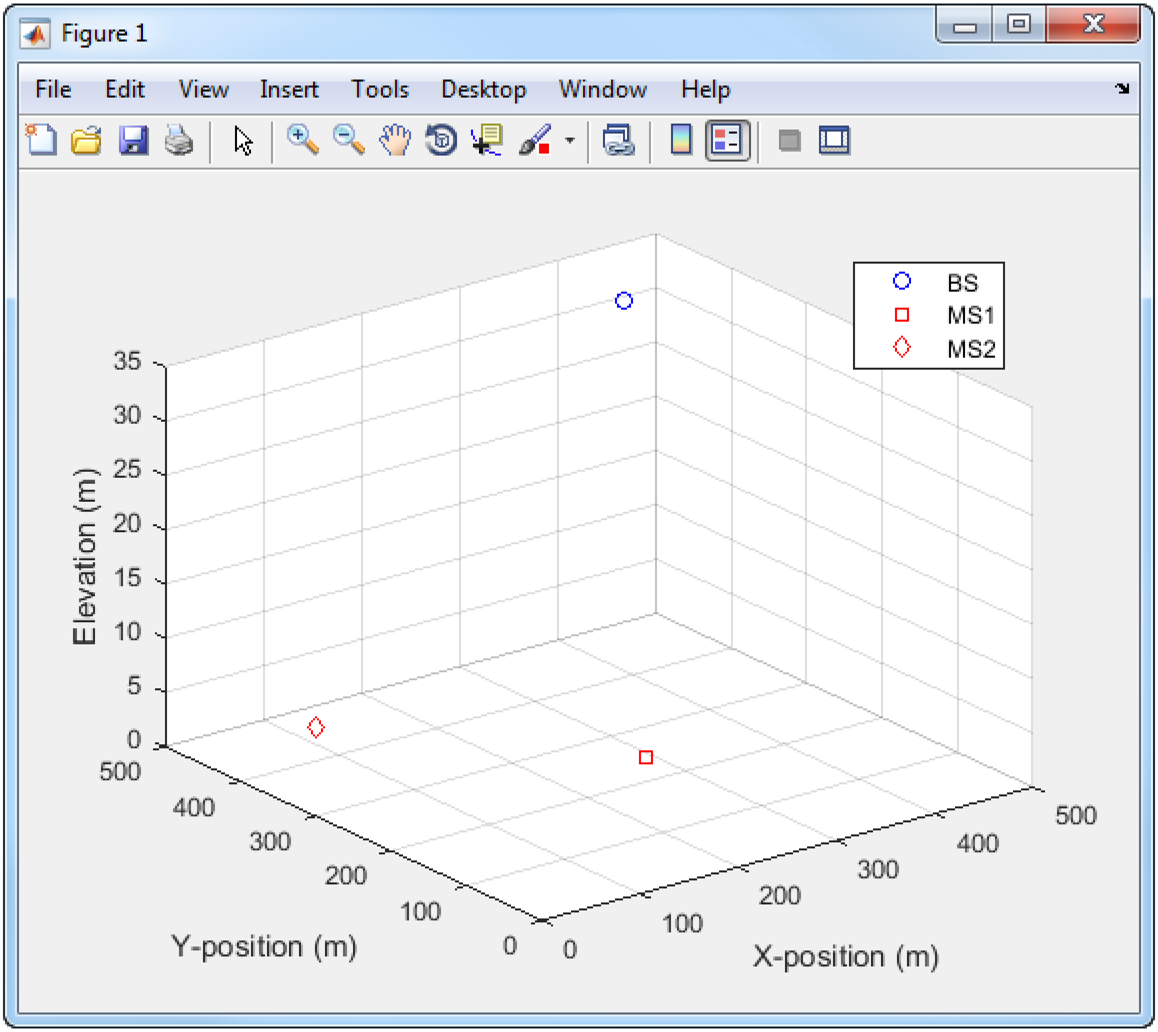
```
BSAA  = winner2.AntennaArray('UCA', 8, 0.02);  % UCA-8 array for BS
MSAA1 = winner2.AntennaArray('ULA', 2, 0.01);  % ULA-2 array for MS
MSAA2 = winner2.AntennaArray('ULA', 4, 0.005); % ULA-4 array for MS
```

Create system layout.

```
MSIdx = [2 3];
BSIdx = {1};
K = 2;
rndSeed = 5;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx, ...
    K,[BSAA,MSAA1,MSAA2],[],rndSeed);
```

Visualize BS and MS positions.

```
BSPos  = cfgLayout.Stations(cfgLayout.Pairing(1,1)).Pos;
MS1Pos = cfgLayout.Stations(cfgLayout.Pairing(2,1)).Pos;
MS2Pos = cfgLayout.Stations(cfgLayout.Pairing(2,2)).Pos;

plot3(BSPos(1),BSPos(2),BSPos(3),'bo', ...
    MS1Pos(1),MS1Pos(2),MS1Pos(3),'rs', ...
    MS2Pos(1),MS2Pos(2),MS2Pos(3),'rd');
grid on;
xlim([0 500]);
ylim([0 500]);
zlim([0 35]);
xlabel('X-position (m)');
ylabel('Y-position (m)');
zlabel('Elevation (m)');
legend('BS','MS1','MS2','Location','northeast');
```



Input Arguments

msIdx — Mobile station index

row vector

Mobile station index, specified as a row vector indicating the indices in `arrays` to serve as mobile stations.

Data Types: `double`

`bsIdx` — Mobile station index

column cell array

Base station index, specified as a column cell array, with each element representing one base station. Each cell element is an integer-valued row vector to indicate the indices in `arrays` to serve as different sectors of that base station.

Data Types: `double`

`K` — Number of links

scalar

Number of links, specified as a scalar representing the number of BS-MS links to be formulated.

Data Types: `double`

`arrays` — Antenna array configurations

vector of structures

Antenna array configurations, specified as a vector of structures defining all available arrays. All MS and BS sectors are chosen from this vector. The array of elements is typically created using the `winner2.AntennaArray` function.

Data Types: `double`

`rmax` — Maximum layout range

`500` (default) | scalar

Maximum layout range, specified as a scalar representing the maximum layout range in meters used to randomly generate the MS and BS positions.

Data Types: `double`

`seed` — Seed value

integer

Seed value used to provide repeatability, specified as an integer. When `seed` is not specified, the global random number generator is used. To assign `seed` when not assigning `rmax`, specify `rmax` as `[]`.

Data Types: `double`

Output Arguments

`cfgLayout` — Configuration layout

structure

Configuration layout, returned as a structure containing these fields, which represent the location and orientation parameters for all simulated stations.

`Stations` — Active stations

row vector of structures

Active stations, returned as a row vector of structures describing the antenna arrays for active stations. `Stations` is created from the `arrays` input and adds an additional `Velocity` field. The row ordering specifies base station (BS) sectors first, followed by the mobile stations (MS). The BS sector and MS positions are randomly assigned. The BS sectors have no velocity. Each MS has a velocity of about 1.42 m/s with a randomly assigned direction.

`NofSect` — Number of sectors

vector

Number of sectors, returned as a vector indicating the number of sectors in each BS.

`Pairing` — BS to MS pairing

matrix

BS to MS pairing, returned as a 2-by-N_L matrix, where N_L specifies the number of links to be modeled. See `Stations` for BS and MS row ordering.

`ScenarioVector` — Spatial scenario

`1` (default) | vector

Spatial scenario, returned as a 1-by-N_L vector of scenario numbers. The default is `1`, which specifies scenario A1.

The scenarios numbers map as {`1`=A1, `2`=A2, `3`=B1, `4`=B2, `5`=B3, `6`=B4, `10`=C1, `11`=C2, `12`=C3, `13`=C4, `14`=D1, `15`=D2a}.

For more information, see WINNER II Channel Models [1], Section 2.3.

`PropagConditionVector` — Propagation condition

`1` (default) | vector

Propagation condition, returned as a 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link. The default is `1`.

`StreetWidth` — Street width

`20` (default) | vector

Street width, returned as a 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. `StreetWidth` is used for the path loss model of the B1 and B2 scenarios. See `ScenarioVector` for the scenario number mapping. All elements must have the same value. `StreetWidth` applies only when the `PathLossModelUsed` field from `winner2.wimparset` is set to `'yes'`.

`Dist1` — Distances from BS to the last LOS point

`NaN` (default) | vector

Distances from BS to the last LOS point, returned as a 1-by-N_L vector. `Dist1` is used for the path loss model of the B1 and B2 scenarios. The default value of `NaN` indicates that the distance is randomly determined in path loss function. See `ScenarioVector` for the scenario number mapping. `Dist1` applies only when the `PathLossModelUsed` field from `winner2.wimparset` is set to `'yes'`.

For more information, see WINNER II Channel Models [1], Figure 4-3.

`NumFloors` — Floor numbers

`1` (default) | vector

Floor numbers, returned as a 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The `NumFloors` property is used for the path loss model of the A2 and B4 scenarios only. See `ScenarioVector` for the scenario number mapping. `NumFloors` applies only when the `PathLossModelUsed` field from `winner2.wimparset` is set to `'yes'`.

`NumPenetratedFloors` — Number of floors penetrated

`0` (default) | vector

Number of floors penetrated, returned as a 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The `NumPenetratedFloors` property is used for the NLOS path loss model of the A1 scenario. See `ScenarioVector` for the scenario number mapping. NumPenetratedFloors applies only when the PathLossModelUsed field from winner2.wimparset is set to 'yes'.

For more information, see WINNER II Channel Models [1], Table 4-4.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
comm.WINNER2Channel

Functions
winner2.AntennaArray | winner2.wim | winner2.wimparset

Introduced in R2017a

winner2.wim

Generate channel coefficients using WINNER II channel model

Download Required: To use this function, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get Add-Ons” (MATLAB) and “Manage Your Add-Ons” (MATLAB).

Syntax

```
chanCoef = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays] = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout,
initCond)
```

Description

`chanCoef = winner2.wim(cfgWim,cfgLayout)` returns channel coefficients based on the WINNER II model parameters for all links defined in the WINNER II network layout.

`[chanCoef,pathDelays] = winner2.wim(cfgWim,cfgLayout)` also returns the path delays for all links.

`[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout)` also returns the final condition of the system after generating the channel coefficients.

`[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout, initCond)` generates the channel coefficients by using the initial system conditions rather than of performing random initialization. `initCond` is of the same form as `finalCond` and is typically the `finalCond` output from the prior call of this function. Use this syntax to repeatedly generate channel coefficients for continuous time samples.

Examples

Continuously Generate WINNER 2 Channel Coefficients

Continuously generate channel coefficients for each link in a two-link system layout.

Configure model parameters.

```
cfgWim = winner2.wimparset;
cfgWim.SampleDensity = 20;
cfgWim.RandomSeed= 10; % For repeatability
```

Configure layout parameters.

```
BSAA  = winner2.AntennaArray('UCA',8,0.02);  % UCA-8 array for BS
MSAA1 = winner2.AntennaArray('ULA',2,0.01);  % ULA-2 array for MS1
MSAA2 = winner2.AntennaArray('ULA',4,0.005); % ULA-4 array for MS2
BSIdx = {1};
MSIdx = [2,3];
NL = 2;
rndSeed = 5;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx, ...
    NL,[BSAA,MSAA1,MSAA2],[],rndSeed);
```

Generate channel coefficients for the first time.

```
[H1,~,finalCond] = winner2.wim(cfgWim,cfgLayout);
```

Generate a second set of channel coefficients.

```
[H2,~,finalCond] = winner2.wim(cfgWim,cfgLayout,finalCond);
```

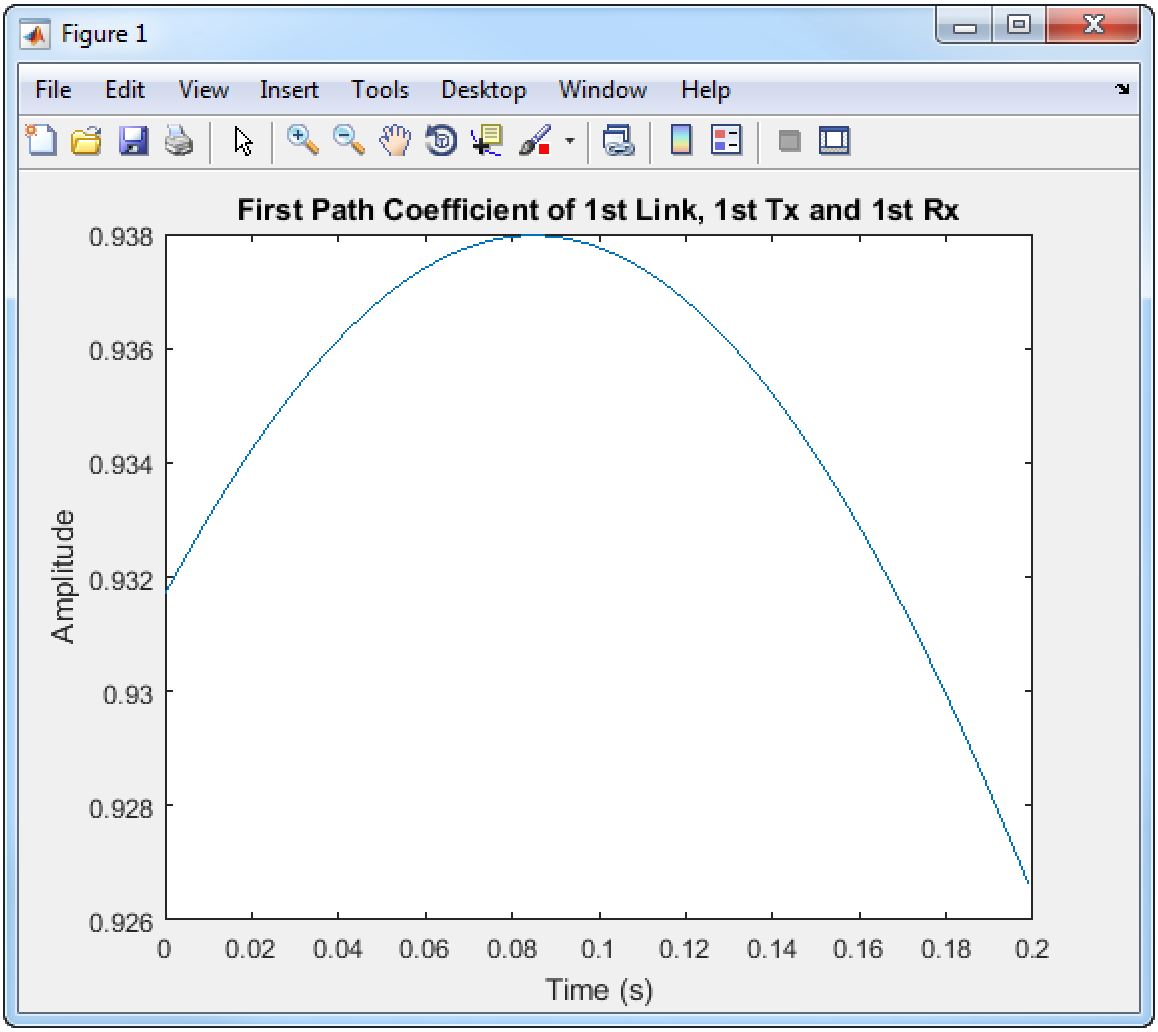
Concatenate H1 and H2 in time domain.

```
H = cellfun(@(x,y) cat(4,x,y), H1, H2, 'UniformOutput', false);
```

Plot H for the first link, 1st Tx, 1st Rx and 1st path.

```
figure;
Ts = finalCond.delta_t(1);  % Sample time for the 1st link
plot(Ts*(0:2*cfgWim.NumTimeSamples-1)', ...
    abs(squeeze(H{1}(1,1,1,:))));
xlabel('Time (s)');
```

```
ylabel('Amplitude');
title('First Path Coefficient of 1st Link, 1st Tx and 1st Rx');
```



The image shows the channel continuity over the two outputs from the `winner2.wim` function.

Input Arguments

`cfgWim` — Configuration layout

structure

Configuration model, specified as a structure containing these fields. `cfgWim` is typically created using the `winner2.wimparset` function.

`NumTimeSamples` — Number of time samples

`100` (default) | scalar

Number of time samples, specified as a scalar.

`FixedPdpUsed` — Use predefined path delays and powers for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path delays and powers for specific scenarios, specified as `'no'` or `'yes'`.

`FixedAnglesUsed` — Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios, specified as `'yes'` or `'no'`.

`IntraClusterDsUsed` — Divide each of the two strongest clusters into three subclusters per link

`'yes'` (default) | `'no'`

Divide each of the two strongest clusters into three subclusters per link, specified as `'yes'` or `'no'`.

`PolarisedArrays` — Use dual-polarized arrays

`'yes'` (default) | `'no'`

Use dual-polarized arrays, specified as `'yes'` or `'no'`.

`UseManualPropCondition` — Use manually defined propagation conditions

`'yes'` (default) | `'no'`

Use manually defined propagation conditions, specified as `'yes'` or `'no'`. Set to `'yes'` to enforce the use of manually defined propagation conditions (LOS/NLOS) in the

`PropagConditionVector` structure field returned by `winner2.layoutparset`. Set to `'no'` to draw propagation conditions from pre-defined LOS probabilities.

`CenterFrequency` — Carrier frequency
`5.25e9` (default) | scalar

Carrier frequency in Hz, specified as a scalar.

`UniformTimeSampling` — Enforce uniform time sampling
`'no'` (default) | `'yes'`

Enforce all links to be sampled at the same time instants, specified as `'no'` or `'yes'`.

`SampleDensity` — Number of time samples per half wavelength
`2e6` (default) | scalar

Number of time samples per half wavelength, specified as a scalar.

`DelaySamplingInterval` — Sampling interval
`5e-9` (default) | scalar

Sampling interval, specified as an scalar indicating the input signal sample time in seconds. `DelaySamplingInterval` defines the sampling grid to which the path delays are rounded. A value of `0` seconds indicates no rounding on path delays.

`ShadowingModelUsed` — Use shadow fading
`'no'` (default) | `'yes'`

Use shadow fading, specified as `'no'` or `'yes'`.

`PathLossModelUsed` — Use path loss model
`'no'` (default) | `'yes'`

Use path loss model, specified as `'no'` or `'yes'`.

`PathLossModel` — Path loss model
`'pathloss'` (default) | character vector

Path loss model, specified as a character vector representing a valid function name. `PathLossModel` applies only when `PathLossModelUsed` is set to `'yes'`.

`PathLossOption` — Wall material
`'CR_light'` (default) | `'CR_heavy'` | `'RR_light'` | `'RR_heavy'`

Wall material, specified as `'CR_light'`, `'CR_heavy'`, `'RR_light'`, or `'RR_heavy'`, indicating the wall material for the *A1* scenario NLOS path loss calculation. `PathLossOption` applies only when `PathLossModelUsed` is set to `'yes'`.

RandomSeed — Seed for random number generators

`[]` (default) | scalar

Seed for random number generators, specified as a scalar or empty brackets. Empty brackets, `[]`, indicate that the global random stream is used.

`cfgLayout` — Configuration layout

structure

Configuration layout, specified as a structure containing these fields, which represent the location and orientation parameters for all simulated stations. `cfgLayout` is typically created using the `winner2.layoutparset` function.

`Stations` — Active stations

row vector of structures

Active stations, specified as a row vector of structures describing the antenna arrays for active stations. `Stations` is created from the `arrays` input of `winner2.layoutparset` and adds an additional `Velocity` field. The row ordering specifies base station (BS) sectors first, followed by the mobile stations (MS). The BS sector and MS positions are randomly assigned. The BS sectors have no velocity. Each MS has a velocity of about 1.42 m/s with a randomly assigned direction.

`NofSect` — Number of sectors

vector

Number of sectors, specified as a vector indicating the number of sectors in each BS.

`Pairing` — BS to MS pairing

matrix

BS to MS pairing, specified as a 2-by-N_L matrix, where N_L specifies the number of links to be modeled. See `Stations` for BS and MS row ordering.

`ScenarioVector` — Spatial scenario

`1` (default) | vector

Spatial scenario, specified as a 1-by-N_L vector of scenario numbers. The default is `1`, which specifies scenario A1.

The scenarios numbers map as {1=A1, 2=A2, 3=B1, 4=B2, 5=B3, 6=B4, 10=C1, 11=C2, 12=C3, 13=C4, 14=D1, 15=D2a}.

For more information, see WINNER II Channel Models [1], Section 2.3.

PropagConditionVector — Propagation condition

1 (default) | vector

Propagation condition, specified as a 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link.

StreetWidth — Street width

20 (default) | vector

Street width, specified as a 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. StreetWidth is used for the path loss model of the B1 and B2 scenarios. See ScenarioVector for the scenario number mapping. All elements must have the same value. StreetWidth applies only when cfgWim.PathLossModelUsed is set to 'yes'.

Dist1 — Distances from BS to the last LOS point

NaN (default) | vector

Distances from BS to the last LOS point, specified as a 1-by-N_L vector. Dist1 is used for the path loss model of the B1 and B2 scenarios. The default value of NaN indicates that the distance is randomly determined in path loss function. See ScenarioVector for the scenario number mapping. Dist1 applies only when cfgWim.PathLossModelUsed is set to 'yes'.

For more information, see WINNER II Channel Models [1], Figure 4-3.

NumFloors — Floor numbers

1 (default) | vector

Floor numbers, specified as a 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The default value is 1. The NumFloors field is used for the path loss model of the A2 and B4 scenarios only. See ScenarioVector for the scenario number mapping. NumFloors applies only when cfgWim.PathLossModelUsed is set to 'yes'.

NumPenetratedFloors — Number of floors penetrated

0 (default) | vector

Number of floors penetrated, specified as a 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The default value is `0`. The `NumPenetratedFloors` is used for the NLOS path loss model of the A1 scenario. See `ScenarioVector` for the scenario number mapping. `NumPenetratedFloors` field applies only when `cfgWim.PathLossModelUsed` is set to `'yes'`.

For more information, see WINNER II Channel Models [1], Table 4-4.

`initCond` — Initial system condition

structure | optional

Initial system condition, specified as a structure. `initCond` is of the same form as `finalCond` and is typically the `finalCond` output from the prior call of `winner2.wim`.

Data Types: `struct`

Output Arguments

`chanCoef` — Channel coefficients

cell array containing 4-D arrays of complex values

Channel coefficients, returned as an N_L-by-1 cell array. N_L is the number of links in the system. The *i*th element of `chanCoef` is an $N_R(i)$-by-$N_T(i)$-by-$N_P(i)$-by-N_S array. N_R, N_T, and N_P are link specific. N_S is the same for all the links.

- $N_R(i)$ is the number of receive antenna elements at MS for the *i*th link.
- $N_T(i)$ is the number of transmit antenna elements at BS for the *i*th link.
- $N_P(i)$ is the number of paths for the *i*th link.
- N_S is the number of time samples given by `cfgWim.NumTimeSamples`.

For more information , see “Channel Power” on page 1-1043.

Data Types: `cell`

`pathDelays` — Path delays

matrix

Path delays, returned as an N_L-by-$maxN_P$ matrix. N_L is the number of links in the system and $maxN_P$ is the maximum number of paths among all links. Each row of the matrix applies to each link. When a link has fewer than $maxN_P$ paths, the corresponding row in `pathDelays` is `NaN` padded.

Data Types: `double`

`finalCond` — Final system condition
structure

Final system condition, returned as a structure. When generating channel coefficients for continuous time samples, use `finalCond` as the `initCond` input for the next call to `winner2.wim`.

For more information, see WINNER II Channel Models [1], Section 5.2.

Data Types: `struct`

Definitions

Channel Power

When path loss and shadowing are off, path gains of the computed WINNER channel are normalized. Specifically, path gains are normalized when the `ShadowingModelUsed` and `PathLossModelUsed` parameters are set to `'no'`.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
`comm.WINNER2Channel`

Functions
`winner2.AntennaArray` | `winner2.layoutparset` | `winner2.wimparset`

Introduced in R2017a

winner2.wimparset

WINNER II model parameter configuration

Download Required: To use this function, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get Add-Ons” (MATLAB) and “Manage Your Add-Ons” (MATLAB).

Syntax

```
cfgWim = winner2.wimparset
```

Description

`cfgWim = winner2.wimparset` returns a structure of WINNER II model parameters with their default values.

Examples

Create a WINNER II model parameter set

```
cfgWim = winner2.wimparset;
```

Adjust default settings.

```
cfgWim.RandomSeed = 31; % set the rng seed for repeatability
cfgWim.NumTimeSamples = 250;
cfgWim.CenterFrequency = 4e9;
```

Output Arguments

`cfgWim` — Configuration layout
structure

Configuration model, returned as a structure containing these fields.

`NumTimeSamples` — Number of time samples

`100` (default) | scalar

Number of time samples, specified as a scalar.

`FixedPdpUsed` — Use predefined path delays and powers for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path delays and powers for specific scenarios, specified as `'no'` or `'yes'`.

`FixedAnglesUsed` — Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios, specified as `'yes'` or `'no'`.

`IntraClusterDsUsed` — Divide each of the two strongest clusters into three subclusters per link

`'yes'` (default) | `'no'`

Divide each of the two strongest clusters into three subclusters per link, specified as `'yes'` or `'no'`.

`PolarisedArrays` — Use dual-polarized arrays

`'yes'` (default) | `'no'`

Use dual-polarized arrays, specified as `'yes'` or `'no'`.

`UseManualPropCondition` — Use manually defined propagation conditions

`'yes'` (default) | `'no'`

Use manually defined propagation conditions, specified as `'yes'` or `'no'`. Set to `'yes'` to enforce the use of manually defined propagation conditions (LOS/NLOS) in the `PropagConditionVector` structure field returned by `winner2.layoutparset`. Set to `'no'` to draw propagation conditions from pre-defined LOS probabilities.

`CenterFrequency` — Carrier frequency

`5.25e9` (default) | scalar

Carrier frequency in Hz, specified as a scalar.

`UniformTimeSampling` — Enforce uniform time sampling

`'no'` (default) | `'yes'`

Enforce all links to be sampled at the same time instants, specified as `'no'` or `'yes'`.

`SampleDensity` — Number of time samples per half wavelength

`2e6` (default) | scalar

Number of time samples per half wavelength, specified as a scalar.

`DelaySamplingInterval` — Sampling interval

`5e-9` (default) | scalar

Sampling interval, specified as an scalar indicating the input signal sample time in seconds. `DelaySamplingInterval` defines the sampling grid to which the path delays are rounded. A value of `0` seconds indicates no rounding on path delays.

`ShadowingModelUsed` — Use shadow fading

`'no'` (default) | `'yes'`

Use shadow fading, specified as `'no'` or `'yes'`.

`PathLossModelUsed` — Use path loss model

`'no'` (default) | `'yes'`

Use path loss model, specified as `'no'` or `'yes'`.

`PathLossModel` — Path loss model

`'pathloss'` (default) | character vector

Path loss model, specified as a character vector representing a valid function name. `PathLossModel` applies only when `PathLossModelUsed` is set to `'yes'`.

`PathLossOption` — Wall material

`'CR_light'` (default) | `'CR_heavy'` | `'RR_light'` | `'RR_heavy'`

Wall material, specified as `'CR_light'`, `'CR_heavy'`, `'RR_light'`, or `'RR_heavy'`, indicating the wall material for the *A1* scenario NLOS path loss calculation. `PathLossOption` applies only when `PathLossModelUsed` is set to `'yes'`.

RandomSeed — Seed for random number generators
[ ] (default) | scalar

Seed for random number generators, specified as a scalar or empty brackets. Empty brackets, `[]`, indicate that the global random stream is used.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
`comm.WINNER2Channel`

Functions
`winner2.layoutparset` | `winner2.wim`

Introduced in R2017a

2

Blocks — Alphabetical List

A-Law Compressor

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

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

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 `sign` function.

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.

Parameters

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

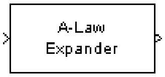
References

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

Introduced before R2006a

A-Law Expander

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.

$$x = \begin{cases} \dfrac{y(1+\log A)}{\mathrm{A}} & \text{for } 0 \le |y| \le \dfrac{\mathrm{V}}{1+\log\mathrm{A}} \\ \exp\left(|\mathrm{y}|(1+\log A)/V-1\right)\dfrac{V}{A}\,\mathrm{sgn}(y) & \text{for } \dfrac{\mathrm{V}}{1+\log\mathrm{A}} < |y| \le V \end{cases}$$

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

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.

Introduced before R2006a

Algebraic Deinterleaver

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.

Parameters

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.

Introduced before R2006a

Algebraic Interleaver

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

$$c(n) = \operatorname{mod}\left(k \cdot \frac{n \cdot (n-1)}{2}, N\right) + 1, n$$

for integers n between 1 and N. The intermediate permutation function is obtained by using the following relationship:

$$\Pi\big(c(n)\big) = c(n+1)$$

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*.

Parameters

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

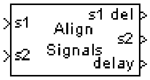
[1] Heegard, Chris and Stephen B. Wicker. *Turbo Coding*. Boston: Kluwer Academic Publishers, 1999.

[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.

Introduced before R2006a

Align Signals

Align two signals by finding delay between them



Align Signals

Library

Utility Blocks

Description

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” 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.

Parameters

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.

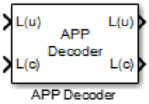
See Also

Find Delay, Error Rate Calculation

Introduced in R2012a

APP Decoder

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 $L(\mathrm{u})$ 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”. 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.

Parameters

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.

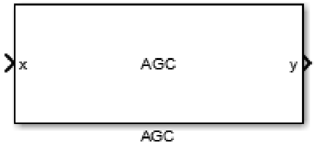
[2] Benedetto, Sergio and Guido Montorsi, “Performance of Continuous and Blockwise Decoded Turbo Codes.” *IEEE Communications Letters*, Vol. 1, May 1997, 77–79.

[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.

Introduced before R2006a

AGC

Adaptively adjust gain for constant signal-level output



Library

RF Impairments Correction

Description

The automatic gain controller (AGC) block adaptively adjusts its gain to achieve a constant signal level at the output.

Parameters

Step size

Specify the step size for gain updates as a double-precision or single-precision real positive scalar. The default is `0.01`.

If you increase **Step size**, the AGC responds faster to changes in the input signal level. However, gain pumping also increases.

Desired output power (W)

Specify the desired output power level as a real positive scalar. The power level is specified in Watts referenced to 1 ohm. The default is `1`.

Averaging length

Specify the length of the averaging window in samples as a positive integer scalar. The default is `100`.

Note If you use the AGC with higher order QAM signals, you might need to reduce the variation in the gain during steady-state operation. Inspect the constellation diagram at the output of the AGC and increase the averaging length as needed. An increase in **Averaging length** reduces execution speed.

Maximum power gain (dB)

Specify the maximum gain of the AGC in decibels as a positive scalar. The default is `60`.

If the AGC input signal power is very small, the AGC gain will be very large. This can cause problems when the input signal power suddenly increases. Use **Maximum power gain (dB)** to avoid this by limiting the gain that the AGC applies to the input signal.

Enable output of estimated input power

Select this check box to provide an input signal power estimate to an output port. By default, the check box is not selected.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.AGC` reference page. The object properties correspond to the block parameters.

Examples

To open these examples, enter the example names at the MATLAB command prompt:

- doc_agc_received_signal_amplitude — Adaptively adjusts the received signal power to approximately 1 Watt.
- doc_agc_plot_step_size — Plots the effect of step size on AGC performance.
- doc_agc_plot_max_gain — Shows how the maximum gain affects the ability of the AGC to reach its target output power.

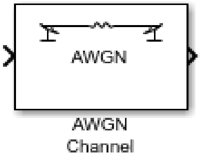
See Also

`comm.AGC`

Introduced in R2013a

AWGN Channel

Add white Gaussian noise to input signal
Library: Communications System Toolbox / Channels



Description

The AWGN Channel block adds white Gaussian noise to the input signal. It inherits the sample time from the input signal.

Ports

Input

In — Input data signal
vector | matrix

Input data signal, specified as an N_S-by-1 vector or an N_S-by-N_C matrix.

N_S represents the number of samples in the input signal. N_C represents the number of channels, as determined by the number of columns in the input signal matrix. Both N_S and N_C can be equal to 1.

The block adds frames of length-N_S Gaussian noise to each of the N_C channels, using a distinct random distribution per channel.

Data Types: `double` | `single`
Complex Number Support: Yes

Var — Variance of additive white Gaussian noise
positive scalar | vector

Variance of additive white Gaussian noise, specified as a positive scalar or a 1-by-N_C vector. N_C represents the number of channels, as determined by the number of columns in the input signal matrix. For more information, see “Specifying the Variance Directly or Indirectly” on page 2-26.

Dependencies

To enable this port, set Mode to `Variance from port`.

Data Types: `double`

Output

`Out` — Output data signal

vector | matrix

Output data signal for the AWGN channel, returned as a vector or matrix. The datatype and dimensions of `Out` match those of the input signal, In.

Parameters

`Initial seed` — Noise generator initial seed

`67` (default) | positive scalar | vector

Noise generator initial seed, specified as a positive scalar or a 1-by-N_C vector.

This block uses the Random Source block to generate noise. Random numbers are generated using the Ziggurat method (V5 RANDN algorithm). The block reuses the same initial seeds every time you rerun the simulation, so that this block outputs the same signal each time you run a simulation.

When the input signal is complex, the block creates random data as:

```
randData = randn(2*NS,NC)
noise = randData(1:2:end) + 1i(randData(2:2:end))
```

N_S is the number of samples and N_C is the number of channels.

You can specify different seed values for each DLL build.

Tunable: Yes

Mode — Variance mode

`Signal to noise ratio (Eb/No)` (default) | `Signal to noise ratio (Es/No)` | `Signal to noise ratio (SNR)` | `Variance from mask` | `Variance from port`

Variance mode, specified as `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)`, `Signal to noise ratio (SNR)`, `Variance from mask`, or `Variance from port`. For more information, see "Relationship Among Eb/No, Es/No, and SNR Modes" on page 2-25 and "Specifying the Variance Directly or Indirectly" on page 2-26.

`Eb/No (dB)` — Ratio of information bit energy per symbol to noise power spectral density

`10` (default) | scalar | vector

Ratio of information bit energy per symbol to noise power spectral density in decibels, specified as a scalar or vector. The information bit energy is the magnitude without channel coding.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Eb/No`.

`Es/No (dB)` — Ratio of information symbol energy per symbol to noise power spectral density

`10` (default) | scalar | vector

Ratio of information symbol energy per symbol to noise power spectral density in decibels, specified as a scalar or vector. The information bit energy is the magnitude without channel coding.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Es/No`.

`SNR (dB)` — Ratio of signal power to noise power

`10` (default) | scalar | vector

Ratio of signal power to noise power in decibels, specified as a scalar or vector.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `SNR`.

`Number of bits per symbol` — Number of bits in each input symbol

1 (default) | scalar | vector

Number of bits in each input symbol, specified as a scalar or vector.

Dependencies

To enable this parameter, set Mode to `Eb/No`.

`Input signal power, referenced to 1 ohm (watts)` — Mean square power of input

`1` (default) | scalar | vector

Mean square power of the input in watts, specified as a scalar or vector.

- When Mode is `Eb/No` or `Es/No`, the parameter is the mean square power of the input symbols.
- When `Mode` is `SNR`, this parameter is the mean square power of the input samples.

Tunable: Yes

Dependencies

To enable this parameter, set `Mode` to `Eb/No`, `Es/No`, or `SNR`.

`Symbol period (s)` — Duration of an information channel

`1` (default) | positive scalar | vector

Duration of an information channel symbol in seconds, specified as a positive scalar or vector. The duration of the information channel is measured without channel coding.

Dependencies

To enable this parameter, set Mode to `Eb/No` or `Es/No`.

`Variance` — Variance of white Gaussian noise

`1` (default) | scalar | vector

Variance of the white Gaussian noise, specified as a scalar or vector. For more information, see “Specifying the Variance Directly or Indirectly” on page 2-26.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Variance from mask`.

Tips

- You can tune parameters in normal mode, accelerator mode, or rapid accelerator mode.
- Unless otherwise indicated, parameters are *nontunable*.
  - For nontunable parameters, when you use the Simulink Coder™ rapid simulation (RSIM) target to build an RSIM executable, you cannot change their values without recompiling the model.
  - If a parameter is *tunable*, you can change its value at any time. This is useful for Monte Carlo simulations in which you run the simulation multiple times (such as on multiple computers) with different amounts of noise.

Algorithms

Relationship Among Eb/No, Es/No, and SNR Modes

For uncoded complex input signals, the AWGN Channel block relates $E_b/N_{0,}$ E_s/N_0, and SNR according to these equations:

$$E_s/N_0 = (T_{sym}/T_{samp}) \cdot SNR$$
$$E_s/N_0 = E_b/N_0 + 10\log_{10}(k) \text{ in dB}$$

- E_s represents the signal energy in joules.
- E_b represents the bit energy in joules.
- N_0 respresents the noise power spectral density in watts/Hz.
- T_{sym} represents the Symbol period (s) parameter of the block in `Es/No` mode.
- k represents the number of information bits per input symbol, Number of bits per symbol.
- T_{samp} represents 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 this equation:

$$E_s/N_0 = 0.5\ (T_{sym}/T_{samp}) \cdot SNR$$

Note

- All values of power assume a nominal impedance of 1 ohm.
- The equation for the real case differs from the corresponding equation for the complex case by a factor of 2. Specifically, the object 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, see “AWGN Channel Noise Level”.

Specifying the Variance Directly or Indirectly

To directly specify the variance of the noise generated by AWGN Channel, specify the Mode as:

- `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.

For `Variance from mask` and `Variance from port` mode:

- 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 for each quadrature component of the complex noise is half of the calculated or specified value.

To specify the variance indirectly, that is, to have the block calculate the variance, specify the Mode as:

- `Signal to noise ratio (Eb/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:

  - Eb/No (dB), the ratio of bit energy to noise power spectral density
  - Number of bits per symbol
  - Input signal power, referenced to 1 ohm (watts), the actual power of the symbols at the input of the block
  - Symbol period (s)
- `Signal to noise ratio (Es/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:

  - Es/No (dB), the ratio of signal energy to noise power spectral density
  - Input signal power, referenced to 1 ohm (watts), the actual power of the symbols at the input of the block
  - Symbol period (s)
- `Signal to noise ratio (SNR)`, where the block calculates the variance from these quantities that you specify in the dialog box:

  - SNR (dB), the ratio of signal power to noise power
  - Input signal power, referenced to 1 ohm (watts), the actual power of the samples at the input of the block

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.

$$NoiseVariance = \frac{SignalPower \times SymbolPeriod}{SampleTime \times 10^{\frac{Es/No}{10}}}$$

Tip Select the symbol period equal to the symbol period of the model. The value depends on what constitutes a symbol and what the oversampling applied to it is. For example, a symbol could have 3 bits and be oversampled by 4. For more information, see “AWGN Channel Noise Level”.

References

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

See Also

Blocks
MIMO Fading Channel | Random Source

System Objects
`comm.AWGNChannel`

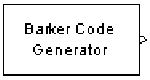
Topics

“Gray Coded 8-PSK”
“Filter Using Simulink Raised Cosine Filter Blocks”
“Reed Solomon Examples with Shortening, Puncturing, and Erasures”

Introduced before R2006a

Barker Code Generator

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]`

Code length	Barker Code
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]`

Parameters

Code length

The length of the Barker code.

Sample time

The time between each sample of the output signal.

Samples per frame

The number of samples in one column of the output signal.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

See Also

PN Sequence Generator

Introduced before R2006a

Baseband File Reader

Read baseband signals from file
Library: Communications System Toolbox / Comm Sources

Description

The Baseband File Reader block reads a signal from a baseband file. A baseband file is a specific type of binary file written by the Baseband File Writer block. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. The block automatically reads the sample rate, center frequency, number of channels, and any descriptive data.

Input/Output Ports

Output

`Data` — Baseband signal
scalar | vector | matrix

Baseband signal, returned as a scalar, vector, or matrix. The signal is read from the file specified by the **Baseband file name** parameter. The sample time is either inherited from the file or can be set by the **Sample Time (s)** parameter.

Data Types: `double`

`EOF` — End-of-file indicator
logical scalar

End-of-file indicator, returned as a logical scalar. The output is `true` when the **Repeatedly read the file** parameter is `false` and the entire file has been read. To enable this port, select the **Output end-of-file indicator** parameter.

Parameters

`Baseband file name` — Name of file from which data is read
`example.bb` (default) | character vector

Specify the name of the baseband file as a character vector.

Click **Browse** to locate the baseband file you want to read. Click **File Info** to display this information:

- File name
- Sample rate
- Center frequency
- Number of samples
- Number of channels
- Data type
- Any metadata fields

Data Types: `char`

`Inherit sample time from file` — Select source of sample time
`on` (default) | `off`

Select this check box to inherit the sample time from the file specified by **Baseband file name**.

`Sample time (s)` — Block sample time
`1` (default) | positive scalar

Specify the block sample time in seconds as a real positive scalar. To enable this parameter, clear the **Inherit sample time from file** check box.

`Samples per frame` — Number of samples in one frame
`100` (default) | positive integer scalar

Specify the samples per frame as a positive integer scalar.

`Repeatedly read the file` — Continuously loop data from file
`off` (default) | `on`

Select this check box to repeatedly read the contents of the baseband file. If the end of the file is reached, the block exhibits this behavior:

- Parameter is selected — The block outputs zeros.
- Parameter is not selected — The block outputs samples from the beginning of the file.

`Simulate using` — Select simulation mode

`Code generation` (default) | `Interpreted execution`

`Code generation`

> On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

`Interpreted execution`

> Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

See Also

Baseband File Writer

Introduced in R2016b

Baseband File Writer

Write baseband signals to file
Library: Communications System Toolbox / Comm Sinks

Description

The Baseband File Writer block writes a baseband signal to a specific type of binary file. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. Sample rate, which is determined by the input signal sample time and frame size, and center frequency are saved when the signal is written to a file.

Input/Output Ports

Input

`Port_1` — Baseband signal
scalar | vector | matrix

This port accepts a baseband signal to be saved under the filename specified by the **Baseband file name** parameter. The saved signal is always complex.

Data Types: `single` | `double`

Parameters

`Baseband file name` — Name of file in which data is saved
`untitled.bb` (default) | character vector

Specify the name of the baseband file as a character vector.

To specify the location where the file is saved, click **Browse**.

`Center frequency (Hz)` — Center frequency of the baseband signal
`1e8` (default) | nonnegative scalar

Specify the center frequency in Hz as a nonnegative scalar.

`Metadata in a structure` — Data describing the baseband signal
`struct()` (default) | structure

Specify data describing the baseband signal as a structure. If the signal has no descriptive data, this parameter is an empty structure. The structure can contain any number of fields. Field names have no restrictions, but the field values must be numeric, logical, or character data types having any dimension.

`Number of latest samples to write` — Number of samples to write
`inf` (default) | positive scalar

Specify the number to write. If this parameter is `inf`, all samples are saved. Otherwise, only the last *N* samples are saved, where *N* is specified by this parameter.

`Simulate using` — Select simulation mode
`Code generation` (default) | `Interpreted execution`

`Code generation`

On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

`Interpreted execution`

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Tips

- The Baseband File Writer block writes baseband signals to uncompressed binary files. To share these files, you can compress them to a zip file using the `zip` function. For more information, see “Create and Extract from Zip Archives” (MATLAB).

See Also

Baseband File Reader

Introduced in R2016b

Baseband PLL

Implement baseband phase-locked loop

Baseband
PLL
Filt
PD
VCO

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.

Parameters

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

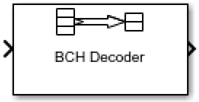
For more information about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization” in *Communications System Toolbox User*'*s Guide*.

Introduced before R2006a

BCH Decoder

Decode BCH code to recover binary vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding



Description

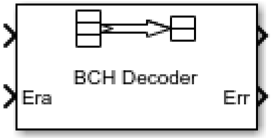
The BCH Decoder block recovers a binary message vector from a binary BCH codeword vector. For proper decoding, the **Codeword length, N** and **Message length, K** parameter values in this block must match the parameters in the corresponding BCH Encoder block. The full-length values of *N* and *K* must produce a valid narrow-sense BCH code.

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 2-45.

See “Tips” on page 2-47 for information about valid *N* values, valid (*N*,*K*) pairs, and error-correcting capabilities for a given BCH code.

If decoding fails, 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.



This icon shows optional ports.

Ports

Input

`In` — Encoded message

binary column vector

Encoded message, specified as a binary column vector. The encoded message is a BCH code with message length *K* and codeword length (*N* – number of punctures).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

`Era` — Erasure vector

binary column vector

Erasure vector, specified as a binary column vector that is the same length as `In`. Erasure values of 1 correspond to erased bits in the same position in the codeword. Values of 0 correspond to bits that are not erased.

Dependencies

To enable this port, select **Enable erasures input port**.

Data Types: `double` | `Boolean`

Output

`Out` — Decoded message

binary column vector

Decoded message, returned as a binary column vector input signal with an integer multiple of **Message length, K** elements or **Shortened message length, S** elements if the code is shortened. Each group of input elements represents one codeword to decode. The input and output signal lengths are listed in the “Input and Output Signal Length in BCH Blocks” on page 2-45 table.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

`Err` — Decoding errors

integer vector

Decoding errors, returned as an integer vector that indicates the number of errors detected during decoding of the codeword. A negative integer indicates that the block detected more errors than it could correct by using the coding scheme.

Dependencies

To enable this port, select **Output number of corrected errors**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

For more information, see “Supported Data Types” on page 2-46.

Parameters

`Codeword length, N` — Codeword length

`15` (default) | integer

Codeword length, specified as an integer of the form *N* = 2^M–1, where *M* is an integer from 3 through 16. For more information, see “Tips” on page 2-47.

`Message length, K` — Message length

`5` (default) | integer

Message length, specified as an integer. The (`N`, `K`) pair must produce a narrow-sense BCH code.

`Shortened message length, S` — Shortened message length

`5` (default) | integer

Shortened message length, specified as an integer. When you specify this parameter, provide full-length *N* and *K* values to specify the (*N*, *K*) code that is shortened to an (*N*–*K* +*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'` (default) | polynominal character vector | binary row vector | binary Galois row vector

Generator polynomial, specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.
- A binary Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'`, which is equivalent to `bchgenpoly(15,5)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^4 + X + 1'` (default) | polynominal character vector | binary row vector

Primitive polynomial in order of descending power. It is a polynomial of order *M* that defines the finite Galois field GF(2), specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^4 + X + 1'`, which is the primitive polynomial used for a (15,5) code, `de2bi(primpoly(4,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Disable generator polynomial checking` — Option to disable generator polynomial checking

on (default) | off

Select this parameter to disable generator polynomial check.

Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the specified generator polynomial, where *N* represents the full codeword length. For larger codes, disabling the check speeds up the simulation process.

Tip Always run the check at least once before disabling this feature.

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Puncture vector` — Puncture vector

`[ones(8,1); zeros(2,1)]` (default) | column vector

Puncture vector, specified as a binary column vector of length *N*–*K*. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 2-46.

Dependencies

To enable this parameter, select **Puncture code**.

`Enable erasures input port` — Option to enable erasures input port

off (default) | on

Selecting this check box enables 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. Values of `0` correspond to nonerased symbols. For more information, see “Puncturing and Erasures” on page 2-46.

`Output number of corrected errors` — Option to enable port to output number of corrected errors

off (default) | on

Selecting this check box enables an additional output port, `Err`, which indicates the number of errors the block corrected in the input codeword.

Definitions

Input and Output Signal Length in BCH Blocks

This table shows how to compute the input and output signal lengths for the BCH encoder and decoder blocks.

The notation `y = c * x` denotes that `y` is an integer multiple of `x`.

Specify Shortened Message Length, S	BCH Encoder	BCH Decoder
off	**Input Length:** *c * K* **Output Length:** *c * (N – P)*	**Input Length:** *c * (N – P)* **Output Length:** *c * K* **Erasures Length:** *c * (N – P)*
on	**Input Length:** *c * S* **Output Length:** *c * (N - K + S - P)*	**Input Length:** *c * (N - K + S - P)* **Output Length:** *c * S* **Erasures Length:** *c * (N - K + S - P)*

- *N* is the codeword length
- *K* is the message length
- *S* is the shortened message length
- *P* is the number of punctures value, and is equal to the number of zeros in the puncture vector.

Puncturing and Erasures

1s and 0s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- 1 means that the data symbol is passed through the block unaltered.
- 0 means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- 1 means that the data symbol is to be replaced with an erasure symbol.
- 0 means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-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 • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Encoder — Encodes data using BCH algorithm.

Tips

- To generate the list of valid (`N`,`K`) pairs along with the corresponding values of the error-correction capability, run `bchnumerr(N)`.
- Valid values for `N` = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of `N` is 65,535.

Algorithms

This block implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

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.

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

See Also

Blocks
BCH Encoder

System Objects
`comm.BCHDecoder`

Functions
`bchdec` | `bchgenpoly` | `primpoly`

Topics

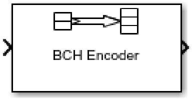
"Block Codes"

Introduced before R2006a

BCH Encoder

Create BCH code from binary vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding



Description

The BCH Encoder block creates a BCH code with message length *K* and codeword length (*N* – number of punctures).

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 2-52.

See “Tips” on page 2-54 for information about valid *N* values, valid (*N,K*) pairs, and error-correcting capabilities for a given BCH code.

Ports

Input

In — Message to encode
binary column vector

Message to encode, specified as a binary column vector input signal with an integer multiple of **Message length, K** elements or **Shortened message length, S** elements if the code is shortened. Each group of input elements represents one message word to encode. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 2-52.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Out` — Encoded message

binary column vector

Encoded message, returned as a binary column vector. The encoded message is a BCH code with message length *K* and codeword length (*N* – number of punctures).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

For more information, see “Supported Data Types” on page 2-54.

Parameters

`Codeword length, N` — Codeword length

`15` (default) | integer

Codeword length, specified as an integer of the form $N = 2^M–1$, where *M* is an integer from 3 through 16. For more information, see “Tips” on page 2-54.

`Message length, K` — Message length

`5` (default) | integer

Message length, specified as an integer. The (`N`, `K`) pair must produce a narrow-sense BCH code.

`Shortened message length, S` — Shortened message length

`5` (default) | integer

Shortened message length, specified as an integer. When you specify this parameter, provide full-length *N* and *K* values to specify the (*N*, *K*) code that is shortened to an (*N*–*K*+*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'` (default) | polynominal character vector | binary row vector | binary Galois row vector

Generator polynomial, specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.
- A binary Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'`, which is equivalent to `bchgenpoly(15,5)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^4 + X + 1'` (default) | polynominal character vector | binary row vector

Primitive polynomial in order of descending power. It is a polynomial of order *M* that defines the finite Galois field GF(2), specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^4 + X + 1'`, which is the primitive polynomial used for a (15,5) code, `de2bi(primpoly(4,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Disable generator polynomial checking` — Option to disable generator polynomial checking

on (default) | off

Select this parameter to disable generator polynomial check.

Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the specified generator polynomial, where *N* represents the full codeword length. For larger codes, disabling the check speeds up the simulation process.

Tip Always run the check at least once before disabling this feature.

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Puncture vector` — Puncture vector

`[ones(8,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as a binary column vector of length *N*–*K*. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. 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 through the block unaltered. This convention applies to both the encoder and the decoder.

Dependencies

To enable this parameter, select **Puncture code**.

Definitions

Input and Output Signal Length in BCH Blocks

This table shows how to compute the input and output signal lengths for the BCH encoder and decoder blocks.

The notation `y = c * x` denotes that `y` is an integer multiple of `x`.

Specify Shortened Message Length, S	BCH Encoder	BCH Decoder
off	**Input Length:** *c * K* **Output Length:** *c * (N – P)*	**Input Length:** *c * (N – P)* **Output Length:** *c * K* **Erasures Length:** *c * (N – P)*
on	**Input Length:** *c * S* **Output Length:** *c * (N - K + S - P)*	**Input Length:** *c * (N - K + S - P)* **Output Length:** *c * S* **Erasures Length:** *c * (N - K + S - P)*

- *N* is the codeword length
- *K* is the message length
- *S* is the shortened message length
- *P* is the number of punctures value, and is equal to the number of zeros in the puncture vector.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Decoder — Decodes BCH encoded data.

Tips

- To generate the list of valid (N,K) pairs along with the corresponding values of the error-correction capability, run `bchnumerr(N)`.
- Valid values for N = 2^M–1, where M is an integer from 3 through 16. The maximum allowable value of N is 65,535.

Algorithms

This block implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

References

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

See Also

Blocks
BCH Decoder

System Objects
`comm.BCHEncoder`

Functions
`bchenc` | `bchgenpoly` | `primpoly`

Topics

"Block Codes"

Introduced before R2006a

Bernoulli Binary Generator

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 column or row vector, a two-dimensional matrix, or a scalar. The number of rows in the output signal corresponds to the number of samples in one frame and is determined by the **Samples per frame** parameter. The number of columns in the output signal corresponds to the number of channels and is determined by the number of elements in the **Probability of a zero** parameter. See “Sources and Sinks” in *Communications System Toolbox User's Guide* for more details.

Parameters

Probability of a zero

The probability with which a zero output occurs. Specify the probability as a scalar or row vector whose elements are real numbers between 0 and 1. The number of elements in the **Probability of a zero** parameter correspond to the number of independent channels output from the block.

Source of initial seed

The source of the initial seed for the random number generator. Specify the source as either `Auto` or `Parameter`. When set to `Auto`, the block uses the global random number stream.

Note When **Source of initial seed** is `Auto` in `Code generation` mode, the random number generator uses an initial seed of zero. Therefore, the block generates the same random numbers each time it is started. Use `Interpreted execution` to ensure that the model uses different initial seeds. If `Interpreted execution` is run in `Rapid accelerator` mode, then it behaves the same as `Code generation` mode.

Initial seed

The initial seed value for the random number generator. Specify the seed as a nonnegative integer scalar. **Initial seed** is available when the **Source of initial seed** parameter is set to `Parameter`.

Sample time

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal. Specify **Samples per frame** as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a binary sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Output data type

The output type of the block can be specified as a `boolean`, `uint8`, `uint16`, `uint32`, `single`, or `double`. The default is `double`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

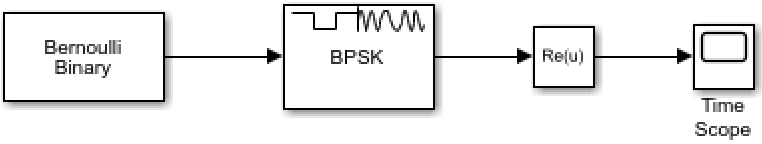
`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

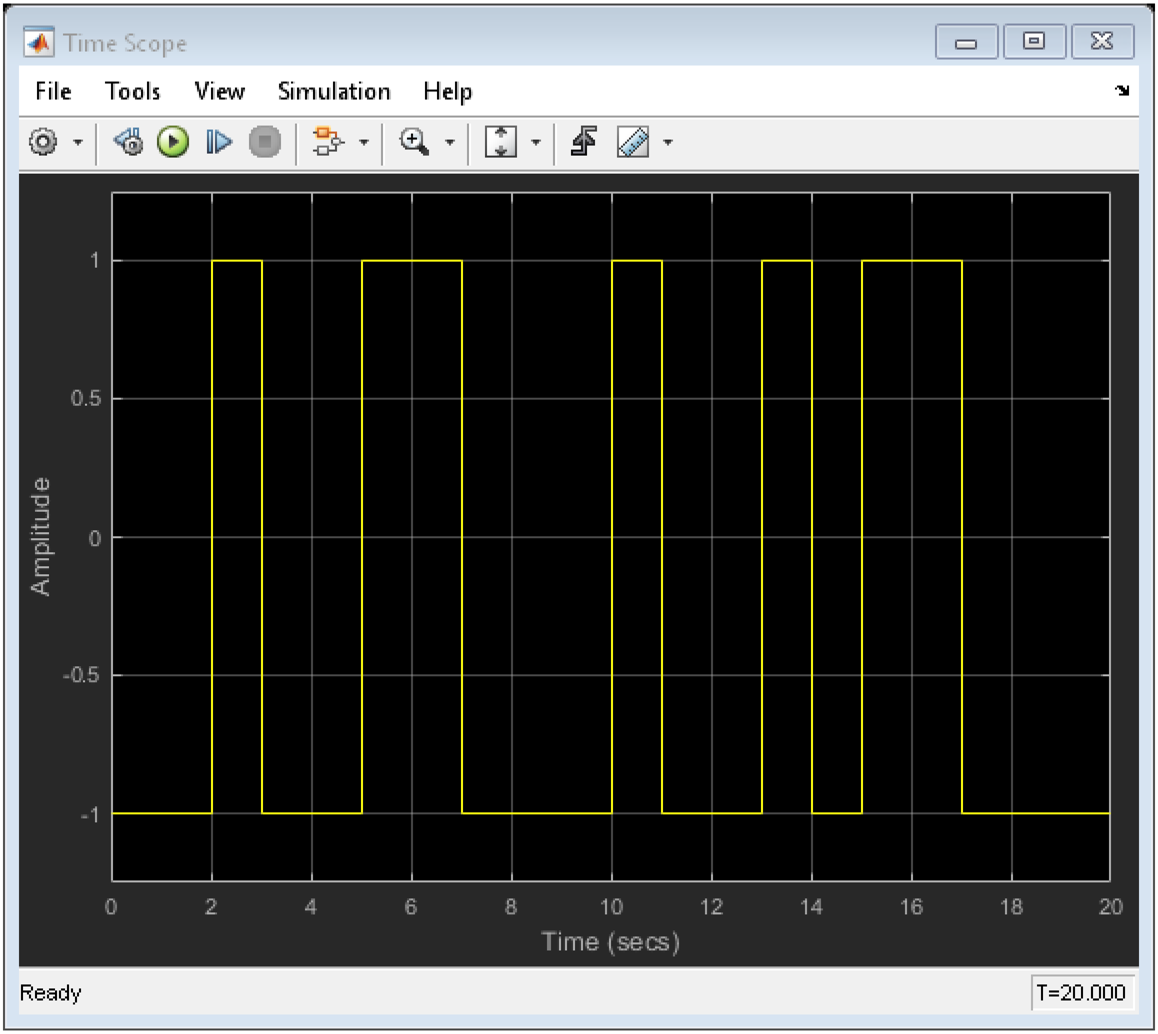
Examples

Generate Bernoulli Binary Numbers

Open the Bernoulli generator model. The model generates binary data, applies BPSK modulation, and displays the output.



Run the model.



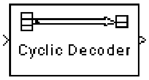
See Also

Random Integer Generator, Binary Symmetric Channel, `randi`, `rand`

Introduced before R2006a

Binary Cyclic Decoder

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.

This block accepts a column vector input signal containing *N* elements, where *N* is the codeword length. The output signal is a column vector containing *K* elements, where *K* is the message length of the cyclic code.

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 polynomial character vector or 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 2-61 table on this page.

Parameters

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 input vector length, a polynomial character vector, 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

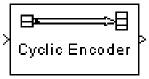
See Also

`cyclpoly`

Introduced before R2006a

Binary Cyclic Encoder

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*.

This block accepts a column vector input signal containing *K* elements. The output 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 polynomial character vector or 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 2-64 table on this page.

Parameters

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, a polynomial character vector, 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

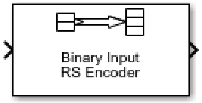
`cyclpoly` (in the Communications System Toolbox documentation)

Introduced before R2006a

Binary-Input RS Encoder

Create Reed-Solomon code from binary vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding

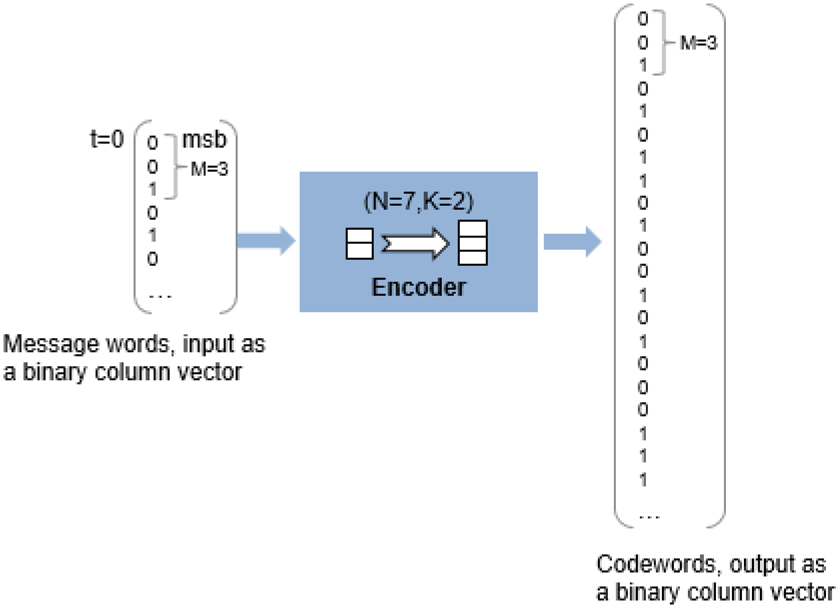


Description

The Binary-Input RS Encoder block creates a Reed-Solomon code.

The symbols for the code are binary sequences of length *M*, corresponding to elements of the Galois field *GF*(2^M). The first bit in each symbol is the most significant bit.

Suppose M = 3, N = 2^3-1 = 7, and K = 2. Then a message is a vector of length 2 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*=7 and message word length *K*=2. Since $N=2^M-1$, when *N*=7, the symbol length, *M*=3.


t=0
0
0
1
0
1
0
...
msb
M=3
Message words, input as a binary column vector
(N=7,K=2)
Encoder
0
0
1
0
1
0
1
1
0
1
0
0
1
0
1
0
0
0
1
1
1
...
M=3
Codewords, output as a binary column vector

Each input message word is a binary vector of length 6, that represents 2 three-bit integers. Each corresponding output codeword is a binary vector of length 21 that represents 7 three-bit integers. For more information, see “Input and Output Signal Length in RS Blocks” on page 2-70.

Ports

Input

In — Message

binary column vector

Message in bits, specified as one of the following:

- When there is no message shortening, a (N_C×K×M)-by-1 binary column vector.
- When there is message shortening, a (N_C×S×M)-by-1 binary column vector.

N_C is the number of message words, K is the **Message length K (symbols)**, M is the number of bits per symbol, and S is the **Shortened message length S (symbols)**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-70.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

Output

`Out` — Reed-Solomon codeword

binary column vector

Reed-Solomon codeword in bits, returned as an (N_C×(N – K + S – P)×M)-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-70.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

For more information, see “Supported Data Types” on page 2-73.

Parameters

`Codeword length N (symbols)` — Codeword length

7 (default) | integer

Codeword length in symbols, specified as an integer.

For more information, see “Restrictions on the M and the Codeword Length N” on page 2-72 and “Input and Output Signal Length in RS Blocks” on page 2-70.

`Message length K (symbols)` — Message word length

3 (default) | integer

Message word length in symbols, specified as an integer from 1 to *N*–2, where *N* is the codeword length.

`Shortened message length S (symbols)` — Shortened message word length

`3` (default) | integer

Shortened message word length in symbols, specified as an integer, such that $S \leq K$. When **Shortened message length S (symbols) < Message length K (symbols)**, the Reed-Solomon code is shortened.

You still specify *N* and *K* values for the full-length (*N*, *K*) code but the decoding is shortened to an (*N*–*K*+*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values from 0 to 2^M–1, in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see “Specify the Generator Polynomial” on page 2-72.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Restrictions on the M and the Codeword Length N” on page 2-72.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector
`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 2-73.

Dependencies

To enable this parameter, select **Puncture code**.

`Output data type` — Output type of the block
`Same as input` (default) | `boolean` | `double`

Output type of the block, specified as `Same as input`, `boolean`, or `double`.

Definitions

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, *K*, or shortened message word length, *S*. The codeword length is *N* – *K* + *S* – *P*, where *N* is the full codeword length and

P is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N - P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that *y* is an integer multiple of *x*.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Input RS Encoder	**Input Length (bits):**
	$N_C \times K \times M$
	Output Length (bits):
	$N_C \times (N–P) \times M$
		Input Length (bits):
		$N_C \times S \times M$
		Output Length (bits):
		$N_C \times (N–K+S–P) \times M$

Binary-Output RS Decoder	**Input Length (bits):**
	$N_C \times (N–P) \times M$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (bits):
	$N_C \times K \times M$
		Input Length (bits):
		$N_C \times (N–K+S–P) \times M$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (bits):
		$N_C \times S \times M$

- *N* is the codeword length.
- *K* is the message word length.
- *S* is the shortened message word length.
- N_C is the number of codewords (and message words).
- *P* is the number of punctures per codeword, and is equal to the number of zeros in the puncture vector.
- *M* is the degree of the primitive polynomial. Each group of *M* bits represents an integer between `0` and $2^M–1$ that belongs to the finite Galois field `GF(2^M)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Also, see “Restrictions on the M and the Codeword Length N” on page 2-72.

Restrictions on the M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, *N*, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, *M*, are from 3 to 16. The valid values for *N* in this case are from 7 to 2^M–1. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 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. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})...(x+\alpha^{b+N-K-1})$$

α 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 running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N` $= 2^M$`-1`.

- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

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))

Port	Supported Data Types
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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

See Also

Blocks
Binary-Output RS Decoder | Integer-Input RS Encoder

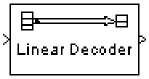
System Objects
`comm.RSEncoder`

Functions

Introduced before R2006a

Binary Linear Decoder

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 *r*th 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.

Parameters

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

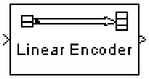
Pair Block

Binary Linear Encoder

Introduced before R2006a

Binary Linear Encoder

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.

Parameters

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

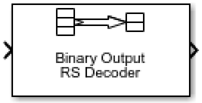
Binary Linear Decoder

Introduced before R2006a

Binary-Output RS Decoder

Decode Reed-Solomon code to recover binary vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding

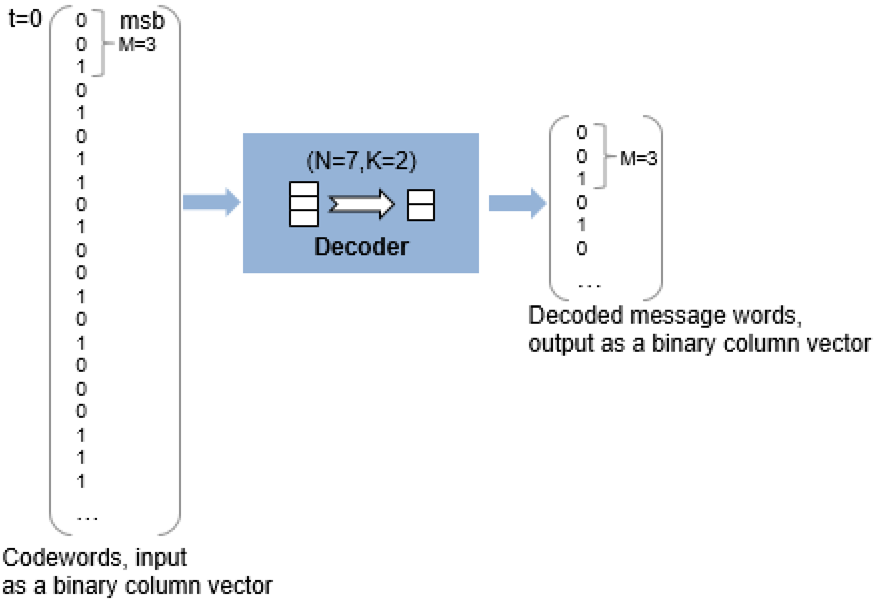


Description

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 must match parameter values in the corresponding Binary-Input RS Encoder block.

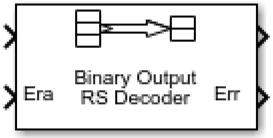
The symbols for the code are binary sequences of length *M*, corresponding to elements of the Galois field $GF(2^M)$. The first bit in each symbol is the most significant bit.

This figure shows the decoder input-output word length for codeword length N=7 and message word length K=2. Since N=2^M–1, when N=7, the symbol length, M=3.



Each input codeword is a binary vector of length 21 that represents 7 three-bit integers. Each corresponding output message word is a binary vector of length 6, that represents 2 three-bit integers. For more information, see “Input and Output Signal Length in RS Blocks” on page 2-85.

This icon shows all ports, including optional ones:



Ports

Input

In — Reed-Solomon codeword

binary column vector

Reed-Solomon codeword in bits, specified as an (N_C×(N – K + S – P)×M)-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-85.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

`Era` — Erasure vector

binary column vector

Erasure vector in symbols, specified as an (N_C×(N – K + S – P))-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

Erasure values of 1 correspond to erased bits in the same position in the codeword. Values of 0 correspond to bits that are not erased. For more information, see “Puncturing and Erasures” on page 2-87.

Dependencies

To enable this port, select **Enable erasures input port**.

Data Types: `double` | `Boolean`

Output

`Out` — Decoded message

binary column vector

Decoded message in bits, returned as one of the following:

- When there is no message shortening, a (N_C×K×M)-by-1 binary column vector.
- When there is message shortening, a (N_C×S×M)-by-1 binary column vector.

N_C is the number of message words, K is the **Message length K (symbols)**, M is the number of bits per symbol, and S is the **Shortened message length S (symbols)**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-85.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

`Err` — Decoding errors

integer vector

Symbol decoding errors, returned as an integer vector with N_C elements, where N_C is the number of codewords. This port indicates the number of symbol errors detected during decoding of each codeword. A negative integer indicates that the block detected more errors than it could correct by using the specified coding scheme.

Note An (N,K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (not bit errors) in each codeword. When a received codeword contains more than (N-K)/2 symbol errors, a decoding failure occurs.

Dependencies

To enable this port, select **Output number of corrected symbol errors**.

Data Types: `double`

For more information, see “Supported Data Types” on page 2-88.

Parameters

`Codeword length N (symbols)` — Codeword length

7 (default) | integer

Codeword length in symbols, specified as an integer.

For more information, see “Restrictions on M and Codeword Length N” on page 2-86 and “Input and Output Signal Length in RS Blocks” on page 2-85.

`Message length K (symbols)` — Message word length

3 (default) | integer

Message word length in symbols, specified as an integer from 1 to N–2, where N is the codeword length.

`Shortened message length S (symbols)` — Shortened message word length

3 (default) | integer

Shortened message word length in symbols, specified as an integer, such that $S \leq K$. When **Shortened message length S (symbols) < Message length K (symbols)**, the Reed-Solomon code is shortened.

You still specify N and K values for the full-length (N, K) code but the decoding is shortened to an (N–K+S, S) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values from 0 to 2^M–1, in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see "Character Representation of Polynomials".
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see "Specify the Generator Polynomial" on page 2-87.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Restrictions on M and Codeword Length N” on page 2-86.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 2-87.

Dependencies

To enable this parameter, select **Punctured code**.

`Enable erasures input port` — Enable erasures input port

off (default) | on

Selecting this check box enables the erasures port, **Era**. For more information, see “Puncturing and Erasures” on page 2-87.

`Output number of corrected symbol errors` — Enable port to output number of corrected symbol errors

off (default) | on

Selecting this check box enables an additional output port, **Err**, which indicates the number of symbol errors the block corrected in the input codeword.

`Output data type` — Output type of the block

`Same as input` (default) | `boolean` | `double`

Output type of the block, specified as `Same as input`, `boolean`, or `double`.

Definitions

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, *K*, or shortened message word length, *S*. The codeword length is $N – K + S – P$, where *N* is the full codeword length and *P* is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N – P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that *y* is an integer multiple of *x*.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Input RS Encoder	**Input Length (bits):**
	$N_C \times K \times M$
	Output Length (bits):
	$N_C \times (N–P) \times M$
		Input Length (bits):
		$N_C \times S \times M$
		Output Length (bits):
		$N_C \times (N–K+S–P) \times M$

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Output RS Decoder	**Input Length (bits):**
	$N_C \times (N–P) \times M$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (bits):
	$N_C \times K \times M$
		Input Length (bits):
		$N_C \times (N–K+S–P) \times M$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (bits):
		$N_C \times S \times M$

- N is the codeword length.
- K is the message word length.
- S is the shortened message word length.
- N_C is the number of codewords (and message words).
- P is the number of punctures per codeword, and is equal to the number of zeros in the puncture vector.
- M is the degree of the primitive polynomial. Each group of M bits represents an integer between `0` and `2^M—1` that belongs to the finite Galois field `GF(2^M)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Also, see “Restrictions on M and Codeword Length N” on page 2-86.

Restrictions on M and Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, N, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, M, are from 3 to 16. The valid values for N in this case are from 7 to 2^M–1.

Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 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. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})\ldots(x+\alpha^{b+N-K-1})$

α 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 running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.

- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

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))
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point

Pair Block

Binary-Input RS Encoder

Algorithms

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see “Algorithms for BCH and RS Errors-only Decoding”.

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.

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

See Also

Blocks
Binary-Input RS Encoder | Integer-Output RS Decoder

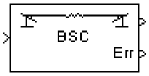
System Objects
`comm.RSDecoder`

Functions
`primpoly` | `rsdec` | `rsgenpoly`

Introduced before R2006a

Binary Symmetric Channel

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**.

Parameters

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.

Examples

An example using the Binary Symmetric Channel block is in the section “Design a Rate 2/3 Feedforward Encoder Using Simulink”.

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

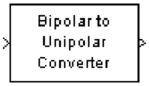
See Also

Bernoulli Binary Generator

Introduced before R2006a

Bipolar to Unipolar Converter

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

Parameters

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** (Simulink) 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

Introduced before R2006a

Bit to Integer Converter

Map vector of bits to corresponding vector of integers

Library

Utility Blocks

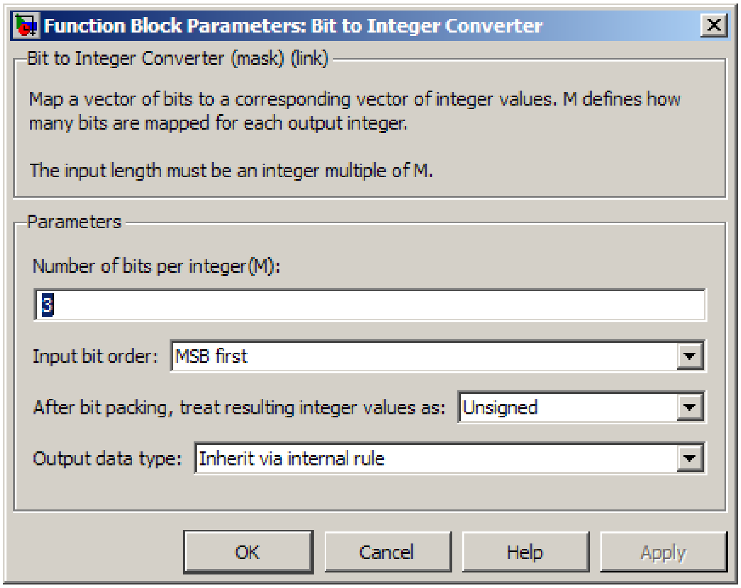
Description

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



Function Block Parameters: Bit to Integer Converter

Bit to Integer Converter (mask) (link)

Map a vector of bits to a corresponding vector of integer values. M defines how many bits are mapped for each output integer.

The input length must be an integer multiple of M.

Parameters

Number of bits per integer(M): 3

Input bit order: MSB first

After bit packing, treat resulting integer values as: Unsigned

Output data type: Inherit via internal rule

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** (Simulink) 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) on page 2-495

See Also

`bi2de`, `bin2dec`

Pair Block

Integer to Bit Converter

Introduced before R2006a

BPSK Demodulator Baseband

Demodulate BPSK-modulated data

Library

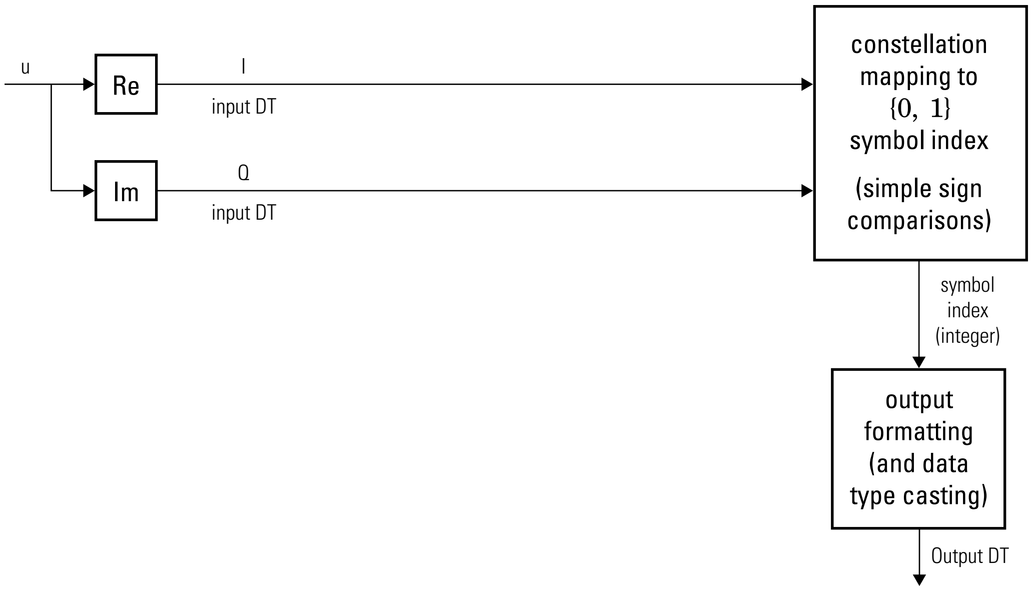
PM, in Digital Baseband sublibrary of Modulation

Description

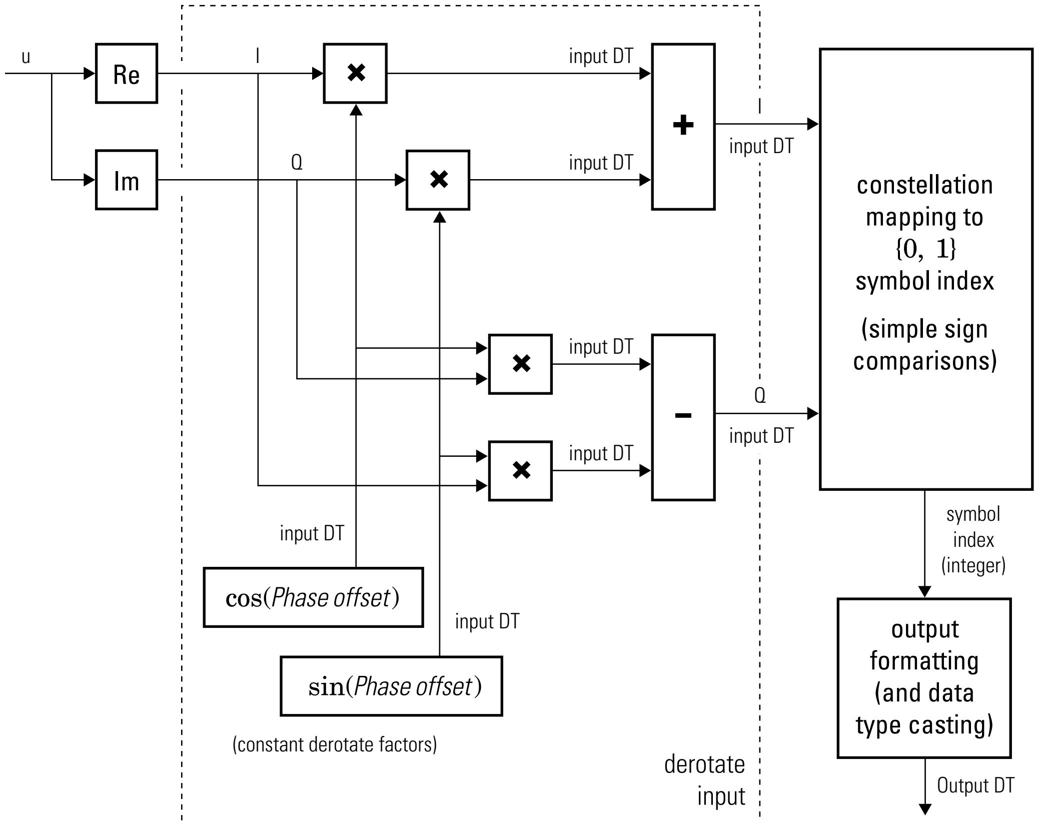
The BPSK Demodulator Baseband block demodulates a signal that was modulated using the binary phase shift keying method. The input is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal. The input signal must be be a discrete-time complex signal. 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 2-108.

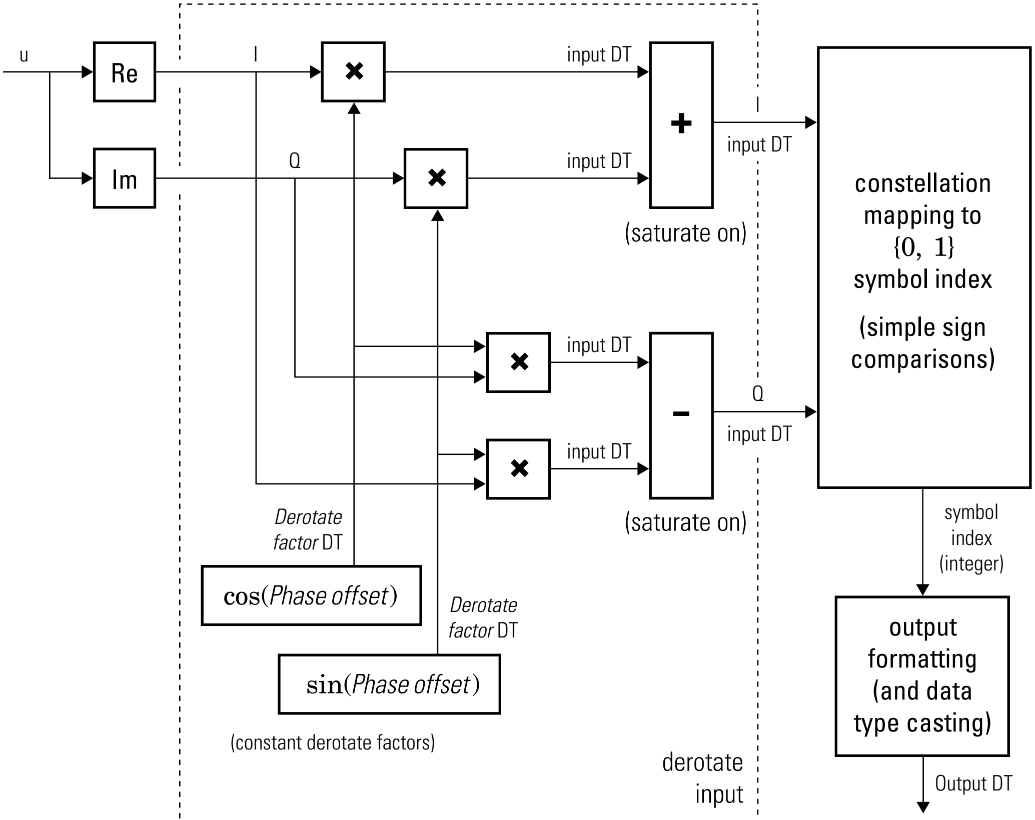
Algorithm


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)



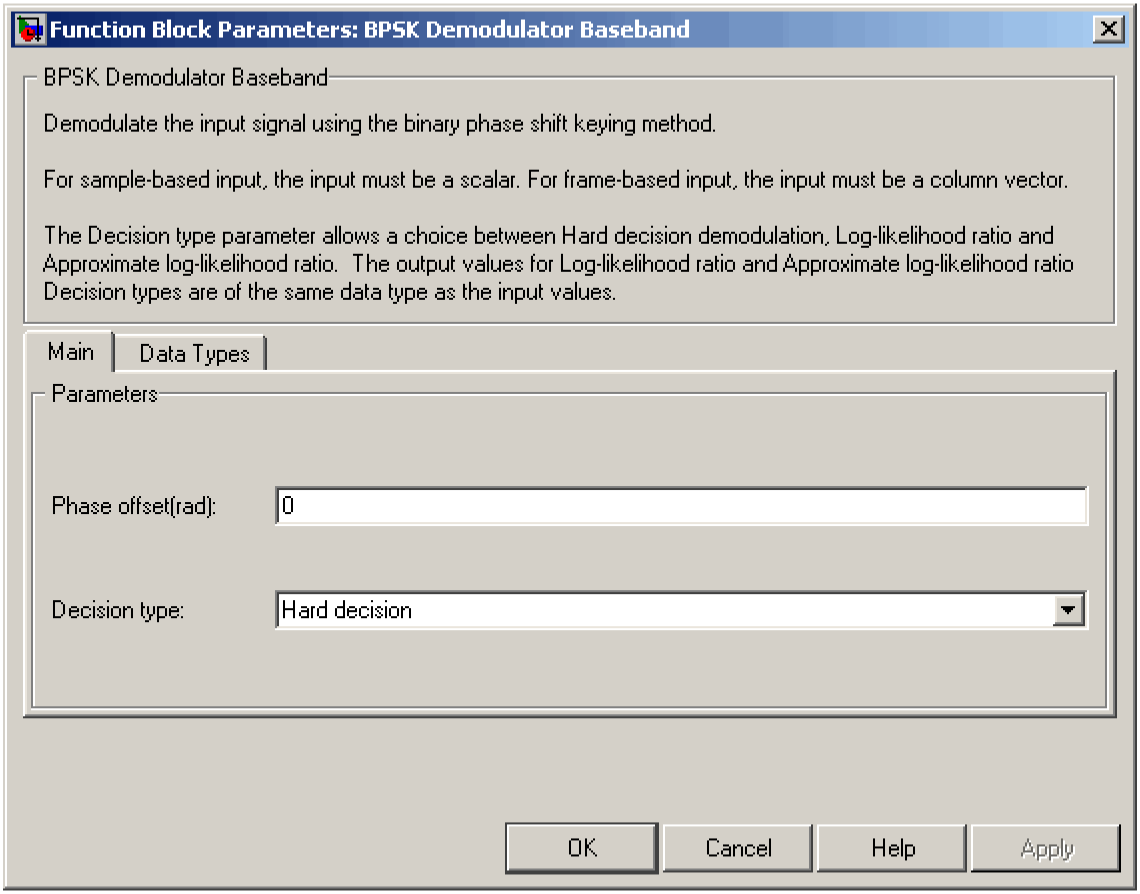
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



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



Function Block Parameters: BPSK Demodulator Baseband

BPSK Demodulator Baseband

Demodulate the input signal using the binary phase shift keying method.

For sample-based input, the input must be a scalar. For frame-based input, the input must be a column vector.

The Decision type parameter allows a choice between Hard decision demodulation, Log-likelihood ratio and Approximate log-likelihood ratio. The output values for Log-likelihood ratio and Approximate log-likelihood ratio Decision types are of the same data type as the input values.

Phase offset(rad): 0

Decision type: Hard decision

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

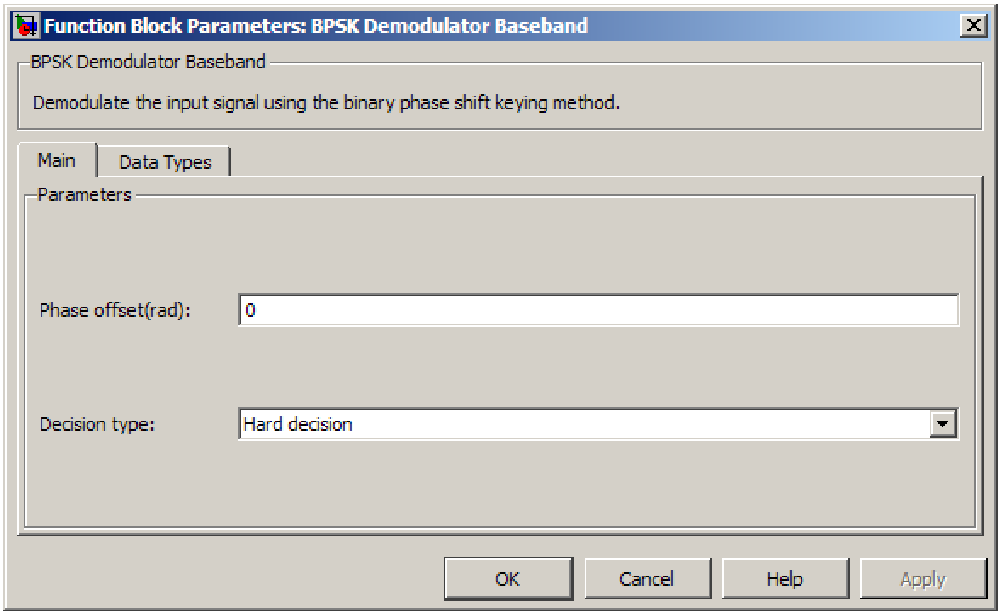
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

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.



Function Block Parameters: BPSK Demodulator Baseband

BPSK Demodulator Baseband

Demodulate the input signal using the binary phase shift keying method.

Parameters

Phase offset(rad): 0

Decision type: Hard decision

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`.

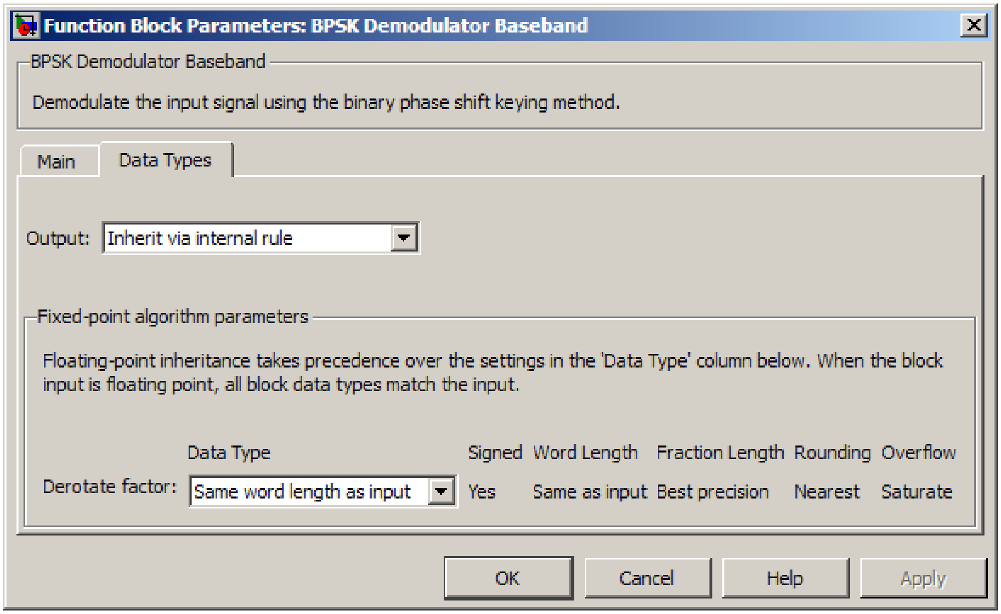
When this parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (`single` or `double`). If the input data type is fixed-point, 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 π/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.



Function Block Parameters: BPSK Demodulator Baseband

BPSK Demodulator Baseband

Demodulate the input signal using the binary phase shift keying method.

Output: Inherit via internal rule

Fixed-point algorithm parameters

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

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Derotate factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate

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

HDL Code Generation

This block supports HDL code generation using HDL Coder™. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see BPSK Demodulator Baseband in the HDL Coder documentation.

Pair Block

BPSK Modulator Baseband

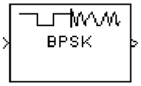
See Also

M-PSK Demodulator Baseband, QPSK Demodulator Baseband, DBPSK Demodulator Baseband

Introduced before R2006a

BPSK Modulator Baseband

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.

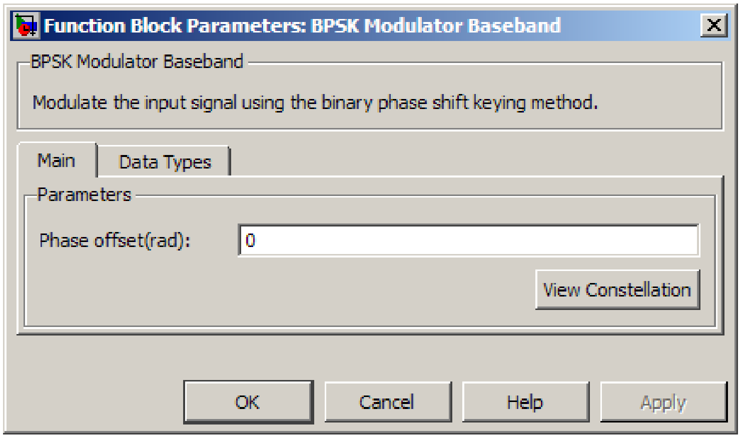
This block accepts a column vector input signal. The input must be a discrete-time binary-valued signal. If the input bit is 0 or 1, respectively, then the modulated symbol is exp(jθ) or -exp(jθ), respectively, where θ represents the **Phase offset** parameter.

For information about the data types each block port supports, see the “Supported Data Types” on page 2-113 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


Function Block Parameters: BPSK Modulator Baseband

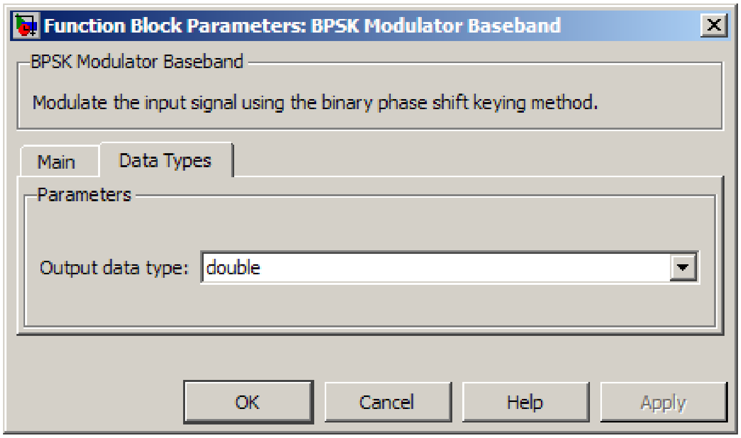
BPSK Modulator Baseband

Modulate the input signal using the binary phase shift keying method.

Phase offset(rad): 0

Phase offset (rad)

The phase of the zeroth point of the signal constellation.


Function Block Parameters: BPSK Modulator Baseband

BPSK Modulator Baseband

Modulate the input signal using the binary phase shift keying method.

Output data type: double

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)

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see BPSK Modulator Baseband in the HDL Coder documentation.

Pair Block

BPSK Demodulator Baseband

See Also

M-PSK Modulator Baseband, QPSK Modulator Baseband, DBPSK Modulator Baseband

Introduced before R2006a

Carrier Synchronizer

Compensate for carrier frequency offset

Carrier Synchronizer
Out
PhEst
Carrier Synchronizer

Library

Synchronization

Description

The Carrier Synchronizer block compensates for carrier frequency and phase offsets using a closed-loop approach for BPSK, QPSK, OQPSK, 8-PSK, QAM, and PAM modulation schemes. The block accepts a single input port. To obtain an estimate of the phase error in radians, select the **Estimated phase error output port** check box. The block accepts a sample- or frame-based complex input signal and returns a complex output signal and a an real phase estimate. The block outputs have the same dimensions as the input.

Parameters

Modulation

Specify the modulation type as `BPSK`, `QPSK`, `OQPSK`, `8PSK`, `QAM`, or `PAM`.

This object supports CPM. It has been tested for a CPM signal having 1 sample per symbol and a modulation index of 0.5.

Modulation phase offset

Specify the method used to calculate the modulation phase offset as either `Auto` or `Custom`.

- `Auto` applies the traditional offset for the specified modulation type.

Modulation	Phase Offset
`BPSK`	0
`QPSK` or `OQPSK`	$\pi/4$
`8PSK`	$\pi/8$
`QAM` or `PAM`	0

- `Custom` enables the **Custom phase offset (radians)** parameter.

Custom phase offset (radians)

Specify the phase offset in radians as a real scalar. This parameter is available only when **Modulation phase offset** is set to `Custom`.

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar.

Damping factor

Specify the damping factor of the loop as a positive real finite scalar.

Normalized loop bandwidth

Specify the normalized loop bandwidth as a real scalar between 0 and 1. The bandwidth is normalized by the sample rate of the carrier synchronizer block.

Estimated phase error output port

Select this check box to provide the estimated phase error to an output port.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, system objects accept a maximum of nine inputs.

`Interpreted execution`

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Algorithms

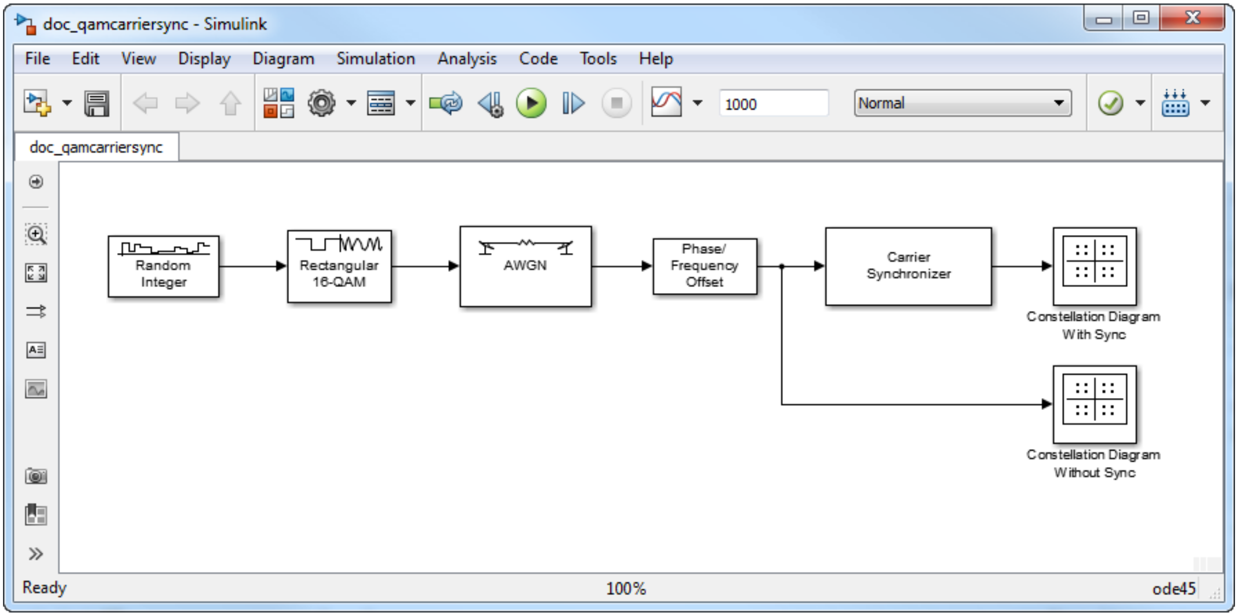
This block implements the algorithm, inputs, and outputs described on the `comm.CarrierSynchronizer` reference page. The object properties correspond to the block parameters.

Examples

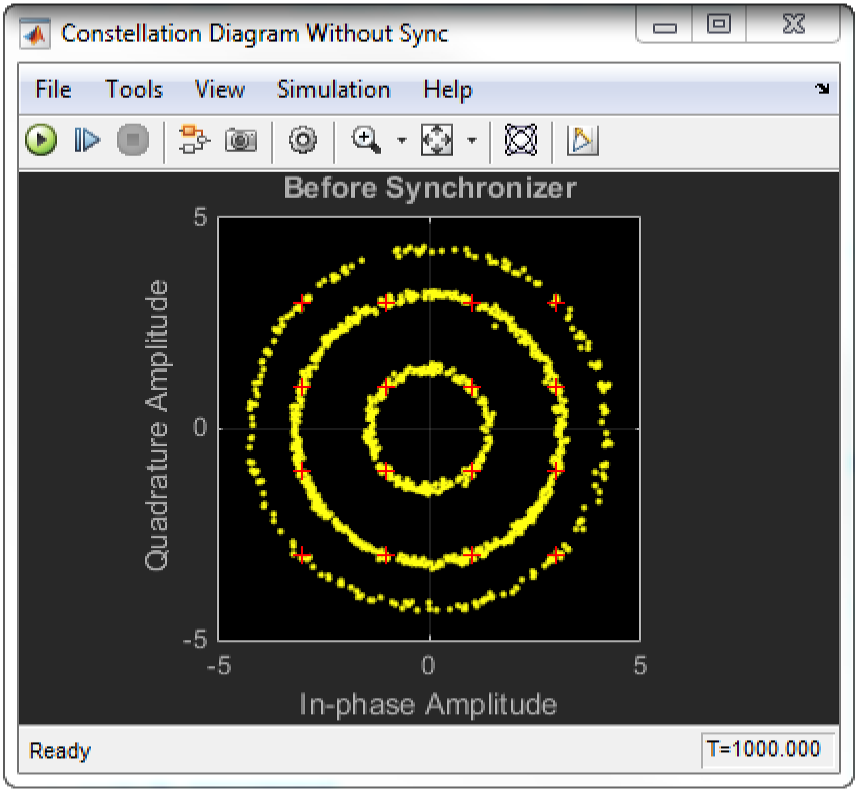
Correct for Frequency and Phase Offset

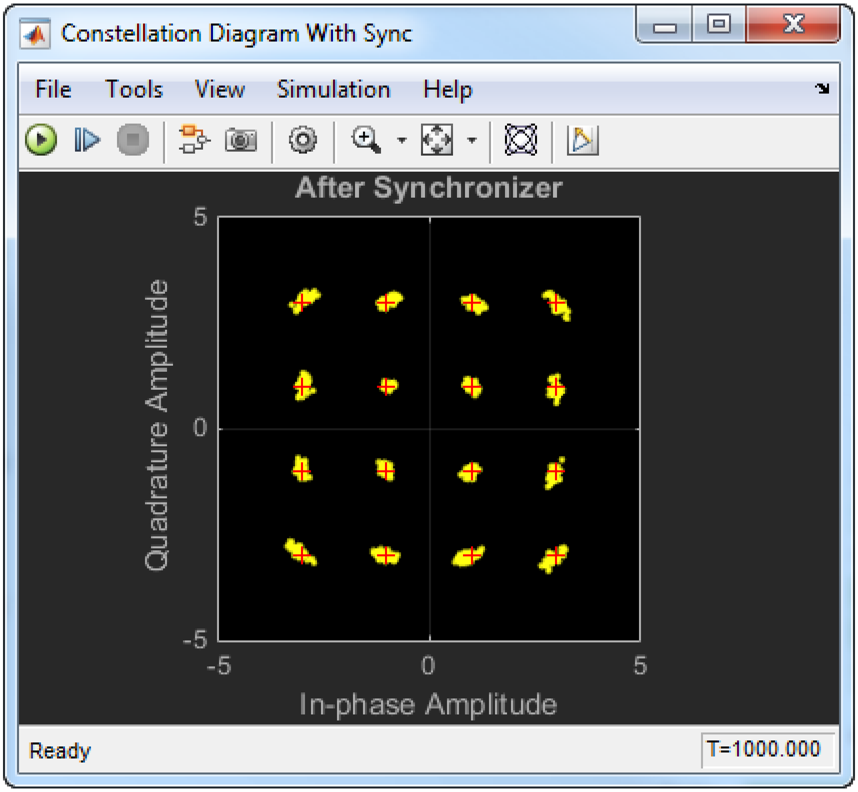
Correct for a phase and frequency offset imposed on a noisy 16-QAM channel using the Carrier Synchronizer block.

Open the doc_qamcarriersync model.



Run the model. The Constellation Diagram Without Sync block shows a spiral pattern that indicates a phase and frequency offset. After the carrier synchronizer converges to a solution, the data displayed on the Constellation Diagram With Sync block are grouped around the reference constellation.


Constellation Diagram Without Sync
File
Tools
View
Simulation
Help
Before Synchronizer
5
0
-5
Quadrature Amplitude
-5
0
5
In-phase Amplitude
Ready
T=1000.000



Experiment with the parameters in the Phase/Frequency Offset and Carrier Synchronizer blocks. By varying these parameters, you can change how quickly the output conforms to an ideal 16-QAM constellation.

Supported Data Types

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

Port	Supported Data Types
Phase Error Estimate	• Double-precision floating point • Single-precision floating point

See Also

- `comm.CarrierSynchronizer`
- Biquad Filter
- “MSK Signal Recovery”

Selected Bibliography

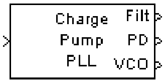
[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 359–393.

[2] Huang, Zhijie, Zhiqiang Yi, Ming Zhang, and Kuang Wang. “8PSK Demodulation for New Generation DVB-S2.” *International Conference on Communications, Circuits and Systems, 2004. ICCCAS 2004.* Vol. 2, 2004, pp. 1447–1450.

Introduced in R2015a

Charge Pump PLL

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.

Parameters

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

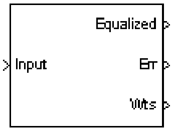
References

For more information about digital phase-locked loops, see the works listed in“Selected Bibliography for Synchronization” in *Communications System Toolbox User*'*s Guide*.

Introduced before R2006a

CMA Equalizer

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 input frame or sample.	Select **Output weights**.

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.

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.

Parameters

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.

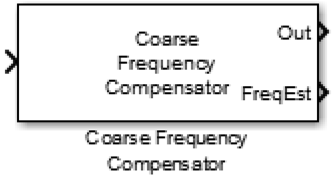
See Also

LMS Linear Equalizer, LMS Decision Feedback Equalizer, RLS Linear Equalizer, RLS Decision Feedback Equalizer

Introduced before R2006a

Coarse Frequency Compensator

Compensate for carrier frequency offset for PAM, PSK, or QAM



Library

Synchronization

Description

The Coarse Frequency Compensator block compensates for a carrier frequency offset for BPSK, QPSK, OQPSK, 8-PSK, PAM, and QAM modulation schemes. The block accepts a single input signal. To obtain an estimate of the frequency offset in Hz, select the **Estimated frequency offset output port** check box. The block accepts a sample- or frame-based complex input signal and returns a complex output signal and a real frequency offset estimate. The output signal has the same dimensions as the input signal. The frequency offset estimate is a scalar.

Parameters

Modulation type of input signal

Specify the modulation type as `BPSK`, `QPSK`, `OQPSK`, `8PSK`, `PAM`, or `QAM`.

The default setting is `QAM`.

Estimation algorithm

Specify the frequency offset estimation algorithm as `FFT-based` or `Correlation-based`. This parameter appears when **Modulation type of input signal** is `BPSK`, `QPSK`, `8PSK`, or `PAM`.

The table shows the allowable combinations of the modulation type and the estimation algorithm.

Modulation	FFT-Based Algorithm	Correlation-Based Algorithm
BPSK, QPSK, 8PSK, PAM	✓	✓
OQPSK, QAM	✓	

Frequency resolution (Hz)

Specify the frequency resolution in Hz as a positive real scalar. This option is available when the `FFT-based` algorithm is used. The default setting is `0.001` Hz.

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar greater than or equal to 4. The default setting is 4.

Maximum frequency offset (Hz)

Specify the maximum frequency offset in Hz as a positive real scalar. This option is appears when you set **Estimation algorithm** to `Correlation-based`. The default setting is `0.05` Hz.

Estimated frequency offset output port

Select this check box to provide the estimated frequency offset to an output port. The default for this parameter is selected.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects accept a maximum of nine inputs.

`Interpreted execution`

Simulate your model using all supported MATLAB functions. Choosing this option can slow simulation performance.

The default setting is `Code generation`.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.CoarseFrequencyCompensator` reference page. The object properties correspond to the block parameters.

Examples

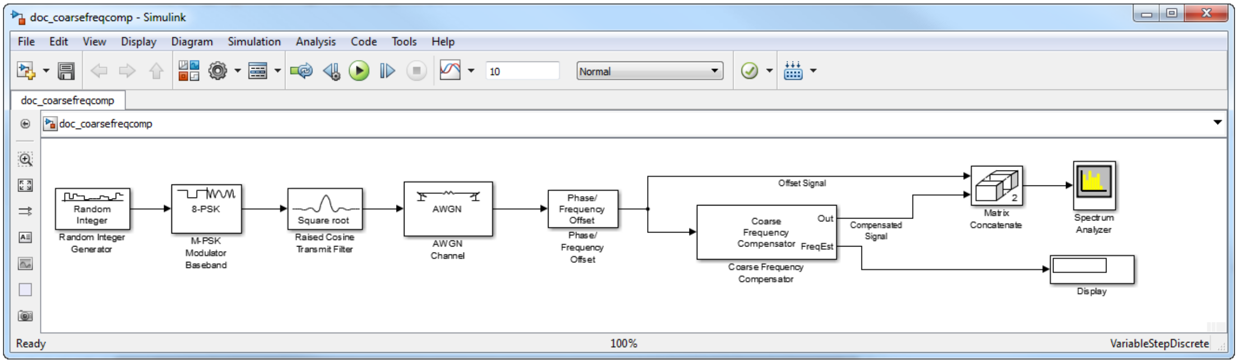
Correct for Frequency and Phase Offset

Correct for a frequency offset imposed on a noisy 8-PSK channel by using the Coarse Frequency Compensator block.

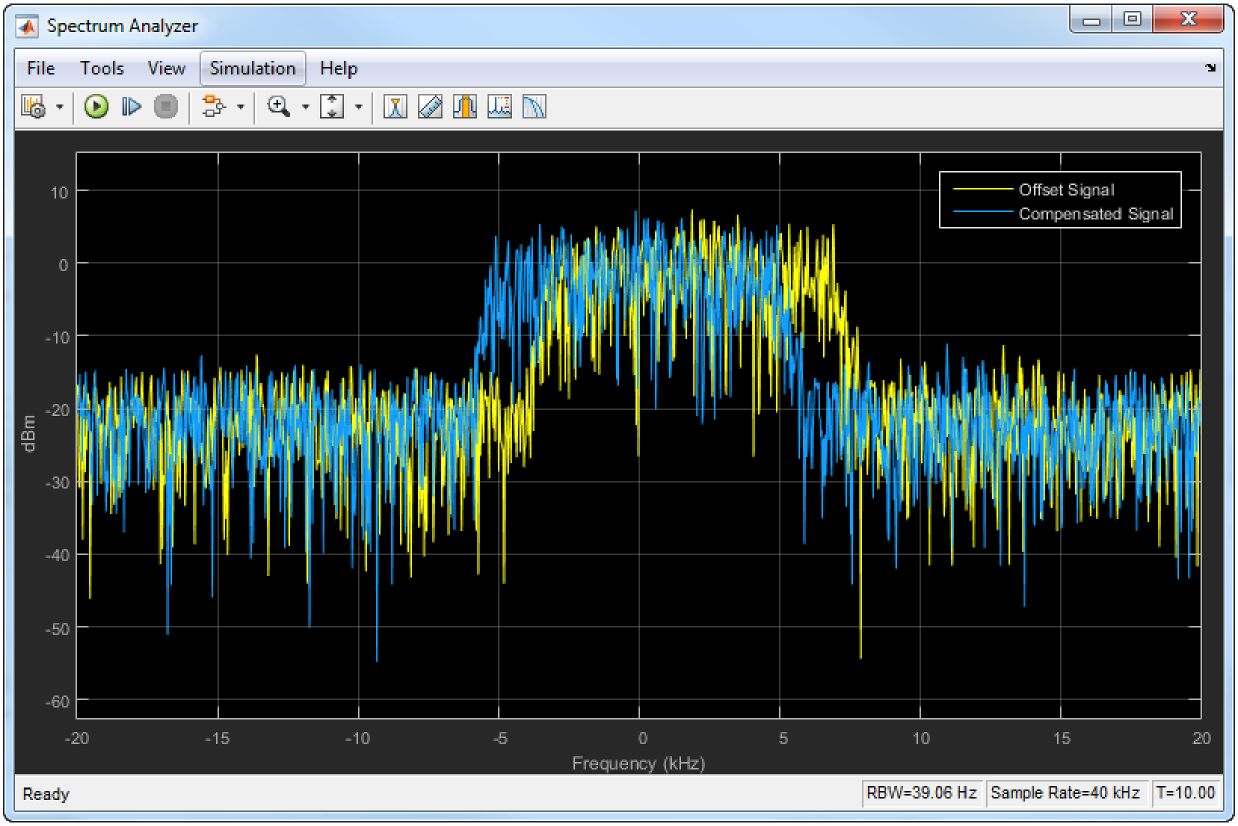
Open the doc_coarsefreqcomp model.

Open the dialog boxes to verify these parameter values:

- Random Integer Generator — **Sample time** is `1e-4`, which is equivalent to a 10 ksym/sec symbol rate.
- Raised Cosine Transmit Filter — **Output samples per symbol** is `4`.
- AWGN Channel — **Mode** is `Signal to noise ratio (SNR)` and **SNR (dB)** is `20`.
- Phase/Frequency Offset — **Frequency offset (Hz)** is `2000`.
- Coarse Frequency Compensator — **Estimation algorithm** is `FFT-based` and **Frequency resolution (Hz)** is `1`.



Run the model. The Spectrum Analyzer block shows both the frequency offset signal and the compensated signal. In addition, the Display block shows the estimate of the frequency offset. Observe that the spectrum plot shows that the Coarse Frequency Compensator correctly centers the signal around 0 Hz. Additionally, the display shows that the estimated frequency offset is 2000 Hz.



Adjust the parameters in the Phase/Frequency Offset and Coarse Frequency Compensator blocks and see their effect on frequency compensation performance.

Supported Data Types

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

References

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions.” *IEEE® Transactions on Communications*.Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

[2] Wang, Y., K. Shi, and E. Serpedi. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach.” *EURASIP Journal on Applied Signal Processing*. 2004:13, pp. 1993–2001.

[3] Nakagawa, T., M. Matsui, T. Kobayashi, K. Ishihara, R. Kudo, M. Mizoguchi, and Y. Miyamoto. “Non-Data-Aided Wide-Range Frequency Offset Estimator for QAM Optical Coherent Receivers.” *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*. March 2011, pp. 1–3.

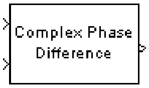
See Also

Carrier Synchronizer | Symbol Synchronizer | `comm.CoarseFrequencyCompensator`

Introduced in R2015b

Complex Phase Difference

Output phase difference between two complex input signals



Library

Utility Blocks

Description

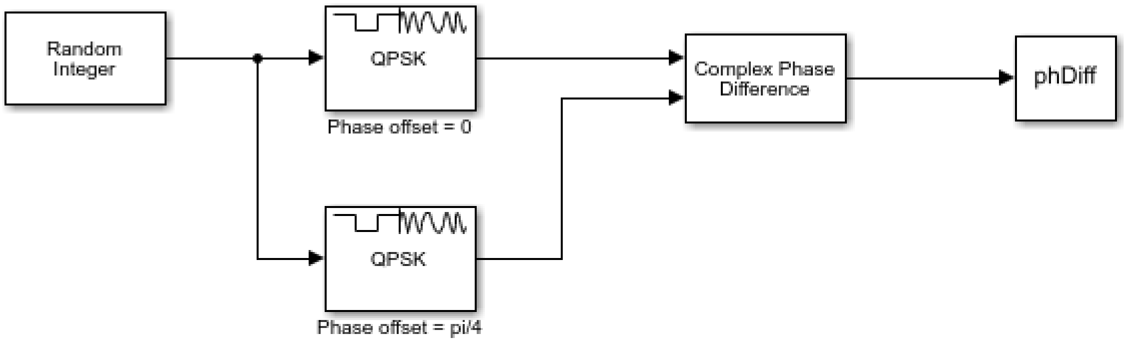
The Complex Phase Difference block accepts two complex input signals that have the same size and frame status. The output is the phase difference from the second to the first, measured in radians. The elements of the output are between -π and π.

The input signals can have any size or frame status. This block processes each pair of elements independently.

Examples

Calculate Complex Phase Difference

Open the complex phase difference model. The model generates random integers and applies QPSK modulation. The first QPSK modulator has a phase offset of 0, while the second has a pi/4 phase offset. The `Complex Phase Difference` block determines the phase difference. The data is passed to the workspace from the `To Workspace` block.



Run the model.

Display the phase difference.

```
phDiff =

   -0.7854
```

The phase difference is equal to -pi/4 as expected.

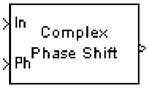
See Also

Complex Phase Shift

Introduced before R2006a

Complex Phase Shift

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.

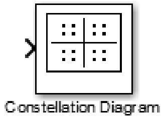
See Also

Complex Phase Difference

Introduced before R2006a

Constellation Diagram

Display constellation diagram for input signals



Library

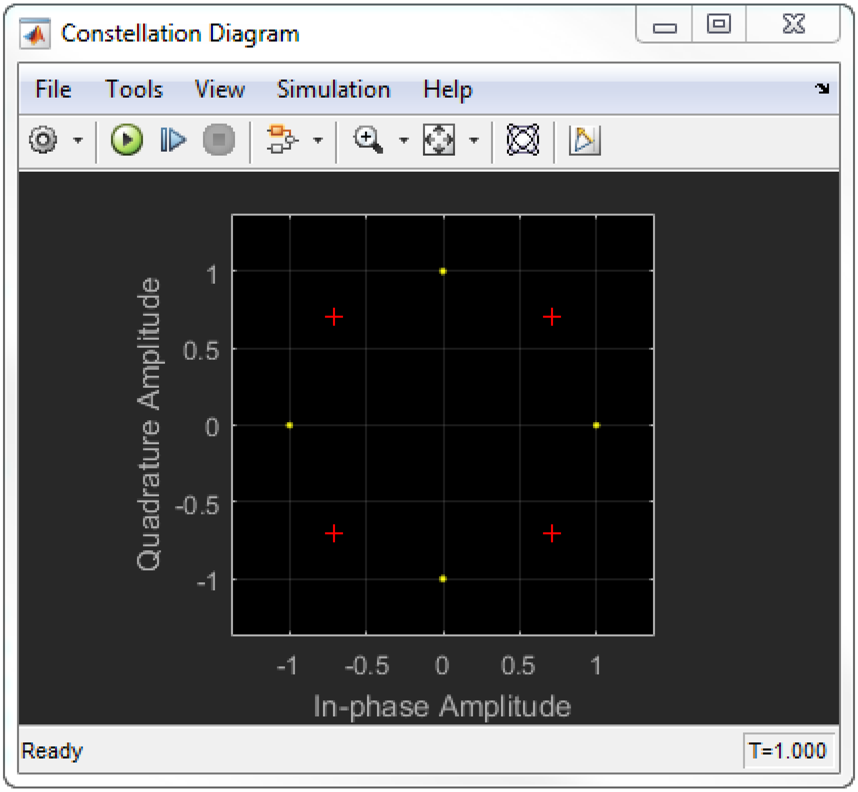
Comm Sinks

Description

The Constellation Diagram block plots constellation diagrams, signal trajectory, and provides the ability to perform EVM and MER measurements.

The symbols that the Constellation Diagram scope displays are always the most recently available symbols from the time buffer.

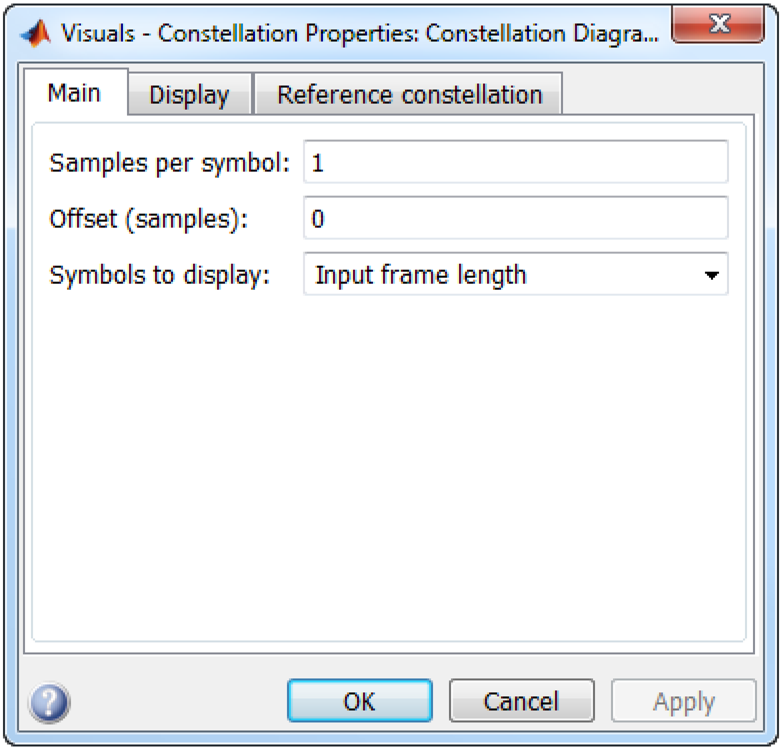
Dialog Box



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** and **Symbols to display** parameters on the **Main** tab. You can modify the reference constellation parameters on the **Reference constellation** tab.

Visuals — Constellation Properties

Main Pane



Samples per symbol: 1

Offset (samples): 0

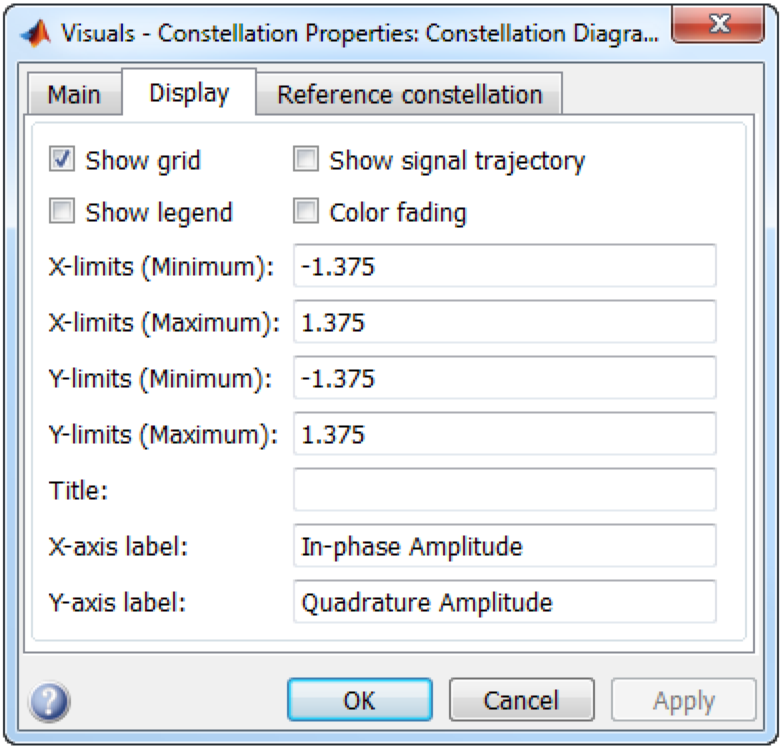
Symbols to display: Input frame length

Number of samples used to represent a symbol. This value must be a positive number.

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.

The maximum number of symbols that can be displayed. Must be a positive integer value.

Display Pane



Visuals - Constellation Properties: Constellation Diagra...

☑ Show grid □ Show signal trajectory

□ Show legend □ Color fading

X-limits (Minimum): -1.375

X-limits (Maximum): 1.375

Y-limits (Minimum): -1.375

Y-limits (Maximum): 1.375

Title:

X-axis label: In-phase Amplitude

Y-axis label: Quadrature Amplitude

Select this check box to turn on the grid.

Select this check box to display a legend for the graph.

Select this check box to display the trajectory of a modulated signal by plotting its in-phase component versus its quadrature component.

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.

Specify the minimum value of the x-axis.

Specify the maximum value of the x-axis.

Specify the minimum value of the y-axis.

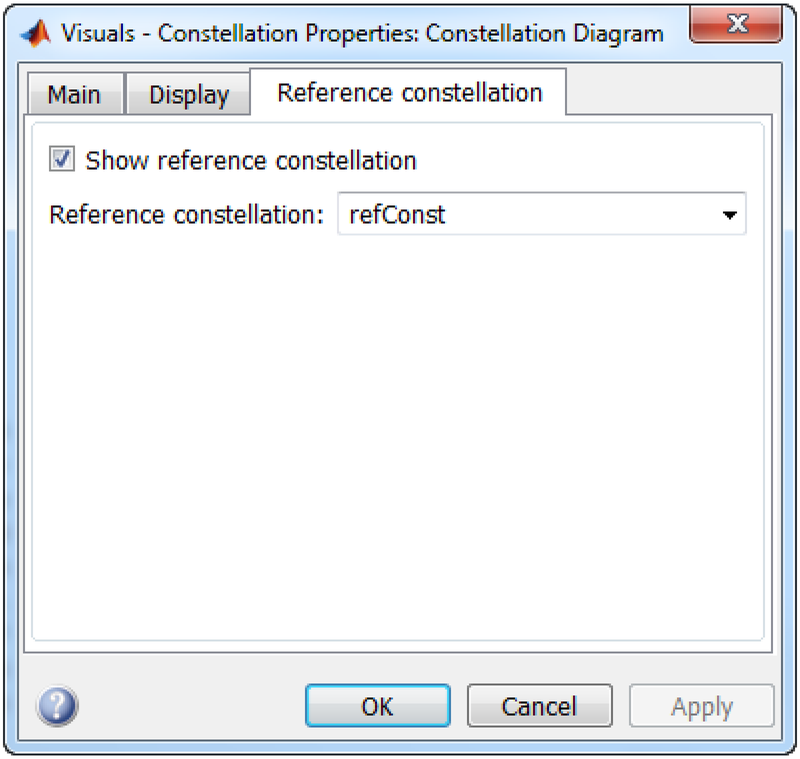
Specify the maximum value of the y-axis.

Specify a label that appears above the constellation diagram plot. By default, there is no title.

Specify the text the scope displays along the x-axis

Specify the text the scope displays along the y-axis

Reference Constellation Pane



Visuals - Constellation Properties: Constellation Diagram

☑ Show reference constellation

Reference constellation: refConst

Select the check box to display the reference constellation.

Select the reference constellation from `BPSK` | `QPSK` | `8-PSK` | `16-QAM` | `64-QAM` | `256-QAM` | `<user-defined>`. If not selected, the reference constellation is specified in the variable `refConst`.

Select the type of constellation normalization as `Minimum distance`, `Average power`, or `Peak power`.

Specify the minimum distance between symbols in the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Minimum distance`.

Specify the average power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Average power`.

Specify the peak power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Peak power`.

Specify the phase offset of the reference constellation in radians as a real scalar.

Measurement Panels

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

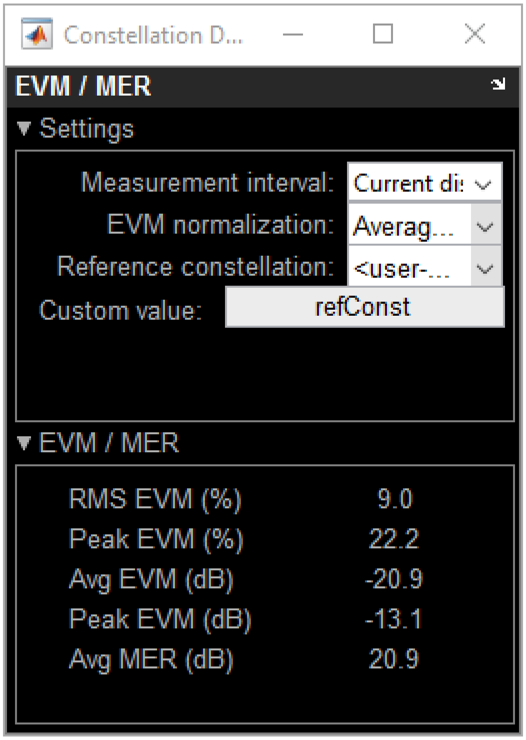
Button	Description
⤒	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
▼	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide 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 controls the Settings and Signal Quality panes. Both panels can be independently expanded or collapsed.



EVM / MER

▼ Settings

Measurement interval: Current di:

EVM normalization: Averag...

Reference constellation: <user-...

Custom value: refConst

▼ EVM / MER

RMS EVM (%)	9.0
Peak EVM (%)	22.2
Avg EVM (dB)	-20.9
Peak EVM (dB)	-13.1
Avg MER (dB)	20.9

You can choose to hide or display the **Signal Quality** panel. In the Scope menu, select **Tools** > **Measurements** > **Signal Quality**.

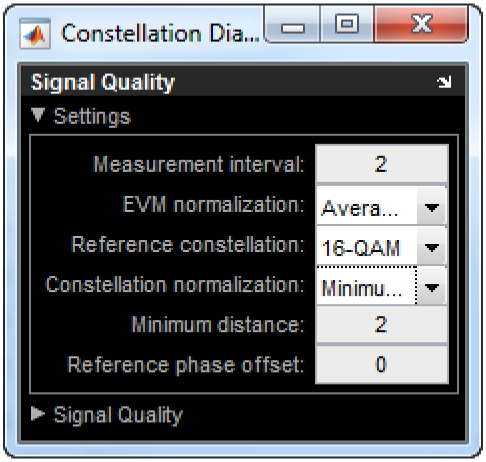
Settings Pane

The **Settings** pane enables you to define the measurement interval and normalization method the scope uses when obtaining signal measurements.

- **Measurement interval** — The duration of the EVM or MER measurement, specified as `'Current Display'`, `'All displays'`, or as a positive integer. The value of this property must be greater than one and less than or equal to the setting for the number of symbols to display. The measurement is computed after the number of input data samples exceeds the measurement interval.
- **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`.

- **Reference constellation** — Select the reference constellation as `BPSK` | `QPSK` | `8-PSK` | `16-QAM` | `64-QAM` | `256-QAM` | `<user-defined>`.
- **Constellation normalization** — Select the type of constellation normalization as `Minimum distance`, `Average power`, or `Peak power`.
- **Minimum distance** — Specify the minimum distance between symbols in the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Minimum distance`.
- **Average reference power** — Specify the average power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Average power`.
- **Peak reference power** — Specify the peak power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Peak power`.
- **Reference phase offset** — Specify the phase offset of the reference constellation in radians as a real scalar.



Measurement interval: 2

EVM normalization: Avera...

Reference constellation: 16-QAM

Constellation normalization: Minimu...

Minimum distance: 2

Reference phase offset: 0

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. The root mean square error vector magnitude, EVM_{RMS}, is measured for the average and peak constellation power.

  On the constellation diagram, you can display the EVM_{RMS} measurements normalized by the `Average constellation power` or `Peak constellation power` as computed using these algorithms.

EVM Normalization Method	Algorithm
`Average constellation power`	EVM_{RMS} in percent for average constellation power normalization $$EVM_{RMS}(percent) = \sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{avg}}} * 100$$
`Peak constellation power`	EVM_{RMS} in percent for peak constellation power normalization $$EVM_{RMS}(percent) = \sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\max}}} *100$$

The display shows the average and peak EVM_{RMS} in percent and in decibels. The EVM reported in decibels is computed as EVM (dB) = 10*log10(EVM_{MS}) = 20*log10(EVM_{RMS})

Where:

- $$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$
- I_k = In-phase measurement of the k^th symbol in the burst
- Q_{k} = Quadrature phase measurement of the k^th 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.
- EVM_{RMS} = sqrt(EVM_{MS})

The max EVM is the maximum EVM value in a frame or $EVM_{\max} = \max\limits_{k\in[1,\ldots,N]}\{EVM_k\}$, where k is the kth symbol in a burst of length N.

EVM Normalization	Algorithm
`Average constellation power`	Average constellation power normalization $EVM_k = \sqrt{\dfrac{e_k}{P_{avg}}} * 100$
`Peak constellation power`	Peak constellation power normalization $EVM_k = \sqrt{\dfrac{e_k}{P_{\max}}} * 100$

For more information, see `comm.EVM`.

- **MER** — MER is the ratio of the average power of the transmitted signal to the average power of the error vector. The scope indicates the measurement result in decibels.

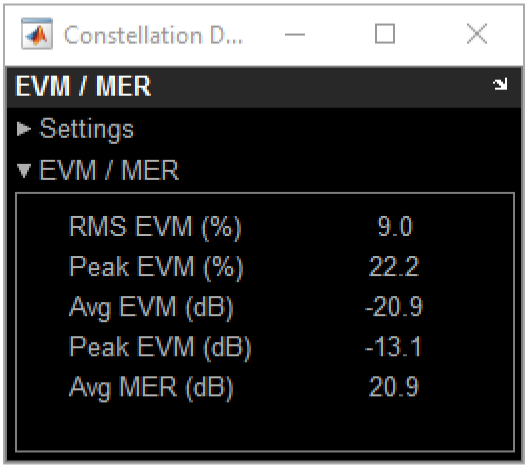
  MER is a measure of the SNR in a modulated signal calculated in dB. The MER over N symbols is

$$MER = 10 * \log_{10}\left(\frac{\sum\limits_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum\limits_{n=1}^{N}\left(e_k\right)}\right) \text{dB.}$$

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.

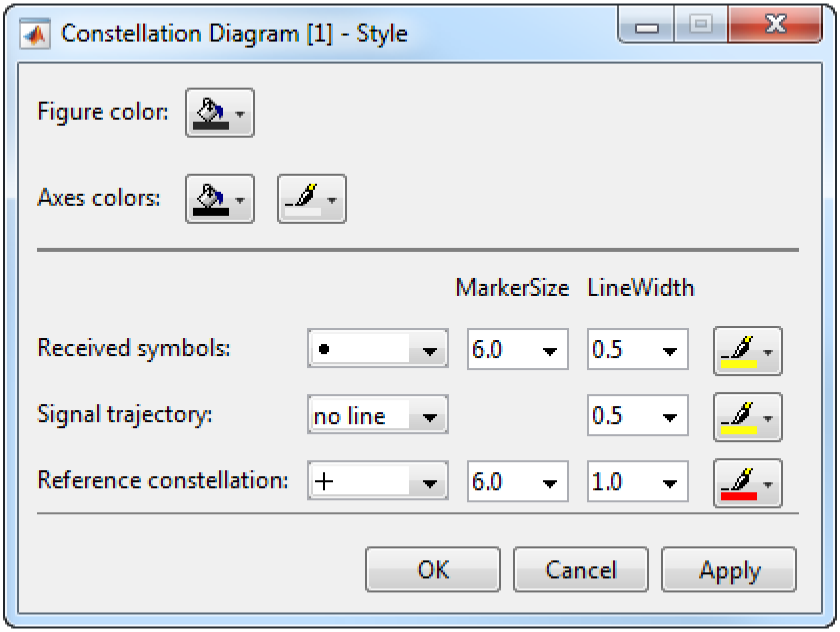
For more information, see `comm.MER`.



```
RMS EVM (%)     9.0
Peak EVM (%)    22.2
Avg EVM (dB)    -20.9
Peak EVM (dB)   -13.1
Avg MER (dB)    20.9
```

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.



Constellation Diagram [1] - Style

Figure color:

Axes colors:

		MarkerSize	LineWidth
Received symbols:	•	6.0	0.5
Signal trajectory:	no line		0.5
Reference constellation:	+	6.0	1.0

Properties

The **Style** dialog box allows you to modify the following elements of the scope figure:

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Specify the color that you want to apply to the background of the axes for the active display. Using a second drop down, you can also specify the color of the ticks, labels, and grid lines.

Specify the marker shape, marker size, marker line width, and color for the signal on the active display. The marker shape cannot be set to `none` unless the `ShowTrajectory` property is `true`.

Specify the line type, width, and color for the signal trajectory plot. The line type can only be set to something other than `no line` when the `ShowTrajectory` property is `true`.

Conversely, the line type must be `no line` when the `ShowTrajectory` property is `false`.

Specify the marker shape, marker size, marker line width, and color for the reference constellation shown on the active display. These settings are only applicable when the `ShowReferenceConstellation` property is `true`.

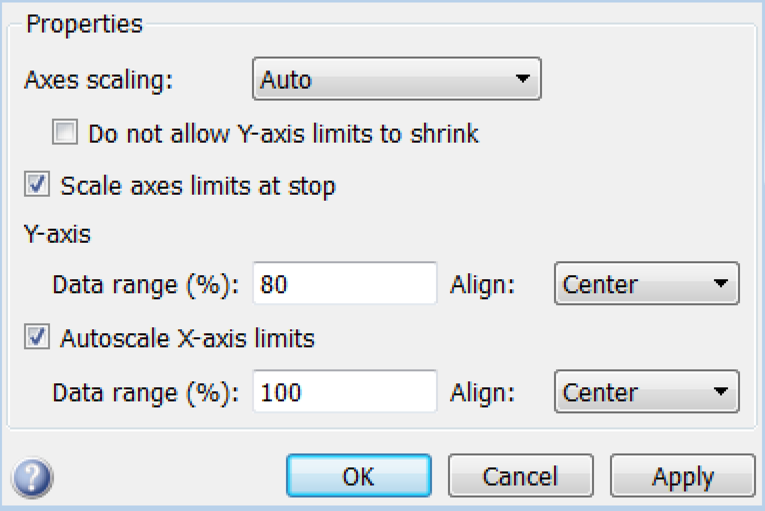
Specify the markers for the selected signal and the reference constellation on the active display. This parameter is similar to the `Marker` property for the MATLAB Handle Graphics® plot objects.

Specifier	Marker Type
none	No marker
○	Circle
□	Square
×	Cross
●	Point (default)
+	Plus sign
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Tools: Plot Navigation Properties

Properties

The Tools—Axes Scaling Properties dialog box appears as follows.



Properties

Axes scaling: Auto

☐ Do not allow Y-axis limits to shrink

☑ Scale axes limits at stop

Y-axis

Data range (%): 80 Align: Center

☑ Autoscale X-axis limits

Data range (%): 100 Align: Center

Specify when the scope automatically scales 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** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - 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. This option is useful and more efficient when your scope

display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire *x*-axis range. If you then set this property 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** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Examples

View Constellation Diagram

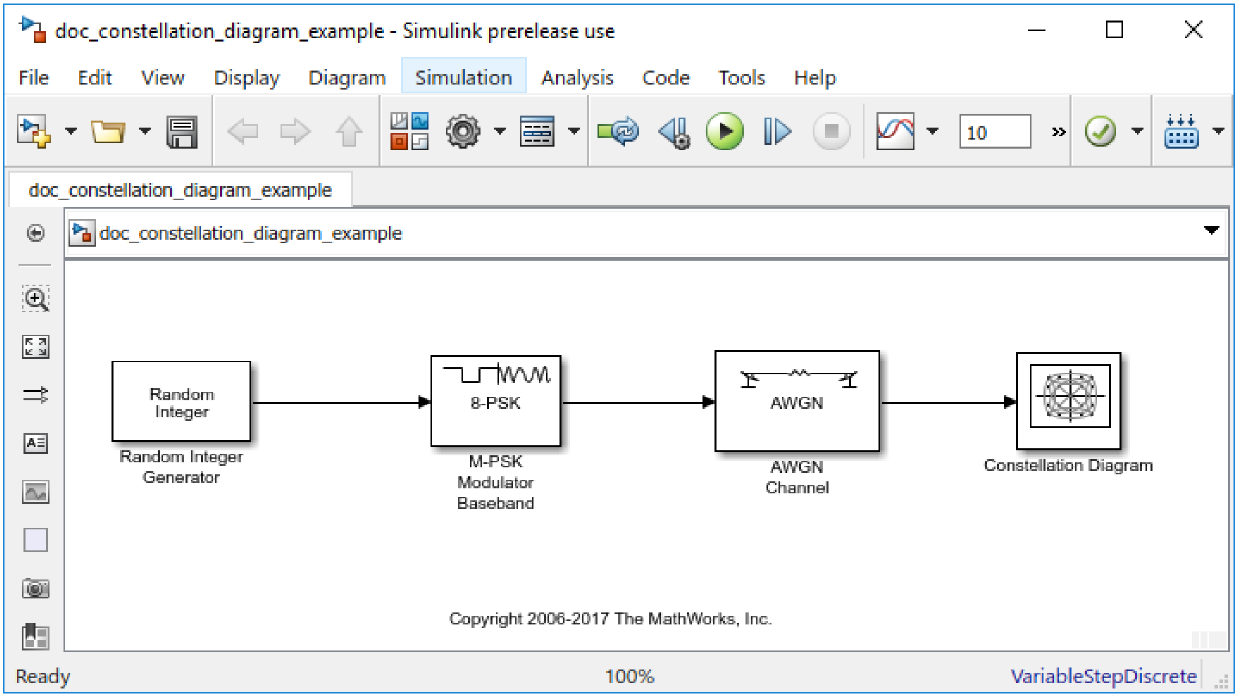
This example shows how to use the Constellation Diagram block to visualize the constellation or scatter plot of a modulated signal.

Open the model, doc_constellation_diagram_example, from the MATLAB command prompt.

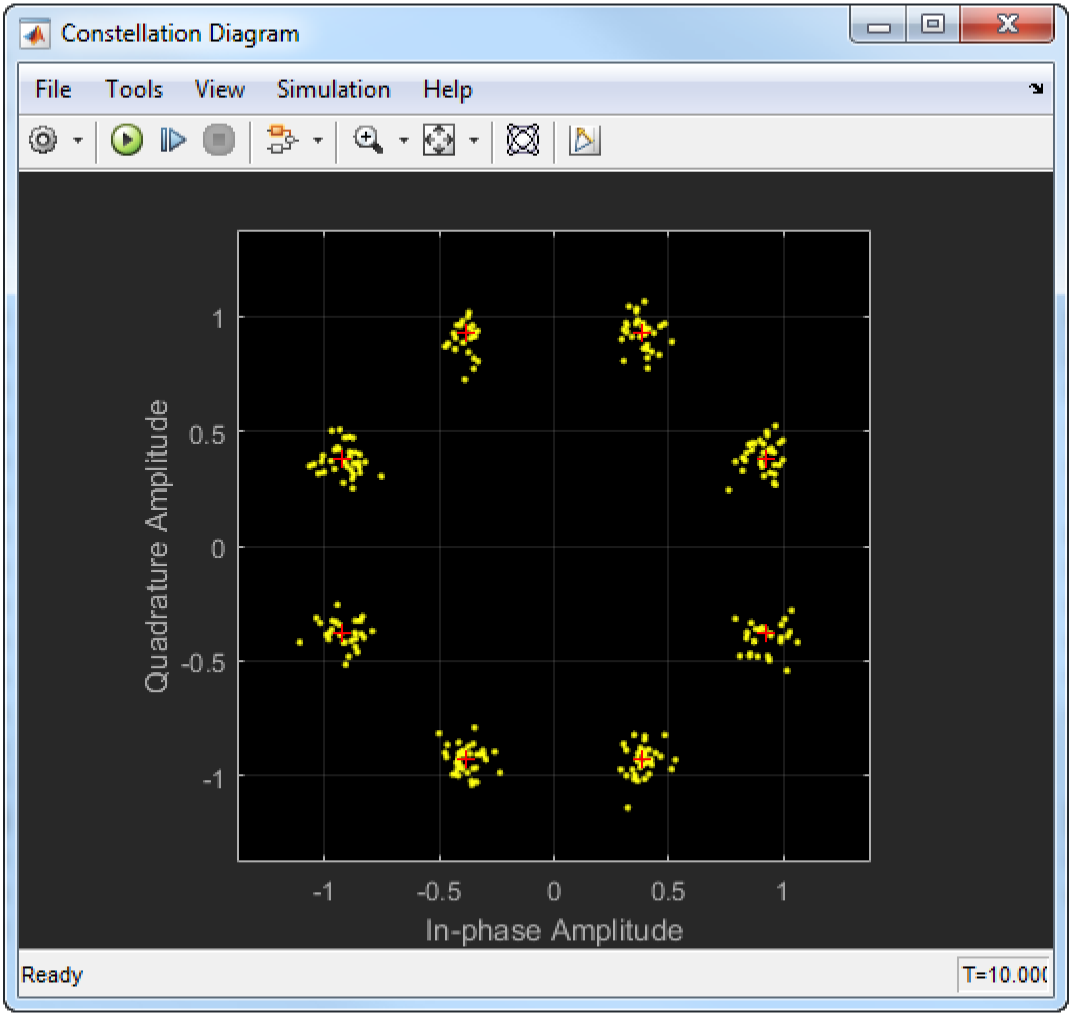
```
doc_constellation_diagram_example
```

The model includes:

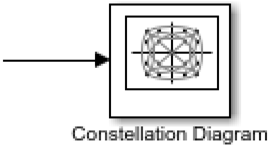
- A Random Integer Generator block
- An M-PSK Modulator Baseband block
- An AWGN Channel block
- A Constellation Diagram block

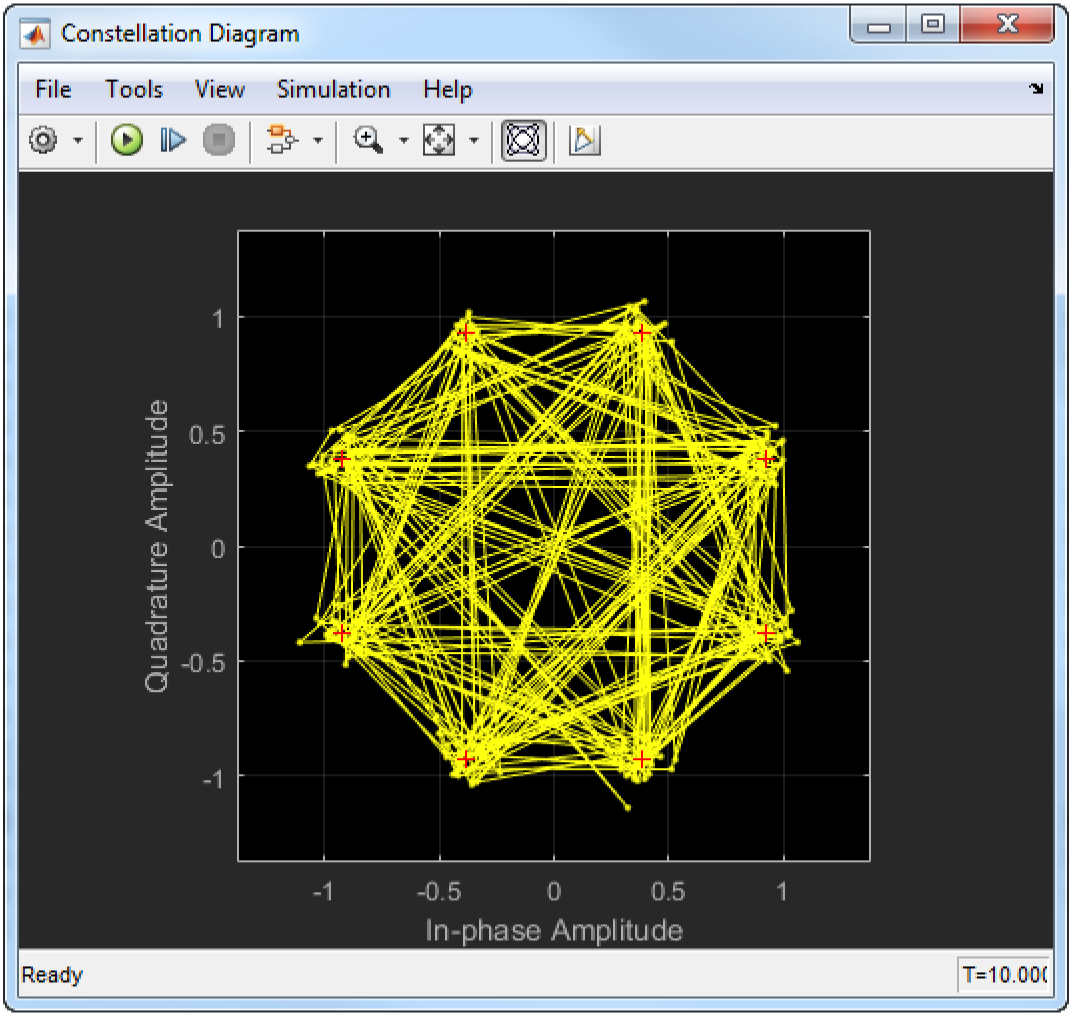

doc_constellation_diagram_example - Simulink prerelease use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10
doc_constellation_diagram_example
doc_constellation_diagram_example
Random Integer
Random Integer Generator
8-PSK
M-PSK Modulator Baseband
AWGN
AWGN Channel
Constellation Diagram
Copyright 2006-2017 The MathWorks, Inc.
Ready
100%
VariableStepDiscrete

Run the model and observe the 8-PSK constellation. The received data points are shown in yellow while the red ‘+’ symbols represent the ideal constellation locations.



Click on the **Show Signal Trajectory** button to display the signal trajectory of the modulated signal. Observe that the constellation diagram icon in the model has changed to reflect that the diagram is now displaying a trajectory and the constellation diagram now shows the signal trajectory.


Constellation Diagram


Constellation Diagram
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=10.00(

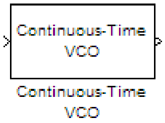
See Also

Eye Diagram | `comm.ConstellationDiagram`

Introduced in R2013b

Continuous-Time VCO

Implement voltage-controlled oscillator



Library

Components sublibrary of Synchronization

Description

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.

Parameters

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

Introduced before R2006a

Convolutional Deinterleaver

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.

Parameters

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”.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Convolutional Deinterleaver in the HDL Coder documentation.

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.

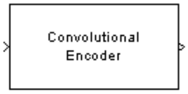
[2] Forney, G., D., Jr. "Burst-Correcting Codes for the Classic Bursty Channel." *IEEE Transactions on Communications*, vol. COM-19, October 1971. 772-781.

[3] Ramsey, J. L. "Realization of Optimum Interleavers." *IEEE Transactions on Information Theory*, IT-16 (3), May 1970. 338-345.

Introduced before R2006a

Convolutional Encoder

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” (Simulink).

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 “Trellis Description of a Convolutional Code”. 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**.

Parameters

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.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Convolutional Encoder in the HDL Coder documentation.

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.

Introduced before R2006a

Convolutional Interleaver

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.

Parameters

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”.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Convolutional Interleaver in the HDL Coder documentation.

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.

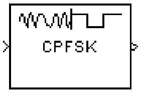
[2] Forney, G., D., Jr. "Burst-Correcting Codes for the Classic Bursty Channel." *IEEE Transactions on Communications*, vol. COM-19, October 1971. 772-781.

[3] Ramsey, J. L. "Realization of Optimum Interleavers." *IEEE Transactions on Information Theory*, IT-16 (3), May 1970. 338-345.

Introduced before R2006a

CPFSK Demodulator Baseband

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·log2(*numStates*).

For the definition of the number of states, see CPM Demodulator Baseband Help page.

Parameters

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.

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

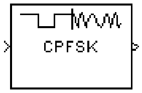
References

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

Introduced before R2006a

CPFSK Modulator Baseband

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.

Parameters

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.

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

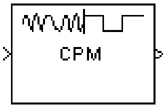
[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

Introduced before R2006a

CPM Demodulator Baseband

Demodulate CPM-modulated data

Library: Communications System Toolbox / Modulation / Digital Baseband Continuous Phase Modulation



Description

The CPM Demodulator Baseband block demodulates a signal that was modulated using continuous phase modulation (CPM).

CPM is a modulation method with memory. The block processing includes a correlator followed by a maximum-likelihood sequence detector (MLSD) that searches the paths through the state trellis for the minimum Euclidean distance path. The block uses the Viterbi algorithm to perform MLSD.

For more information about this demodulation and the filtering applied, see “CPM Demodulation” on page 2-182 and “Pulse Shape Filtering” on page 2-183.

Ports

Input

In — Input signal
scalar | column vector

Input signal, specified as a scalar or column vector. The length of the input signal must be an integer multiple of the number of samples per symbol specified in the **Samples per symbol** parameter. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 2-185.

Data Types: `double` | `single`

Output

`Out` — Output signal

scalar | column vector

Output signal, returned as a scalar or column vector. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 2-185.

Supported Data Types

- Double-precision floating point
- Boolean (when **Output type** is set to `Bit`)
- 8-, 16-, and 32-bit signed integers (when **Output type** is set to `Integer`)

Data Types: `double` | `Boolean` | `int8` | `int16` | `int32`

For more information on the processing rates, see “Single-Rate Processing” on page 2-185, and “Multirate Processing” on page 2-186.

Parameters

`M-ary number` — Modulation order

`4` (default) | positive integer

Modulation order indicating the alphabet size, specified as a positive integer that is a nonzero power of two. M must have the form 2^K for some positive integer K, where K is the number of bits per symbol.

`Output type` — Determines whether output consists of integers or groups of bits

`Integer` (default) | `Bit`

Determines whether the output consists of integers or groups of bits, specified as `Integer` or `Bit`.

`Symbol set ordering` — Bit mapping

`Binary` (default) | `Gray`

Bit mapping, specified as `Binary` or `Gray`.

This parameter determines how the block maps each integer to a group of output bits. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 2-185.

Dependencies

To enable this parameter, set **Output type** to `Bit`.

`Modulation index` — Modulation index {h_i}

`0.5` (default) | nonnegative scalar | column vector

Modulation index {h_i}, specified as a nonnegative scalar or column vector.

{h} represents a sequence of modulation indices. For more information, see “CPM Demodulation” on page 2-182.

`Frequency pulse shape` — Type of pulse shaping

`Rectangular` (default) | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`

Type of pulse shaping used to smooth the phase transitions of the modulated signal, specified as `Rectangular`, `Raised Cosine`, `Spectral Raised Cosine`, `Gaussian`, or `Tamed FM`. For more information on the filtering options, see “Pulse Shape Filtering” on page 2-183.

`Main lobe pulse duration (symbol intervals)` — Number of symbol intervals of largest lobe of the spectral raised cosine pulse

`1` (default) | positive scalar

Number of symbol intervals of the largest lobe of the spectral raised cosine pulse, specified as a positive scalar.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`Rolloff` — Rolloff factor of spectral raised cosine pulse shape

`0.2` (default) | nonnegative scalar

Rolloff factor of the spectral raised cosine pulse shape, specified as a scalar from 0 to 1.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`BT product` — Product of bandwidth and time

`0.3` (default) | nonnegative scalar

Product of bandwidth and time, specified as a nonnegative scalar. Use **BT product** to reduce the bandwidth, at the expense of increased intersymbol interference.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Gaussian`.

`Pulse length (symbol intervals)` — Frequency pulse length

`1` (default) | positive scalar

Frequency pulse shape length, specified as a positive scalar. Refer to *LT* in "Pulse Shape Filtering" on page 2-183 for more information on the frequency pulse length.

`Symbol prehistory` — Data symbols used before the start of simulation

`1` (default) | scalar | vector

Data symbols used before the start of simulation in reverse chronological order. If **Symbol prehistory** is a vector, then its length must be one less than the **Pulse length** parameter value.

`Phase offset (rad)` — Initial phase offset

`0` (default) | scalar

Initial phase offset of output in radians, specified as a scalar.

`Samples per symbol` — Symbol sampling rate

`8` (default) | positive scalar

Symbol sampling rate, specified as a positive scalar. This parameter represents the number of samples output for each integer or binary word input. For all nonbinary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see "Upsample Signals and Rate Changes".

`Rate options` — Block processing rate

`Enforce single-rate processing` (default) | `Allow multirate processing`

Block processing rate, specified as one of these options:

- `Enforce single-rate processing` — 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` — 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.

`Traceback depth` — Number of trellis branches
16 (default) | positive integer

Number of trellis branches used to construct each traceback path, specified as a positive integer. For more information, see “Traceback Depth and Output Delays” on page 2-186.

`Output data type` — Output data type
`double` (default) | `boolean` | `int8` | `int16` | `int32`

Output data type, specified as `double`, `boolean`, `int8`, `int16`, or `int32`. For more information, see **Supported Data Types** in `Out`.

Definitions

CPM Demodulation

The CPM demodulation processing 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 is rational ($h = m/p$), there are a finite number of phase states in the symbol. 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 $\{h_0, h_1, h_2, \ldots, h_{H\text{-}1}\}$.

- $h_i = m_i/p_i$ represents the modulation index in proper rational form.
- m_i represents the numerator of the modulation index.
- p_i represents the denominator of the modulation index.
- m_i and p_i are relatively prime positive numbers.
- The least common multiple (LCM) of $\{p_0, p_1, p_2, \ldots, p_{H\text{-}1}\}$ is denoted as p.
- $h_i = m'_i / p$

$\{h_i\}$ determines the number of phase states:

$$numPhaseStates = \begin{Bmatrix} p,\ for\ all\ even\ m'_i \\ 2p,\ for\ any\ odd\ m'_i \end{Bmatrix}$$

and affects the number of trellis states:

$numStates = numPhaseStates*M^{(L-1)}$

- L represents the **Pulse length**.
- M represents the **M-ary number**.

CPM Modulation

The input to the demodulator 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, $\pm(M-1)$.
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_i\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{h_0, h_1, h_2, ..., h_{H-1}\}$. When H=1, there is only one modulation index, h_0, which is denoted as h.

Pulse Shape Filtering

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 this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to these pulse shape expressions, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t)=\begin{cases}\dfrac{1}{2LT}, & 0\le t\le LT\\ 0 & \text{otherwise}\end{cases}$
`Raised Cosine`	$g(t)=\begin{cases}\dfrac{1}{2LT}\left[1-\cos\left(\dfrac{2\pi t}{LT}\right)\right], & 0\le t\le LT\\ 0 & \text{otherwise}\end{cases}$
`Spectral Raised Cosine`	$g(t)=\dfrac{1}{L_{main}T}\dfrac{\sin\left(\dfrac{2\pi t}{L_{main}T}\right)}{\dfrac{2\pi t}{L_{main}T}}\dfrac{\cos\left(\beta\dfrac{2\pi t}{L_{main}T}\right)}{1-\left(\dfrac{4\beta}{L_{main}T}t\right)^2},\quad 0\le\beta\le 1$
`Gaussian`	$g(t)=\dfrac{1}{2T}\left\{Q\left[2\pi B_b\dfrac{t-\frac{T}{2}}{\sqrt{\ln 2}}\right]-Q\left[2\pi B_b\dfrac{t+\frac{T}{2}}{\sqrt{\ln 2}}\right]\right\}$, wher $Q(t)=\int_t^\infty\dfrac{1}{\sqrt{2\pi}}e^{-\tau^2/2}d\tau$
`Tamed FM` (tamed frequency modulation)	$g(t)=\frac{1}{8}\left[g_0(t-T)+2g_0(t)+g_0(t+T)\right]$, where $\mathrm{g}_0(t)\approx\dfrac{1}{T}\left[\dfrac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}}-\dfrac{\pi^2}{24}\dfrac{2\sin\left(\frac{\pi t}{T}\right)-\frac{2\pi t}{T}\cos\left(\frac{\pi t}{T}\right)-\left(\frac{\pi t}{T}\right)^2\mathrm{si}}{\left(\frac{\pi t}{T}\right)^3}\right.$

- L_{main} is the main lobe pulse duration in symbol intervals.
- β is the rolloff factor of the spectral raised cosine.
- $B_{\text{b}}$is the product of the bandwidth and the Gaussian pulse.

- The duration of the pulse, *LT*, is the pulse length in symbol intervals. As defined by the expressions, the `Spectral Raised Cosine`, `Gaussian`, and `Tamed FM` pulse shapes have infinite length. For all practical purposes, *LT* specifies the truncated finite length.

For more information on pulse shape filtering, see [1]

Integer-Valued and Binary-Valued Output Signals

When the **Output type** parameter is set to `Integer`:

- The block produces odd integers between –(*M*–1) and *M*–1. The modulation order, *M*, is specified by the **M-ary number** parameter.
- The **Output datatype** parameter cannot be set to `boolean`.

When the **Output type** parameter is set to `Bit`:

- The block produces groupings of *K* bits. Each grouping is called a binary word.
- The **Output datatype** can only be `double` or `boolean`.
- In binary output mode, the block processing follows this procedure:

  1. Maps each input symbol to an intermediate value, as in the integer output mode.
  2. Maps the odd integer *k* to the nonnegative integer (*k*+*M*–1)/2.
  3. Maps each nonnegative integer to a binary word, using `Binary` or `Gray` mapping, as specified by the **Symbol set ordering** 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. 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*, 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 the **Rate options** parameter is set 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 vector consists of *D*+1 zero-value symbols.
- When the **Rate options** parameter is set to `Enforce single-rate processing`, the delay vector consists of *D* zero-value symbols.

The optimal **Traceback depth** parameter value depends 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 and *h*=2/3, applying this rule $(5 * \log 2(3 * 2^2) = 18)$ gives a result that is close to the optimum value of 20.

Pair Block

CPM Modulator Baseband — Modulates data using continuous phase modulation.

References

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

See Also

Blocks
CPFSK Demodulator Baseband | CPM Modulator Baseband | GMSK Demodulator Baseband | MSK Demodulator Baseband | Viterbi Decoder

Topics

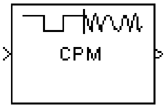
"CPM Phase Tree"

Introduced before R2006a

CPM Modulator Baseband

Modulate using continuous phase modulation

Library: Communications System Toolbox / Modulation / Digital Baseband Continuous Phase Modulation



Description

The CPM Modulator Baseband block modulates an input signal using continuous phase modulation (CPM). The output of the modulator 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:

- $\{\alpha_\mathrm{i}\}$ represents a sequence of M-ary data symbols selected from the alphabet ±1, ±3, $\pm(M–1)$.
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_\mathrm{i}\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{h_0, h_1, h_2, \ldots, h_{H\text{-}1}\}$. When H=1, there is only one modulation index, h_0, which is denoted as h.

For more information about this modulation and the filtering applied, see “CPM Modulation” on page 2-193 and “Pulse Shape Filtering” on page 2-193.

Ports

Input

In — Input signal

scalar | column vector

Input signal, specified as a scalar or column vector.

When the **Input type** parameter is set to `Integer`, the block accepts odd integers between –(*M*–1) and *M*–1. *M* represents the **M-ary number** parameter.

When the **Input type** parameter is set 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. *K* is the number of bits per symbol and *M* is the modulation order. The input vector length must be an integer multiple of *K*. The block maps each group of *K* bits onto a symbol, as specified by the **Symbol set ordering** parameter. For each group of *K* bits, the block outputs one modulated symbol, oversampled by the **Samples per symbol** parameter value.

Supported Data Types

- Double-precision floating point
- Boolean (when **Input type** is set to `Bit`)
- 8-, 16-, and 32-bit signed integers (when **Input type** is set to `Integer`)

Data Types: `double` | `Boolean` | `int8` | `int16` | `int32`

Output

Out — Output signal

scalar | column vector

Output signal, returned as a scalar or column vector.

- When the **Input type** parameter is set to `Integer`, the block outputs one modulated symbol for each input symbol.
- When the **Input type** parameter is set to `Bit`, the block outputs one modulated symbol for each group of *K* bits.

In both cases, the modulated symbols are oversampled by the **Samples per symbol** parameter value.

Data Types: `double` | `single`

For more information on the processing rates, see “Single-Rate Processing” on page 2-195, and “Multirate Processing” on page 2-195.

Parameters

`M-ary number` — Modulation order

`4` (default) | positive integer

Modulation order indicating the alphabet size, specified as a positive integer that is a nonzero power of two. M must have the form 2^K for some positive integer K, where K is the number of bits per symbol.

`Input type` — Integer or group of bits input indicator

`Integer` (default) | `Bit`

Indicates whether the input consists of integers or groups of bits, specified as `Integer` or `Bit`.

`Symbol set ordering` — Bit mapping

`Binary` (default) | `Gray`

Bit mapping, specified as `Binary` or `Gray`. For more information, see “Symbol Sets” on page 2-195.

Dependencies

To enable this parameter, set **Input type** to `Bit`.

`Modulation index` — Modulation index {h_i}

`0.5` (default) | nonnegative scalar | column vector

Modulation index {h_i}, specified as a nonnegative scalar or column vector.

{h} represents a sequence of modulation indices. For more information, see “CPM Modulation” on page 2-193.

`Frequency pulse shape` — Type of pulse shaping

`Rectangular` (default) | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`

Type of pulse shaping used to smooth the phase transitions of the modulated signal, specified as `Rectangular`, `Raised Cosine`, `Spectral Raised Cosine`, `Gaussian`, or `Tamed FM`. For more information on the filtering options, see “Pulse Shape Filtering” on page 2-193.

`Main lobe pulse duration (symbol intervals)` — Number of symbol intervals of largest lobe of spectral raised cosine pulse

`1` (default) | positive scalar

Number of symbol intervals of the largest lobe of the spectral raised cosine pulse, specified as a positive scalar.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`Rolloff` — Rolloff factor of spectral raised cosine pulse shape

`0.2` (default) | nonnegative scalar

Rolloff factor of the spectral raised cosine pulse, specified as a scalar from 0 to 1.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`BT product` — Product of bandwidth and time

`0.3` (default) | nonnegative scalar

Product of bandwidth and time, specified as a nonnegative scalar. Use **BT product** to reduce the bandwidth, at the expense of increased intersymbol interference.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Gaussian`.

`Pulse length (symbol intervals)` — Frequency pulse length

`1` (default) | positive scalar

Frequency pulse length, specified as a positive scalar. Refer to *LT* in “Pulse Shape Filtering” on page 2-193 for more information on the frequency pulse length.

`Symbol prehistory` — Data symbols used before the start of simulation

`1` (default) | scalar | vector

Data symbols used before the start of simulation, specified as a scalar or vector in reverse chronological order. If **Symbol prehistory** is a vector, then its length must be one less than the **Pulse length (symbol intervals)** parameter value.

`Phase offset (rad)` — Initial phase offset

`0` (default) | scalar

Initial phase offset of output in radians, specified as a scalar.

`Samples per symbol` — Symbol sampling rate

`8` (default) | positive scalar

Symbol sampling rate, specified as a positive scalar. This parameter represents the number of samples output for each integer or binary word input. For all nonbinary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see "Upsample Signals and Rate Changes".

`Rate options` — Block processing rate

`Enforce single-rate processing` (default) | `Allow multirate processing`

Block processing rate, specified as one of these options:

- `Enforce single-rate processing` — The input and output signals have the same 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` — The input and output signals have different sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

`Output data type` — Output data type

`double` (default) | `single`

Output data type, specified as `double` or `single`.

Definitions

CPM Modulation

The output of the modulator 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, $\pm(M-1)$.
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_i\}$ 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.

h_i specifies the modulation index. 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.

Pulse Shape Filtering

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 this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to these pulse shape expressions, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t)=\begin{cases}\dfrac{1}{2LT}, & 0\le t\le LT\\ 0 & \text{otherwise}\end{cases}$
`Raised Cosine`	$g(t)=\begin{cases}\dfrac{1}{2LT}\left[1-\cos\left(\dfrac{2\pi t}{LT}\right)\right], & 0\le t\le LT\\ 0 & \text{otherwise}\end{cases}$
`Spectral Raised Cosine`	$g(t)=\dfrac{1}{L_{main}T}\dfrac{\sin\left(\dfrac{2\pi t}{L_{main}T}\right)}{\dfrac{2\pi t}{L_{main}T}}\dfrac{\cos\left(\beta\dfrac{2\pi t}{L_{main}T}\right)}{1-\left(\dfrac{4\beta}{L_{main}T}t\right)^2},\quad 0\le\beta\le 1$
`Gaussian`	$g(t)=\dfrac{1}{2T}\left\{Q\left[2\pi B_b\dfrac{t-\frac{T}{2}}{\sqrt{\ln 2}}\right]-Q\left[2\pi B_b\dfrac{t+\frac{T}{2}}{\sqrt{\ln 2}}\right]\right\}$, wher $Q(t)=\int_t^{\infty}\dfrac{1}{\sqrt{2\pi}}e^{-\tau^2/2}d\tau$
`Tamed FM` (tamed frequency modulation)	$g(t)=\frac{1}{8}\left[g_0(t-T)+2g_0(t)+g_0(t+T)\right]$, where $\mathrm{g}_0(t)\approx\dfrac{1}{T}\left[\dfrac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}}-\dfrac{\pi^2}{24}\dfrac{2\sin\left(\frac{\pi t}{T}\right)-\frac{2\pi t}{T}\cos\left(\frac{\pi t}{T}\right)-\left(\frac{\pi t}{T}\right)^2\mathrm{si}}{\left(\frac{\pi t}{T}\right)^3}\right.$

- L_{main} is the main lobe pulse duration in symbol intervals.
- β is the rolloff factor of the spectral raised cosine.
- $B_{\text{b}}$is the product of the bandwidth and the Gaussian pulse.
- The duration of the pulse, *LT*, is the pulse length in symbol intervals. As defined by the expressions, the `Spectral Raised Cosine`, `Gaussian`, and `Tamed FM` pulse shapes have infinite length. For all practical purposes, *LT* specifies the truncated finite length.

For more information on pulse shape filtering, see [1].

Symbol Sets

In binary input mode, the block processing follows this procedure:

1. Maps each binary word to k, an integer from 0 to M–1. The binary word mapping options are `Binary` or `Gray`, as specified by the **Symbol set ordering** parameter.
2. Maps k to the intermediate value $2k–(M–1)$
3. Proceeds with block processing as in the integer input mode.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. In this mode, the input to the block can be multiple symbols. The block implicitly implements the rate change by making a size change at the output when compared to the input.

- When you set **Input type** to `Integer`, the input can be a scalar or a column vector with the length equal to the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of the number of bits per symbol.

The output width equals $N_{\mathrm{Sym}} \times N_{\mathrm{SPS}}$, where N_{Sym} is the number of symbols in the frame and N_{SPS} is the number of samples per symbol.

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 $T_{\mathrm{Sym}} / N_{\mathrm{SPS}}$, where T_{Sym} is the symbol period and N_{SPS} is the number of samples per symbol.

Pair Block

CPM Demodulator Baseband — Demodulates continuous phase modulated data.

References

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

See Also

Blocks

CPFSK Modulator Baseband | CPM Demodulator Baseband | GMSK Modulator Baseband | MSK Modulator Baseband

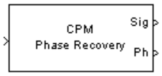
Topics

“CPM Phase Tree”
“CPM Modulation”

Introduced before R2006a

CPM Phase Recovery

Recover carrier phase using 2P-Power method



Note CPM Phase Recovery will be removed in a future release. Use the Carrier Synchronizer block instead.

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.

Parameters

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 clear the **Treat each discrete rate as a separate task** checkbox.

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.

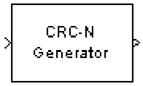
See Also

M-PSK Phase Recovery, CPM Modulator Baseband

Introduced before R2006a

CRC-N Generator

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.

Parameters

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.

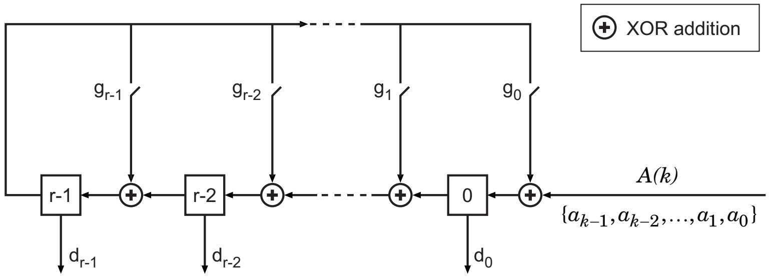
Checksums per frame

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 $a(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1x + a_0$ by $g(x) = g_{r-1}x^{r-1} + g_{r-2}x^{r-2} + \cdots + g_1x + g_0$, and returns the remainder $d(x) = d_{r-1}x^{r-1} + d_{r-2}x^{r-2} + \cdots + d_1x + d_0$.

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

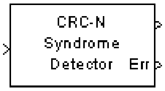
See Also

General CRC Generator, General CRC Syndrome Detector

Introduced before R2006a

CRC-N Syndrome Detector

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.

Parameters

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

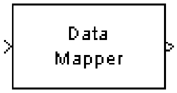
General CRC Generator, General CRC Syndrome Detector

Introduced before R2006a

Data Mapper

Map integer symbols from one coding scheme to another

Library: Utility Blocks



Description

The Data Mapper block accepts integer inputs and maps them to integer outputs. The mapping types include: binary to Gray coded, Gray coded to binary, and user defined. Additionally, a pass through option is available.

Gray coding is an ordering of binary numbers such that all adjacent numbers differ by only one bit.

Input/Output Ports

Input

`Port_1` — Input port
scalar | column vector | matrix

Input signal, specified as a scalar, vector, or matrix of integers. Elements of the input signal must be nonnegative values. The block truncates noninteger values to integer values. When the input is a matrix, the columns are treated as independent channels.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

`Port_2` — Output signal
scalar | column vector | matrix

Output signal, returned as a scalar, column vector, or matrix. The dimensions of the output signal match those of the input signal.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32

Parameters

Mapping mode — Mapping mode

Binary to Gray (default) | Gray to Binary | User Defined | Straight through

Mapping mode, specified as one of the four options. The mapping for the Binary to Gray and the Gray to Binary modes are shown in the following table when the inputs range from 0 to 7.

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. When you select the Straight Through mode, the output equals the input.

Symbol set size (M) — Symbol set size

8 (default) | positive integer

Symbol set size, specified as a positive integer. This parameter restricts the inputs and outputs to integers in the range of 0 to M-1.

Mapping vector — Maps input elements to the output elements

[0 1 3 2 7 6 4 5] (default) | vector

Mapping vector, specified as vector of nonnegative integers whose length equals . This parameter defines the relationship between the input and output integers. For example, the vector `[1 5 0 4 2 3]` defines the following mapping:

$$0 \rightarrow 1$$
$$1 \rightarrow 5$$
$$2 \rightarrow 0$$
$$3 \rightarrow 4$$
$$4 \rightarrow 2$$
$$5 \rightarrow 3$$

See Also

`bin2gray` | `gray2bin`

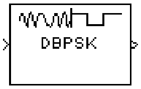
Topics

"Phase Modulation"

Introduced before R2006a

DBPSK Demodulator Baseband

Demodulate DBPSK-modulated data



Library

PM, in Digital Baseband sublibrary of Modulation

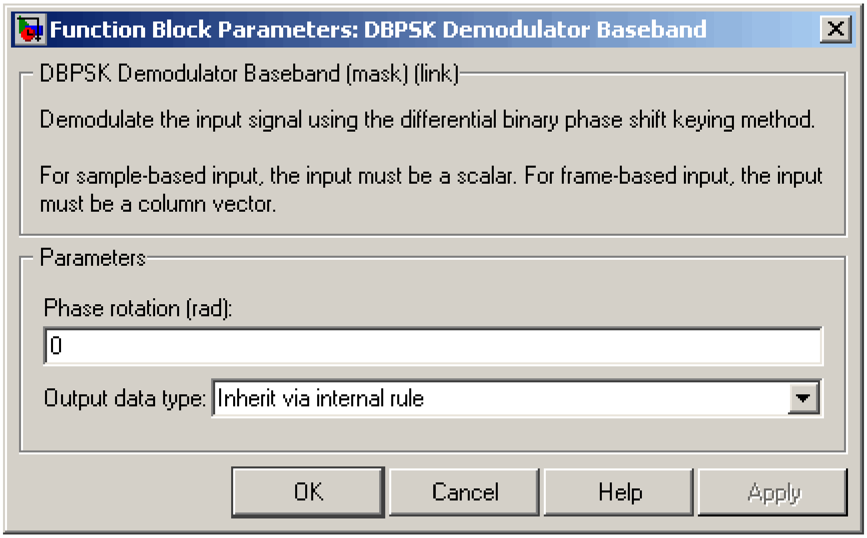
Description

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 2-210.

Dialog Box



Function Block Parameters: DBPSK Demodulator Baseband

DBPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential binary phase shift keying method.

For sample-based input, the input must be a scalar. For frame-based input, the input must be a column vector.

Parameters

Phase rotation (rad):

0

Output data type: Inherit via internal rule

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 2-210.

Supported Data Types

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

Port	Supported Data Types
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

Introduced before R2006a

DBPSK Modulator Baseband

Modulate using differential binary phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

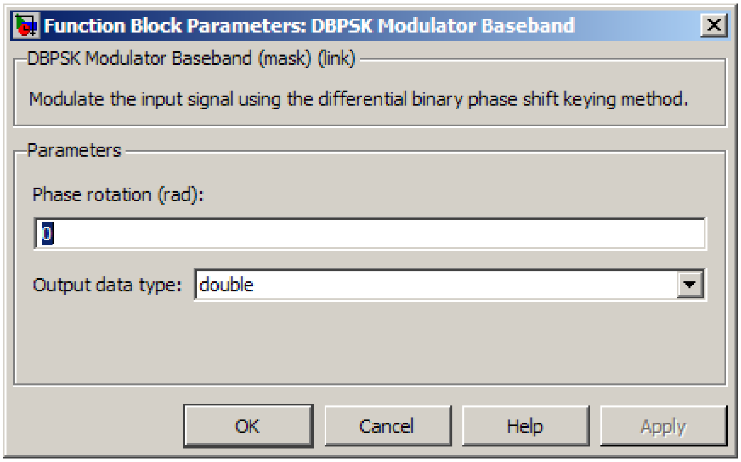
The DBPSK Modulator Baseband block modulates using the differential binary phase shift keying method. The output is a baseband representation of the modulated signal.

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 2-213.

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



Function Block Parameters: DBPSK Modulator Baseband

DBPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential binary phase shift keying method.

Parameters

Phase rotation (rad): 0

Output data type: double

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

Introduced before R2006a

Deinterlacer

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.

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)

Introduced before R2006a

Derepeat

Reduce sampling rate by averaging consecutive samples

Library: Communications System Toolbox / 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. *N* represent the `Derepeat factor, N` parameter.

Ports

Input

`In` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Output signal
scalar | vector | matrix

Output signal, returned as a scalar or column vector.

Data Types: `double`
Complex Number Support: Yes

For more information on the processing rates, see “Single-Rate Processing” on page 2-219, and “Multirate Processing” on page 2-219.

Parameters

`Derepeat factor, N` — Derepeat factor

5 (default) | integer

Derepeat factor, specified as an integer. The derepeat factor is the number of consecutive input samples to average to produce each output sample.

Data Types: `double`

`Input processing` — Input processing control

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Input processing control, specified as one of these options:

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel.

`Rate options` — Block processing rate

`Allow multirate processing` (default) | `Enforce single-rate processing`

Block processing rate, specified as one of these options:

- `Allow multirate processing` — The block downsamples the signal such that the output sample rate is `Derepeat factor, N` times slower than the input sample rate. For more information, see “Multirate Processing” on page 2-219.
- `Enforce single-rate processing` — The block maintains the input sample rate by decreasing the output frame size by a factor equal to the `Derepeat factor, N` parameter value. Also, in single-rate processing mode you can use this block in a triggered subsystem. For more information, see “Single-Rate Processing” on page 2-219

`Initial condition` — Initial condition

0 (default) | scalar | vector | matrix

Initial condition, specified as a scalar, vector, or matrix. This parameter specifies values that are output when it is 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.

Data Types: `double`

Definitions

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. *N* must be less than the frame size. *N* represents the `Derepeat factor, N` parameter.

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. To process all input values, *N* must 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, for a single-channel input with 64 elements that is derepeated by a factor of 4, the block outputs 16 elements. The input and output frame periods are equal.

Also, in single-rate processing mode you can use this block in a triggered subsystem.

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 *N* times slower than the input. *N* represents the `Derepeat factor, N` parameter.

- When you set the `Input processing` parameter to `Elements as channels (sample based)`, 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 generates 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 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, for a single-channel input with a frame period of 1 second that is derepeated by a factor of 4, the output has a frame period of 4 seconds. The input and output frame sizes are equal.

Pair Block

Repeat — This block is one possible inverse operation.

See Also

Blocks
Downsample | Repeat

Introduced before R2006a

Descrambler

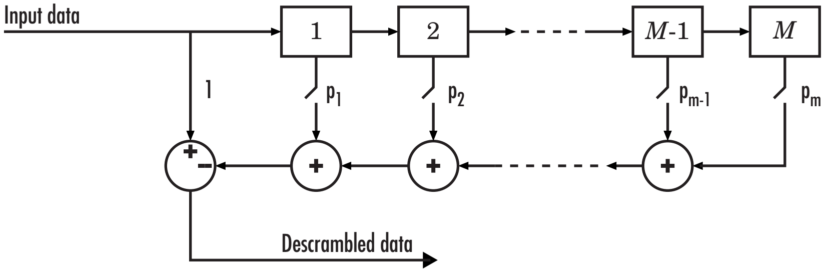
Descramble input signal

Library: Communications System Toolbox / 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.

This schematic shows the descrambler operation. The adders and subtracter operate modulo *N*, where *N* is the value specified by the Calculation base parameter.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Scramble polynomial parameter, you specify the on or off state for each switch in the descrambler. To make the Descrambler block reverse the operation of the Scrambler block, use the same parameter settings 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.

To achieve repeatable initial descrambler conditions, you can use one of these optional input ports:

- Select the Reset on nonzero input via port parameter and reset the scrambler with Rst.
- Set the Initial states source parameter to `Input port` and provide the initial states with ISt.

This block can accept input sequences that vary in length during simulation. For more information about sequences that vary in length, see “Variable-Size Signal Basics” (Simulink).

Ports

Input

`in` — Input data signal

vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal. The input values must be integers from `0` to Calculation base – `1`.

Data Types: `double`

`Rst` — Reset scrambler

scalar

Reset scrambler, specified as a scalar. The scrambler is reset if a nonzero input is applied to the port.

Dependencies

To enable this port, set Initial states source to `Dialog Parameter` and select Reset on nonzero input via port.

`ISt` — Initial states

scalar

Initial states of the descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **ISt** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from `0` to Calculation base – `1`.

Dependencies

To enable this port, set Initial states source to `Input port`.

Output

`Out1` — Output descrambled data

vector

Output descrambled data, returned as an N_S-by-1 vector. N_S equals the number of samples in the input signal.

Data Types: `double`

Parameters

`Calculation base` — Calculation base

4 (default) | nonnegative integer

Calculation base used in the descrambler for modulo operations, specified as a nonnegative integer. The input and output of this block are integers from `0` to **Calculation base** – `1`.

`Scramble polynomial` — Polynomial that defines connections in descrambler

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Polynomial that defines the connections in the descrambler, specified as a character vector, integer vector, or binary vector. The **Scramble polynomial** parameter defines if each switch in the descrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1z^{-1} + p_2z^{-2} + ...$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of *z* that appear in the polynomial that has a coefficient of 1. In this case, the order of the descramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Initial states source — Set the source for descrambler initial states
Dialog Parameter (default) | Input port

- Dialog Parameter – Specify descrambler initial states by using the Initial states parameter.
- Input port – Specify descrambler initial states by using the ISt port.

Initial states — Initial states of descrambler registers
[0 1 2 3] (default) | nonnegative integer vector

Initial states of descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **Initial states** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from 0 to Calculation base – 1.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

Reset on nonzero input via port — Reset descrambler via input port
off (default) | on

Select this parameter to reset the Descrambler block via input port Rst.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

See Also

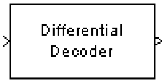
Blocks
PN Sequence Generator | Scrambler

System Objects
comm.Descrambler

Introduced before R2006a

Differential Decoder

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.

Parameters

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 • integer • fixed-point

References

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, Sixth edition, Upper Saddle River, N. J., Prentice Hall, 2001.

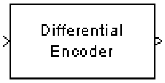
Pair Block

Differential Encoder

Introduced before R2006a

Differential Encoder

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.

Parameters

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 • 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

Introduced before R2006a

Discrete-Time VCO

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.

Parameters

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

Introduced before R2006a

DQPSK Demodulator Baseband

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 quadrature 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 2-233.

Outputs and Constellation Types

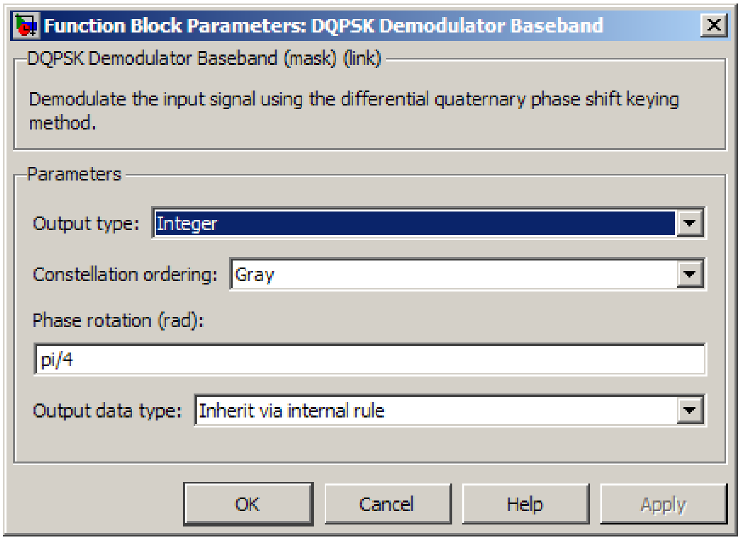
When you set **Output type** parameter to `Integer`, the block maps a phase difference of

$\theta + \pi m/2$

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



Function Block Parameters: DQPSK Demodulator Baseband

DQPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential quaternary phase shift keying method.

Parameters

Output type: Integer

Constellation ordering: Gray

Phase rotation (rad):

pi/4

Output data type: Inherit via internal rule

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

Introduced before R2006a

DQPSK Modulator Baseband

Modulate using differential quadrature phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The DQPSK Modulator Baseband block modulates using the differential quadrature 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 2-238.

Integer-Valued Signals and Binary-Valued Signals

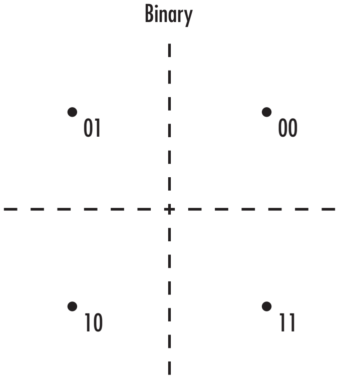
When you set the **Input type** parameter to `Integer`, the valid input values are 0, 1, 2, and 3. In this case, the block accepts a scalar or column vector input signal. If the first input is *m*, then the modulated symbol is

$\exp(j\theta + j\pi m/2)$

where θ represents the **Phase rotation** parameter. If a successive input is *m*, then the modulated symbol is the previous modulated symbol multiplied by $\exp(j\theta + j\pi m/2)$.

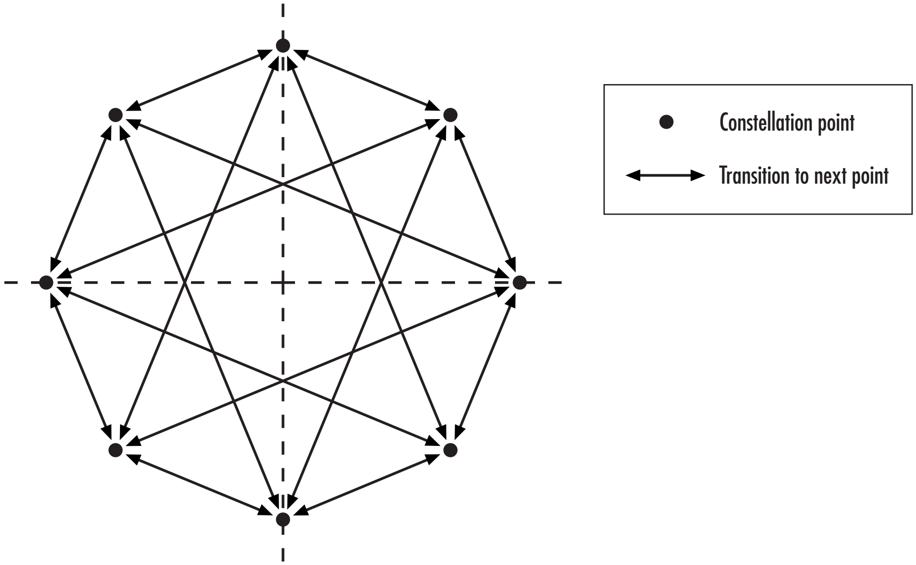
When you set the **Input type** parameter to `Bit`, the input contains pairs of binary values. In this case, the block accepts a column vector whose length is an even integer. The following figure shows the complex numbers by which the block multiples the previous symbol to compute the current symbol, 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.



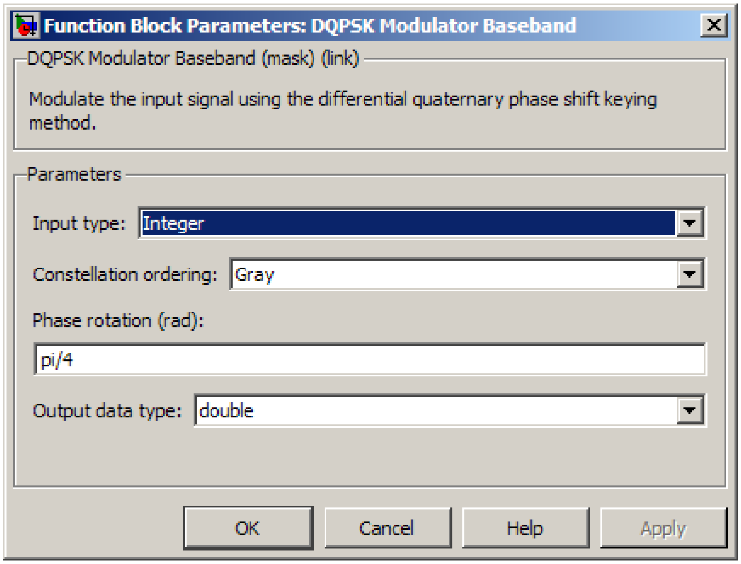
The following figure shows the signal constellation for the DQPSK modulation method

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 2*k* points.

Dialog Box



Function Block Parameters: DQPSK Modulator Baseband

DQPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential quaternary phase shift keying method.

Parameters

Input type: Integer

Constellation ordering: Gray

Phase rotation (rad): pi/4

Output data type: double

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

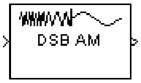
See Also

M-DPSK Modulator Baseband, DBPSK Modulator Baseband, QPSK Modulator Baseband

Introduced before R2006a

DSB AM Demodulator Passband

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.

Parameters

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.

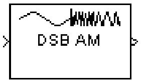
Pair Block

DSB AM Modulator Passband

Introduced before R2006a

DSB AM Modulator Passband

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.

Parameters

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

Introduced before R2006a

DSBSC AM Demodulator Passband

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.

Parameters

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

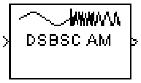
See Also

DSB AM Demodulator Passband, SSB AM Demodulator Passband

Introduced before R2006a

DSBSC AM Modulator Passband

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.

Parameters

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

Introduced before R2006a

Early-Late Gate Timing Recovery

Recover symbol timing phase using early-late gate method

Note Early-Late Gate Timing Recovery has been removed. Use the Symbol Synchronizer block instead.

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.

Parameters

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 (Simulink) 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

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.

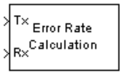
See Also

Symbol Synchronizer

Introduced before R2006a

Error Rate Calculation

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. Similarly, no data is saved to the workspace if the **Simulation mode** is set to `Accelerator` or `Rapid Accelerator`. If you need error rate information in these cases, 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 2-257 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.

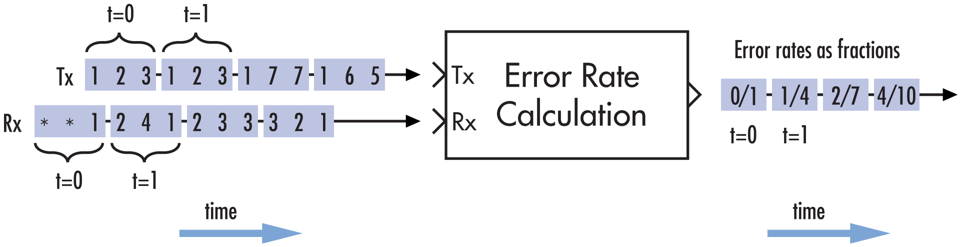
Stopping the Simulation Based on Error Statistics

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:

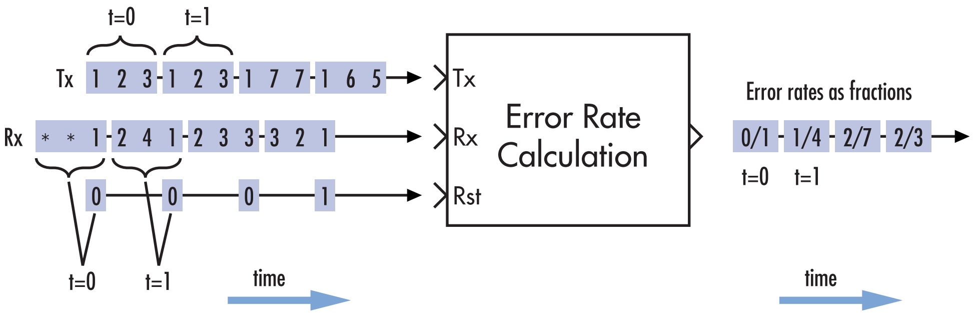
- **Receive delay** = 2
- **Computation delay** = 0
- **Computation mode** = Entire frame

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.

Parameters

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

Port	Supported Data Types
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

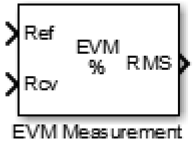
See Also

Align Signals, Find Delay

Introduced before R2006a

EVM Measurement

Measure error vector magnitude



Library

Utility Blocks

Description

The EVM Measurement block measures the error vector magnitude (EVM), which is an indication of modulator or demodulator performance.

The block has one or two input signals: a received signal and, optionally, a reference signal. You must select if the block uses a reference from an input port or from a reference constellation.

The block normalizes to the average reference signal power, average constellation power, or peak constellation power. For RMS EVM, maximum EVM, and *X*-percentile EVM, the output computations reflect the normalization method.

The default EVM output is the RMS EVM in percent, with an option of maximum EVM or *X*-percentile EVM values. The maximum EVM represents the worst-case EVM value per burst. For the *X*-percentile option, you can enable an output port that returns the number of symbols processed in the percentile computations.

The table shows the output type, the parameter that selects the output type, the computation units, and the corresponding measurement interval.

Output	Activation Parameter	Units	Measurement Interval
RMS EVM	None (output by default)	Percentage	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Maximum EVM	**Output maximum EVM**	Percentage	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Percentile EVM	**Output X-percentile EVM**	Percentage	`Entire history`
Number of symbols	**Output X-percentile EVM** and **Output the number of symbols processed**	None	`Entire history`

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always `double`.

Parameters

Normalize RMS error vector by

Selects the method by which the block normalizes measurements:

- `Average reference signal power`
- `Average constellation power`
- `Peak constellation power`

The default is `Average reference signal power`.

Average constellation power

Normalizes EVM measurement by the average constellation power. This parameter is available only when you set **Normalize RMS error vector** to `Average constellation power`.

Peak constellation power

Normalizes EVM measurement by the peak constellation power. This parameter only is available if you set **Normalize RMS error vector** to `Peak constellation power`.

Reference signal

Specifies the reference signal source as either `Input port` or `Estimated from reference constellation`.

Reference constellation

Specifies the reference constellation points as a vector. This parameter is available only when **Reference signal** is `Estimated from reference constellation`. The default is `constellation(comm.QPSKModulator)`.

Measurement interval

Specify the measurement interval as: `Input length`, `Entire history`, `Custom`, or `Custom with periodic reset`. This parameter affects the RMS and maximum EVM outputs only.

- To calculate EVM using only the current samples, set this parameter to `'Input length'`.
- To calculate EVM for all samples, set this parameter to `'Entire history'`.
- To calculate EVM over an interval you specify and to use a sliding window, set this parameter to `'Custom'`.
- To calculate EVM over an interval you specify and to reset the object each time the measurement interval is filled, set this parameter to `'Custom with periodic reset'`.

Custom measurement interval

Specify the custom measurement interval in samples as a real positive integer. This is the interval over which the EVM is calculated. This parameter is available when **Measurement interval** is `Custom` or `Custom with periodic reset`. The default is `100`.

Averaging dimensions

Specify the dimensions over which to average the EVM measurements as a scalar or row vector whose elements are positive integers. For example, to average across the rows, set this parameter to `2`. The default is `1`.

This block supports var-size inputs of the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must be constant. For example, if the input size is `[1000 3 2]` and **Averaging dimensions** is `[1 3]`, then the output size is `[1 3 1]`. The number of elements in the second dimension is fixed at 3.

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 is available only when you select **Output X-percentile EVM**. The *X*th percentile is the EVM value below which X% of all the computed EVM values lie. The parameter defaults to the 95th percentile. That is, 95% of all EVM values are below this value.

Output the number of symbols processed

Outputs the number of symbols that the block uses to compute the *X*-percentile value. This parameter is available only when you select **Output X-percentile EVM**.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

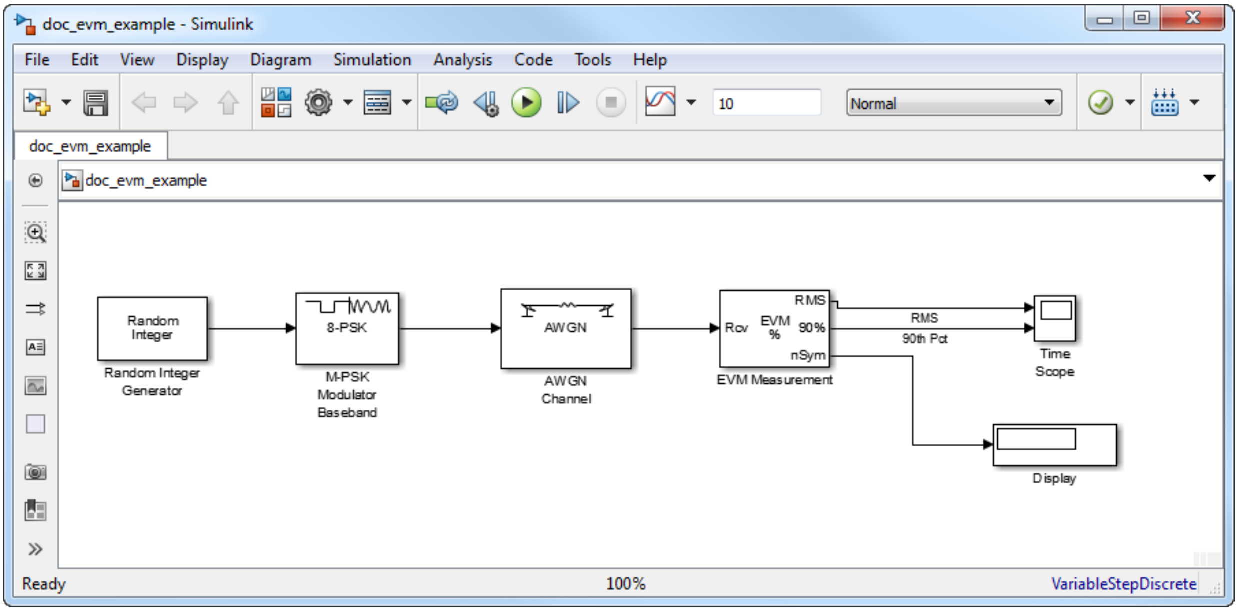
Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Examples

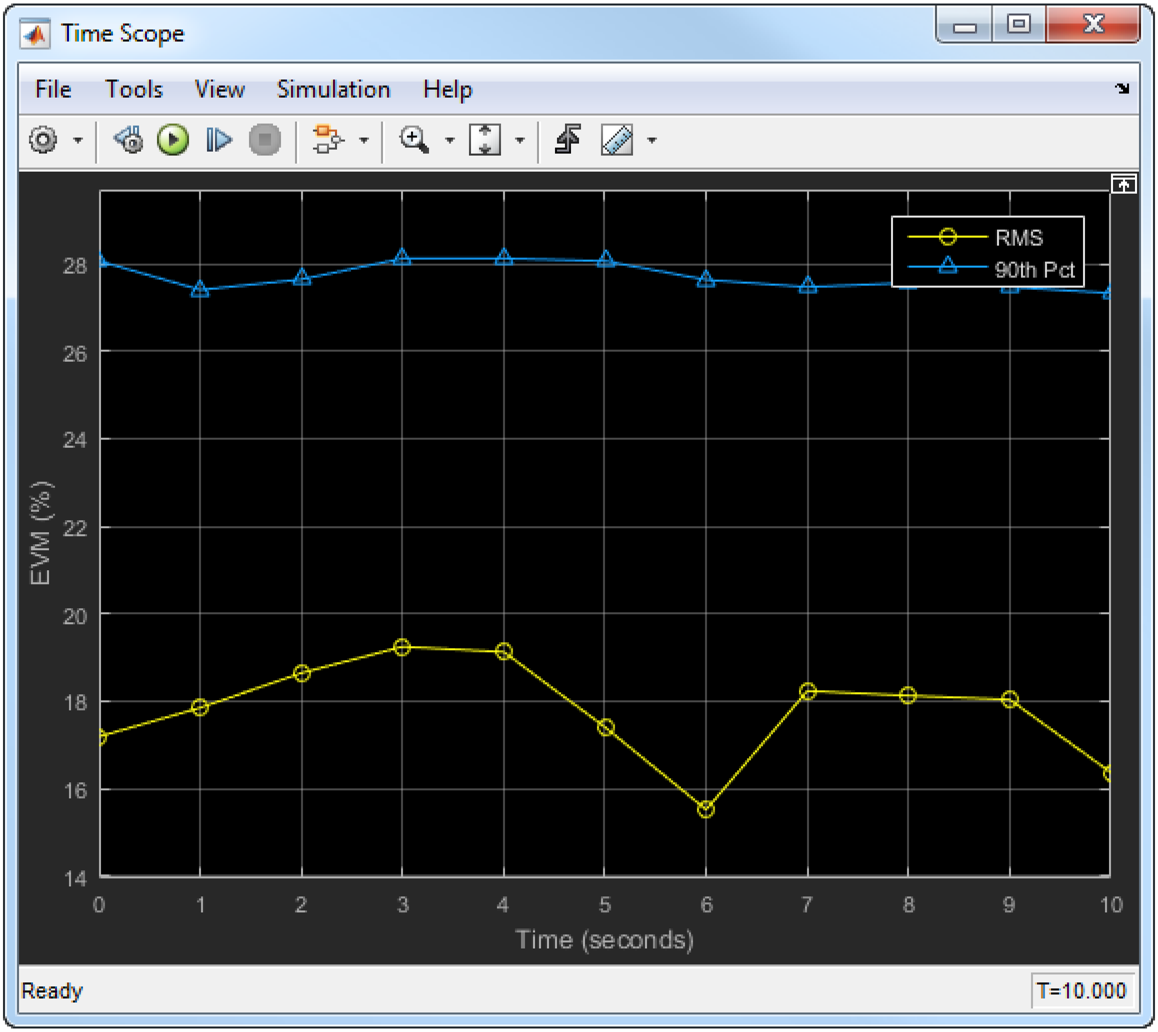
Measure RMS and 90th Percentile EVM

Measure the RMS and 90th percentile EVM for an 8-PSK signal in an AWGN channel.

Open the model by typing doc_evm_example on the command line.



Run the model. The Display block shows the number of symbols used to estimate the EVM. The Time Scope shows the RMS and 90th percentile EVM values.



Observe that 90% of the symbols had an EVM value of less than 28% and that the average EVM is approximately 17%.

Experiment with the model by changing the signal-to-noise ratio in the AWGN Channel block. Examine its effect on the EVM values.

- “EVM and MER Measurements with Simulink”

Algorithms

Both the EVM block and the EVM object provide three 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 block or object calculates the RMS EVM value differently for each normalization method.

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	$EVM_{RMS}(percent) = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{avg}}} * 100$
Peak power	$EVM_{RMS}(percent) = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\max}}} * 100$

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 $EVM_{\max} = \max_{k \in [1,...,N]} \{EVM_k\}$, where k is the kth symbol in a burst of length N.

The definition for EVM_k varies depending upon which normalization method you select for computing measurements. The block or object supports these algorithms.

EVM Normalization	Algorithm
Reference signal	$EVM_k = \sqrt{\dfrac{e_k}{\dfrac{1}{N}\sum\limits_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$EVM_k = \sqrt{\dfrac{e_k}{P_{avg}}} * 100$
Peak power	$EVM_k = \sqrt{\dfrac{e_k}{P_{\max}}} * 100$

The block or object computes the 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 fall.

References

[1] IEEE Standard 802.16-2004. “Part 16: Air interface for fixed broadband wireless access systems." October 2004. URL: 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 | `comm.EVM`

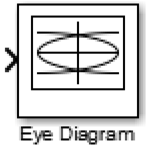
Topics

“EVM and MER Measurements with Simulink”
“Error Vector Magnitude (EVM)”

Introduced in R2009b

Eye Diagram

Display eye diagram of time-domain signal



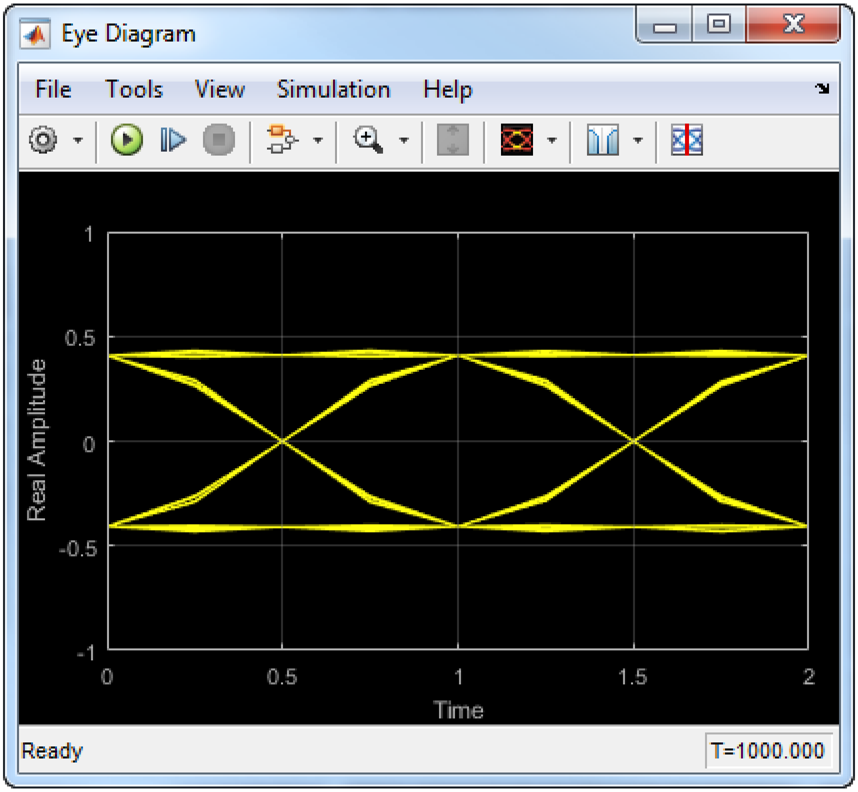
Library

Comm Sinks

Description

The Eye Diagram 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 the effects of pulse shaping or channel distortions.

The Eye Diagram block has one input port. This block accepts a column vector or scalar input signal. The block accepts a signal with the following data types: double, single, base integer, and fixed point. All data types are cast as double before the block displays results.

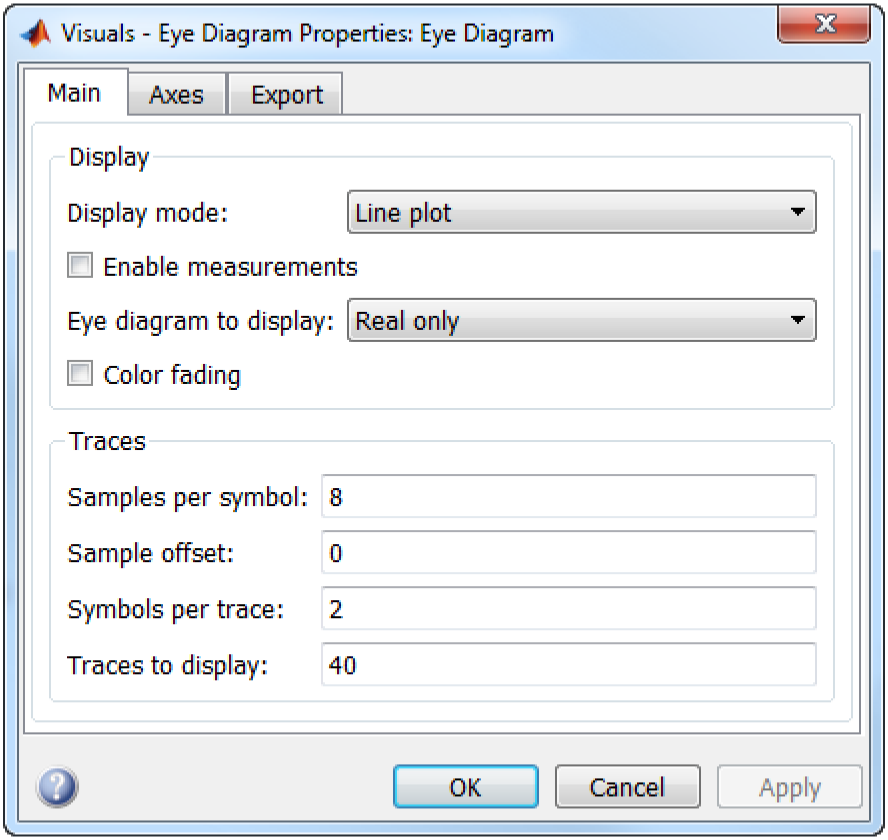


Dialog Box

To modify the eye diagram display, select **View > Configuration Properties** or click the **Configuration Properties** button (⚙). Then select the **Main**, **2D color histogram**, **Axes**, or **Export** tabs and modify the settings.

Visuals — Eye Diagram Properties

Main Tab



Display

Display mode: Line plot

□ Enable measurements

Eye diagram to display: Real only

□ Color fading

Traces

Samples per symbol: 8

Sample offset: 0

Symbols per trace: 2

Traces to display: 40

Display mode of the eye diagram, specified as `Line plot` or `2D color histogram`. Selecting `2D color histogram` makes the histogram tab available. This parameter is tunable.

Select this check box to enable eye measurements of the input signal.

Select this radio button to display the jitter histogram. This parameter is available when **Display mode** is `2D color histogram` and **Enable measurements** is selected. This can also be accessed by using the histogram button drop down on the toolbar.

Select this radio button to display the noise histogram. This parameter is available when **Display mode** is `2D color histogram` and **Enable measurements** is selected. This can also be accessed by using the histogram button drop down on the toolbar.

Select this check box to display the horizontal bathtub curve. This parameter is available when **Enable measurements** is selected. This can also be accessed by using the bathtub curve button on the toolbar.

Select this check box to display the vertical bathtub curve. This parameter is available when **Enable measurements** is selected. This can also be accessed by using the bathtub curve button on the toolbar.

Select either `Real only` or `Real and imaginary` to display one or both eye diagrams. To make eye measurements, this parameter must be `Real only`. This parameter is tunable.

Select this check box to fade the points in the display as the interval of time after they are first plotted increases. The default value is `false`. This parameter is available only when the **Display mode** is `Line plot`. This property is tunable.

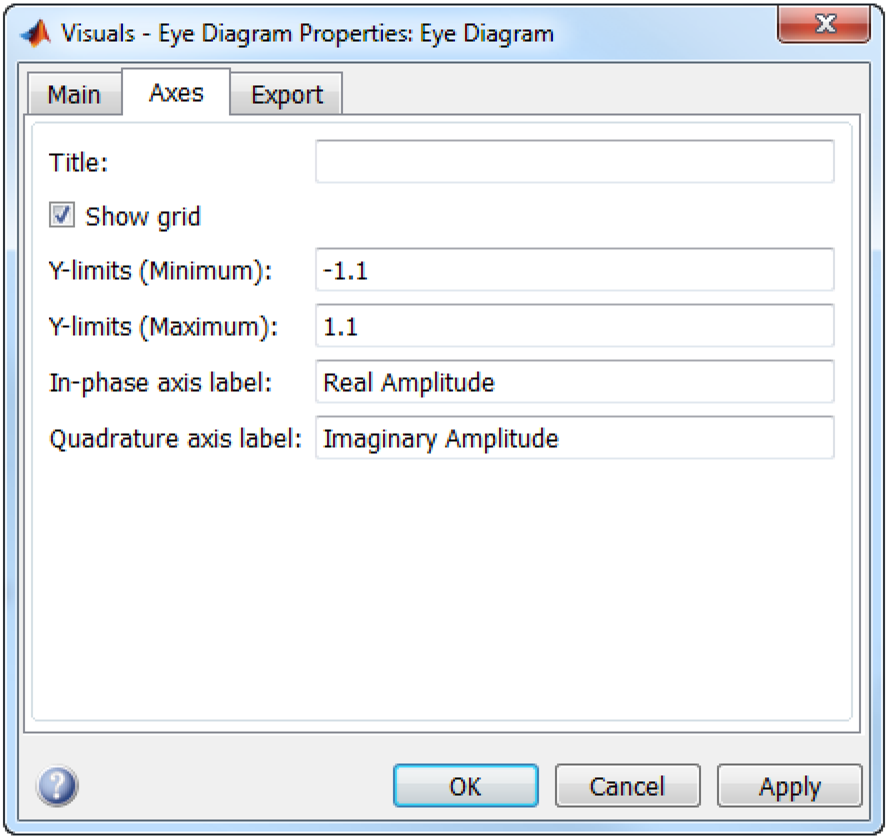
Number of samples per symbol. Use with **Symbols per trace** to determine the number of samples per trace. This parameter is tunable.

Sample offset, specified as a nonnegative integer smaller than the product of **Samples per symbol** and **Symbols per trace**. The offset provides the number of samples to omit before plotting the first point. This parameter is tunable.

Number of symbols plotted per trace, specified as a positive integer. This parameter is tunable.

Number of traces plotted. This parameter is available only when the **Display mode** is `Line plot`. This parameter is tunable.

Axes Tab



Title:

☑ Show grid

Y-limits (Minimum): -1.1

Y-limits (Maximum): 1.1

In-phase axis label: Real Amplitude

Quadrature axis label: Imaginary Amplitude

Label that appears above the eye diagram plot. By default, the plot has no title. This parameter is tunable.

Toggle this check box to turn the grid on and off. This parameter is tunable.

Minimum value of the *y*-axis. This parameter is tunable.

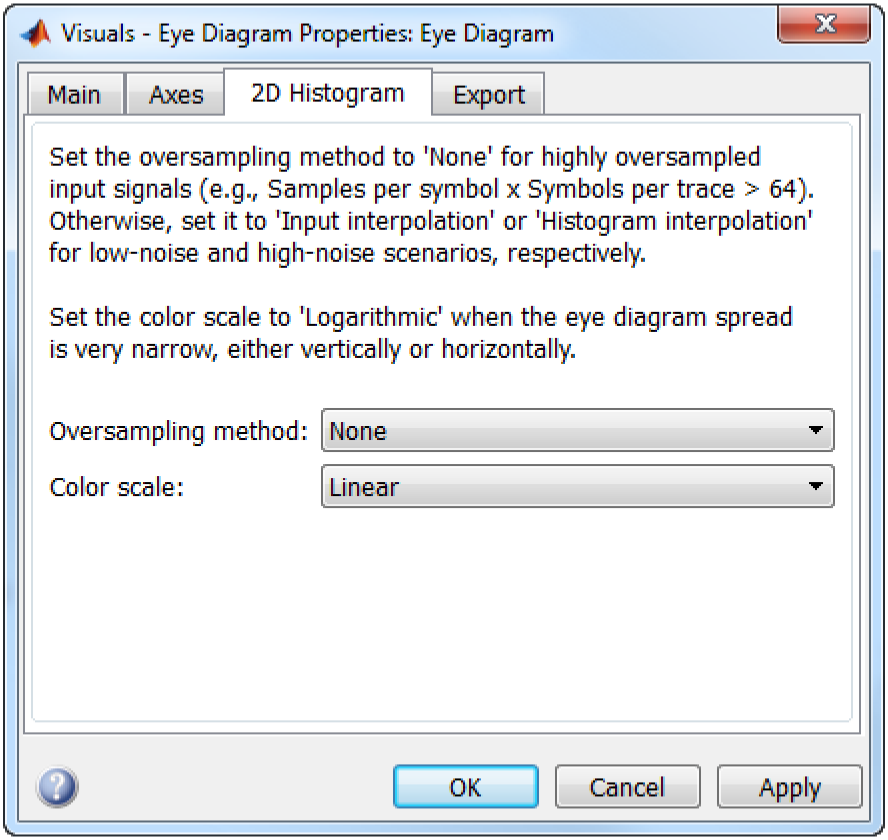
Maximum value of the *y*-axis. This parameter is tunable.

Text that the scope displays along the real axis. This parameter is tunable.

Text that the scope displays along the imaginary axis. This parameter is tunable.

2D Histogram Tab

The 2D histogram tab is available when you click the histogram button or when the display mode is set to `2D color histogram`.



Visuals - Eye Diagram Properties: Eye Diagram

Set the oversampling method to 'None' for highly oversampled input signals (e.g., Samples per symbol x Symbols per trace > 64). Otherwise, set it to 'Input interpolation' or 'Histogram interpolation' for low-noise and high-noise scenarios, respectively.

Set the color scale to 'Logarithmic' when the eye diagram spread is very narrow, either vertically or horizontally.

Oversampling method: None

Color scale: Linear

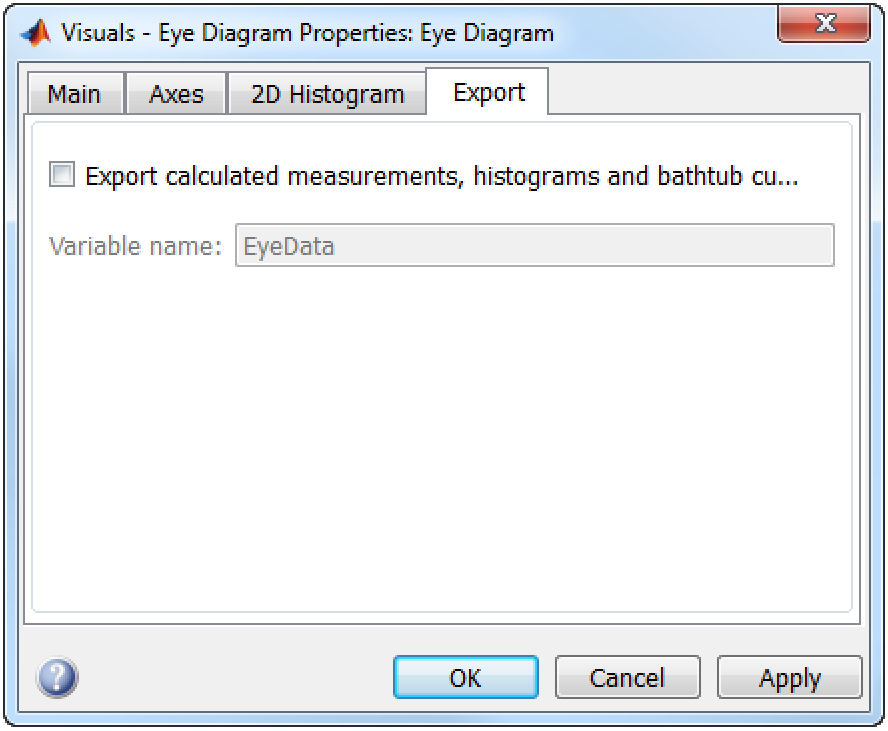
Oversampling method, specified as `None`, `Input interpolation`, or `Histogram interpolation`. This parameter is tunable.

To plot eye diagrams as quickly as possible, set the **Oversampling method** to `None`. The drawback to not oversampling is that the plots look pixelated when the number of samples per trace is small. To create smoother, less-pixelated plots using a small number of samples per trace, set the **Oversampling method** to `Input interpolation` or `Histogram interpolation`. `Input interpolation` is the faster of the two interpolation methods and produces good results when the signal-to-noise ratio (SNR) is high. With a lower SNR, this oversampling method is not recommended because it introduces a bias to the centers of the histogram ranges. `Histogram interpolation` is not as fast as the other techniques, but it provides good results even when the SNR is low.

Color scale of the histogram plot, specified as either `Linear` or `Logarithmic`. Set **Color scale** to `Logarithmic` if certain areas of the eye diagram include a disproportionate number of points. This parameter is tunable.

The toolbar contains a histogram reset button , which resets the internal histogram buffers and clears the display. This button is not available when the display mode is set to `Line plot`.

Export Tab



□ Export calculated measurements, histograms and bathtub cu...

Variable name: EyeData

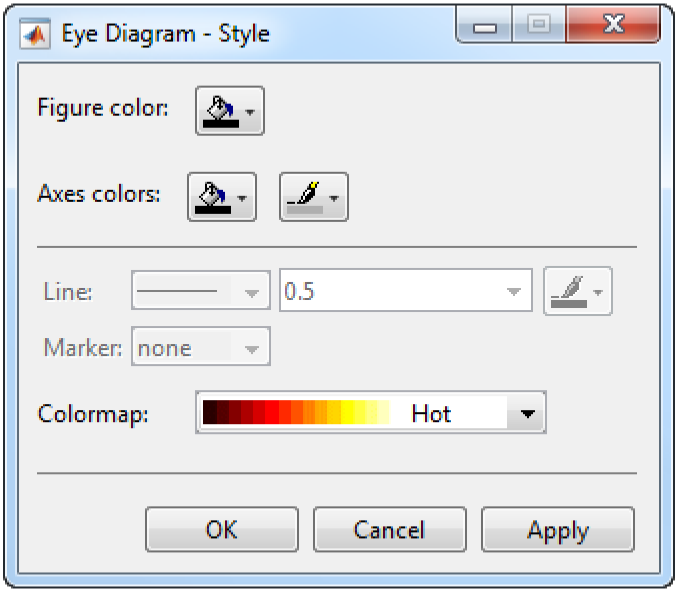
Select this check box export the eye diagram measurements to the MATLAB workspace. This parameter is tunable.

Specify the name of the variable to which the eye diagram measurements are saved. The default is `EyeData`. This parameter is tunable. The data is saved as a structure having these fields:

- `MeasurementSettings`
- `Measurements`
- `JitterHistogram`
- `NoiseHistogram`
- `HorizontalBathtub`
- `VerticalBathtub`
- `BlockName`

Style Dialog Box

In the **Style** dialog box, you can customize the style of the active display. You can change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. To open this dialog box, select **View > Style**.



Line: 0.5

Marker: none

Colormap: Hot

Properties

Specify the background color of the scope figure. By default, the figure color is black.

Specify the fill and line colors for the axes.

Specify the line style, line width, and line color for the displayed signal.

Specify data point markers for the selected signal. This parameter is similar to the `Marker` property for MATLAB Handle Graphics plot objects.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

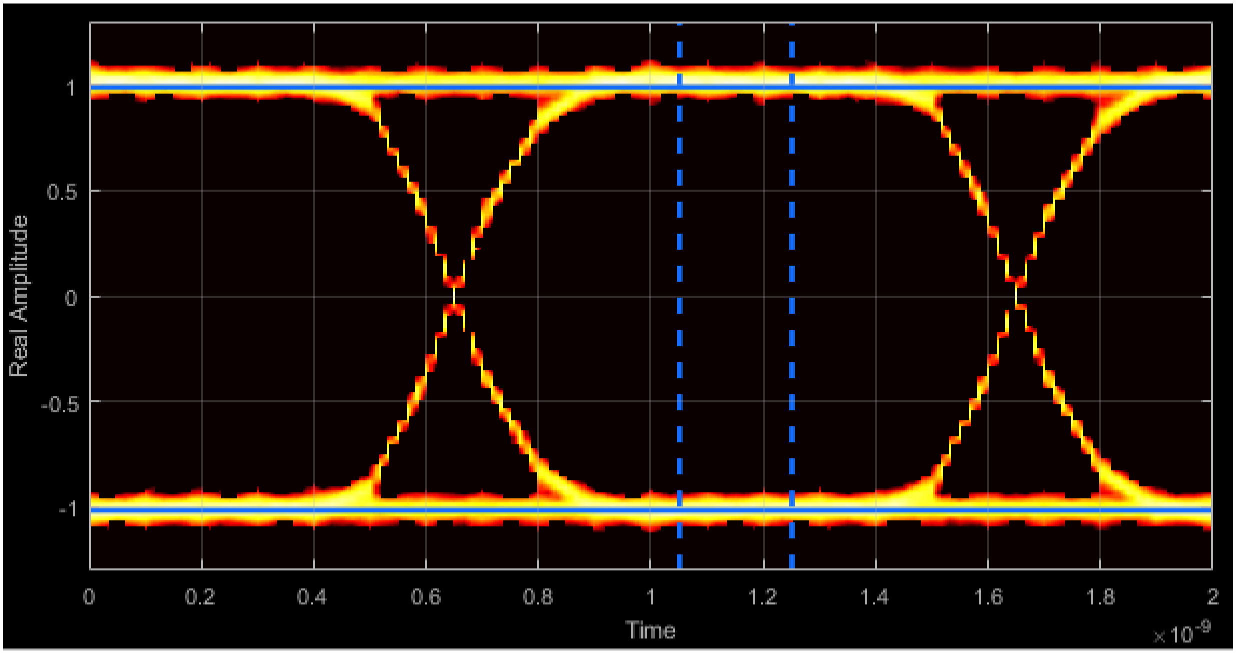
Specify the colormap of the histogram plots as one of these schemes: `Parula`, `Jet`, `HSV`, `Hot`, `Cool`, `Spring`, `Summer`, `Autumn`, `Winter`, `Gray`, `Bone`, `Copper`, `Pink`, `Lines`, or `Custom`. This parameter is active when the Eye Diagram is in `Histogram` mode. The default is Hot. If you select Custom, a dialog box pops up from which you can enter code to specify your own colormap.

Measurements

To open the measurements panel, click on the **Eye Measurements** button or select Tools > Measurements > Eye Measurements from the toolbar menu.

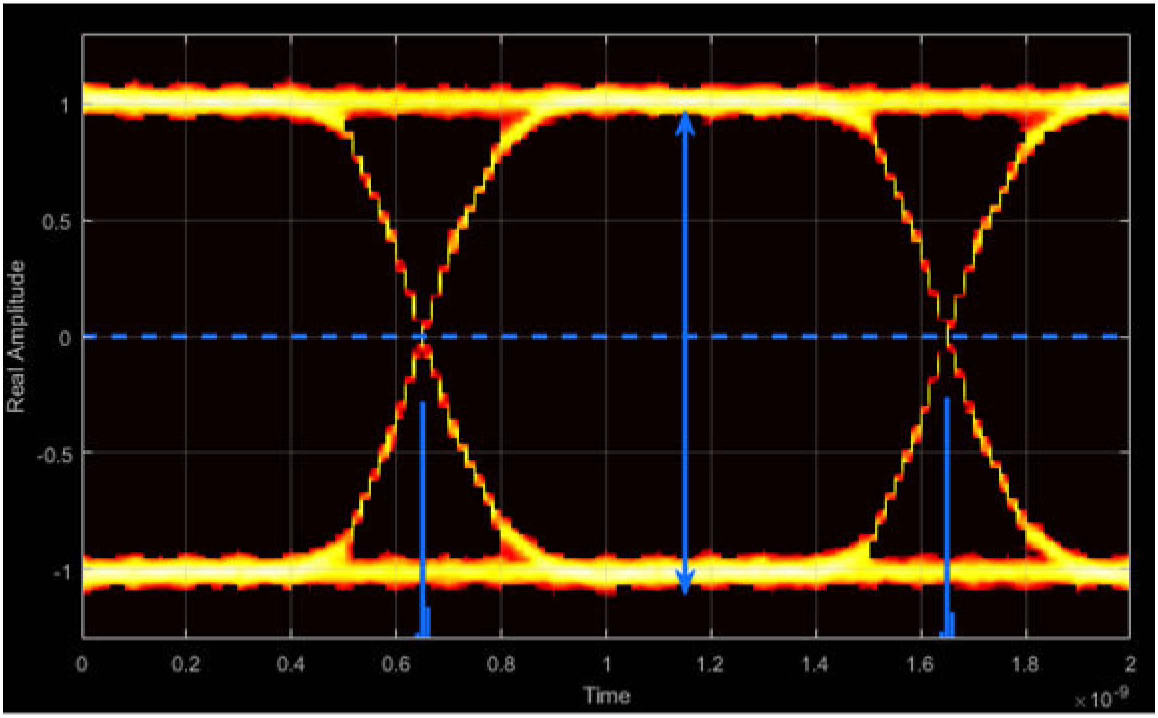
Eye Levels — Amplitude level used to represent data bits

Eye level is the amplitude level used to represent data bits. For the displayed NRZ signal, the levels are –1 V and +1 V. The eye levels are calculated by averaging the 2-D histogram within the eye level boundaries.



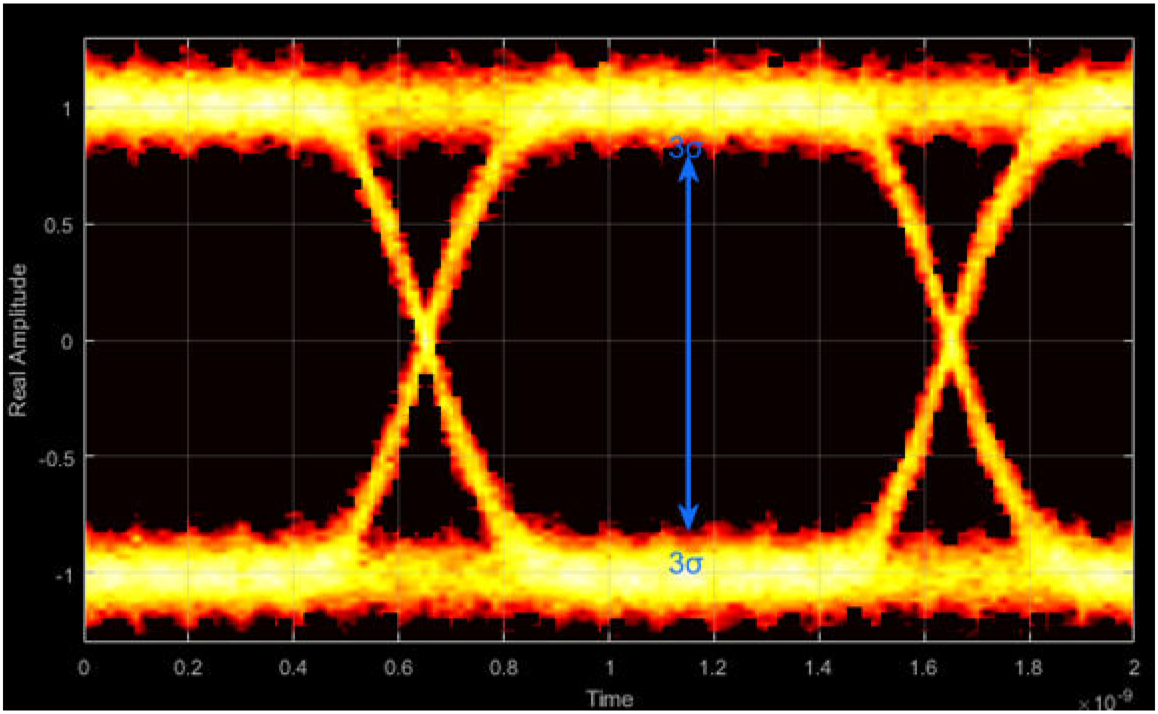
Eye Amplitude — Distance between eye levels

Eye amplitude is the distance in V between the mean value of two eye levels.



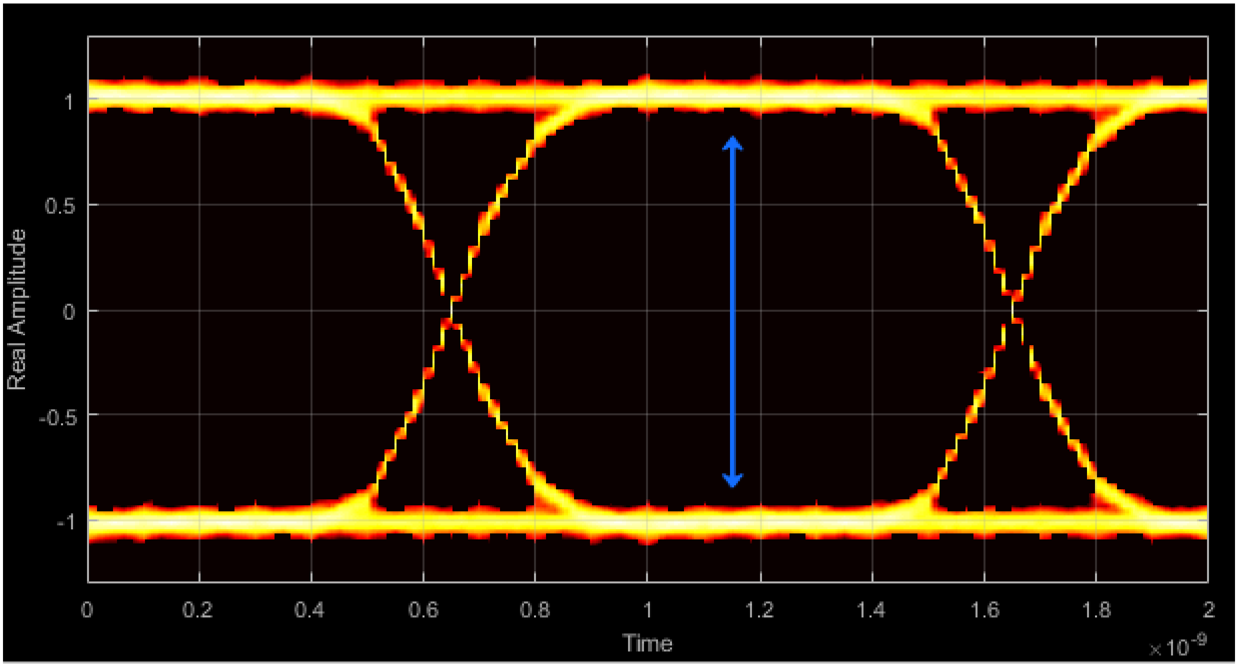
Eye Height — Statistical minimum distance between eye levels

Eye height is the distance between μ – 3σ of the upper eye level and μ + 3σ of the lower eye level. μ is the mean of the eye level and σ is the standard deviation.



`Vertical Opening` — Distance between BER threshold points

The vertical opening is the distance between the two points that correspond to the BER threshold. For example, for a BER threshold of 10^{-12}, these points correspond to the 7σ distance from each eye level.



Eye SNR — Signal-to-noise ratio

The eye SNR is the ratio of the eye level difference to the difference of the vertical standard deviations corresponding to each eye level:

$$\mathrm{SNR} = \frac{L_1 - L_0}{\sigma_1 - \sigma_0},$$

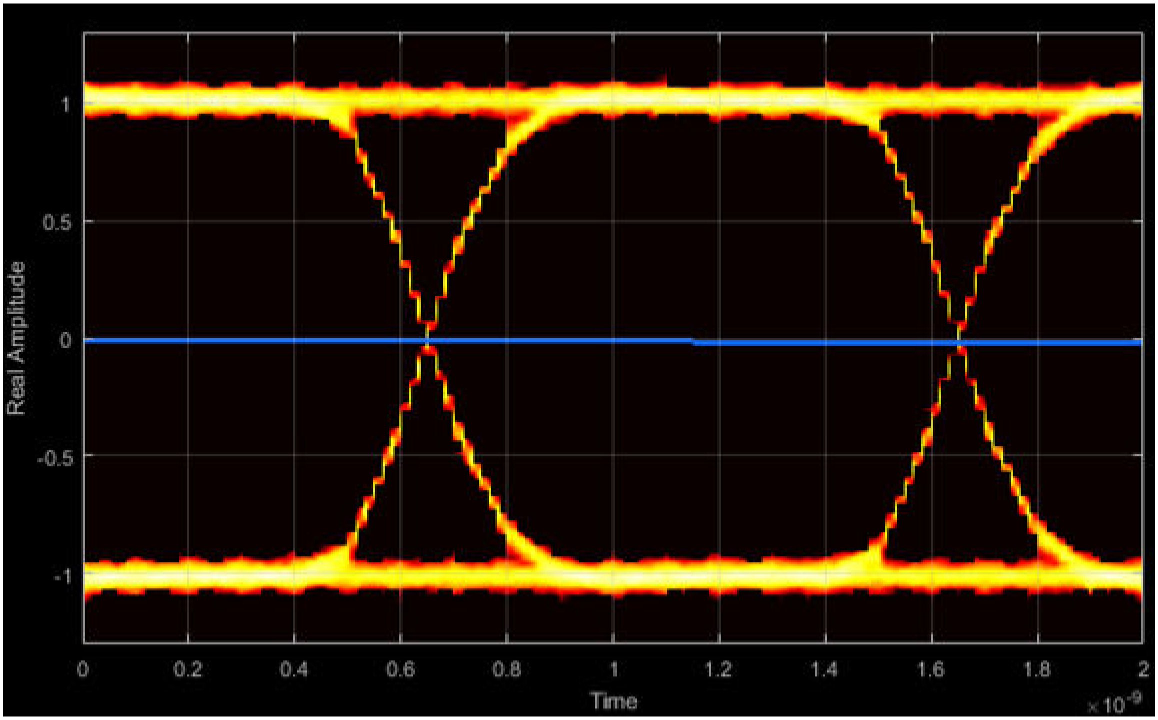
where L_1 and L_0 represent the means of the upper and lower eye levels and σ_1 and σ_0 represent their standard deviations.

Q Factor — Quality factor

The Q factor is calculated using the same formula as the Eye SNR. However, the standard deviations of the vertical histograms are replaced with those computed with the dual-Dirac analysis.

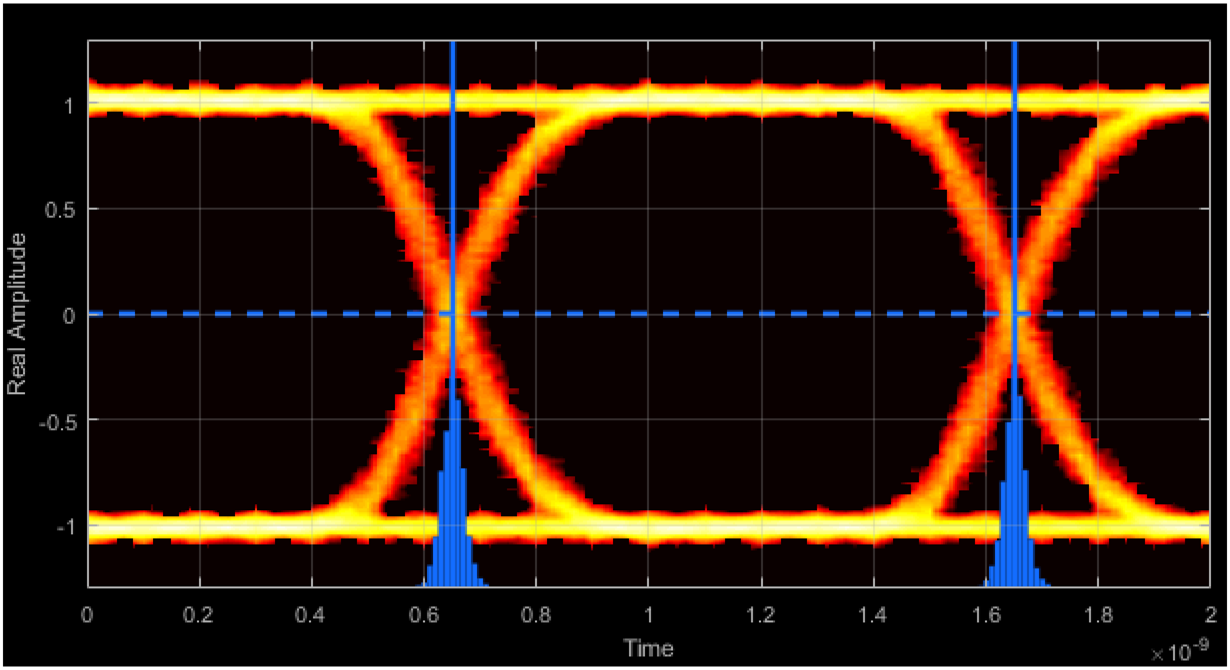
`Crossing Levels` — Amplitude levels for eye crossings

The crossing levels are the amplitude levels at which the eye crossings occur.



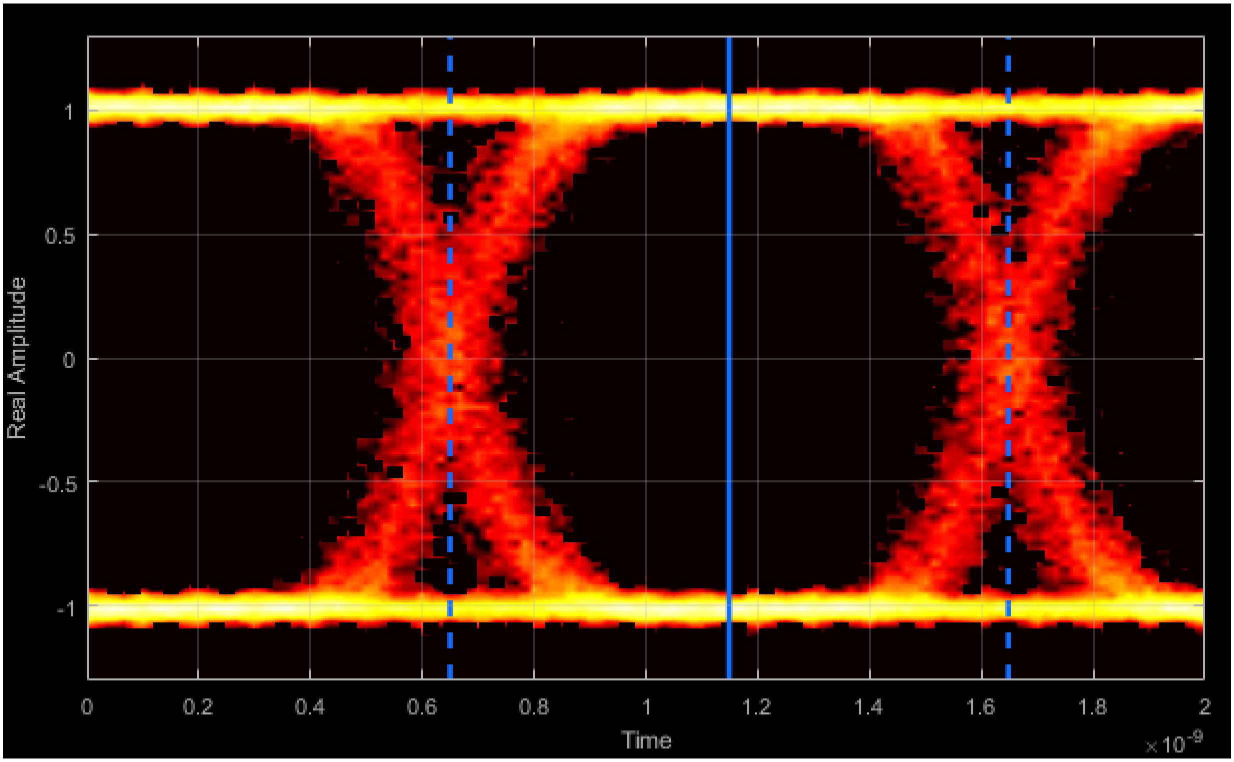
`Crossing Times` — Times for which crossings occur

The crossing times are the times at which the crossings occur. The times are computed as the mean values of the horizontal (jitter) histograms.



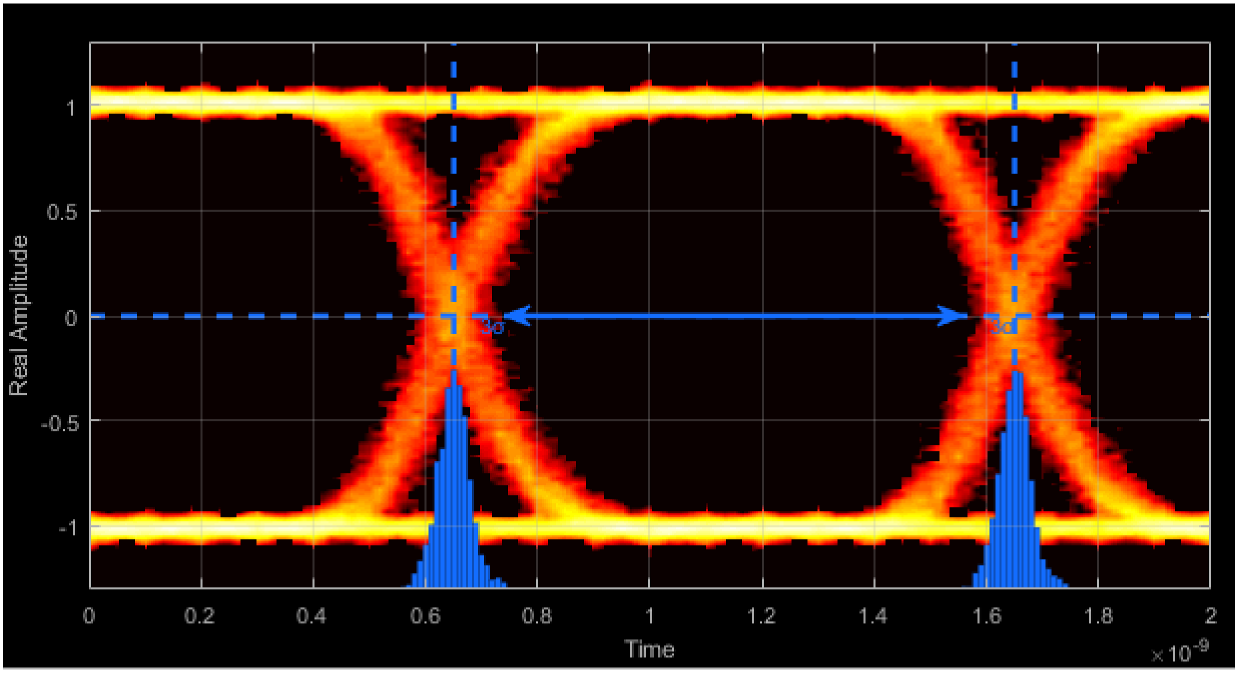
Eye Delay — Mean time between eye crossings

Eye delay is the midpoint between the two crossing times.



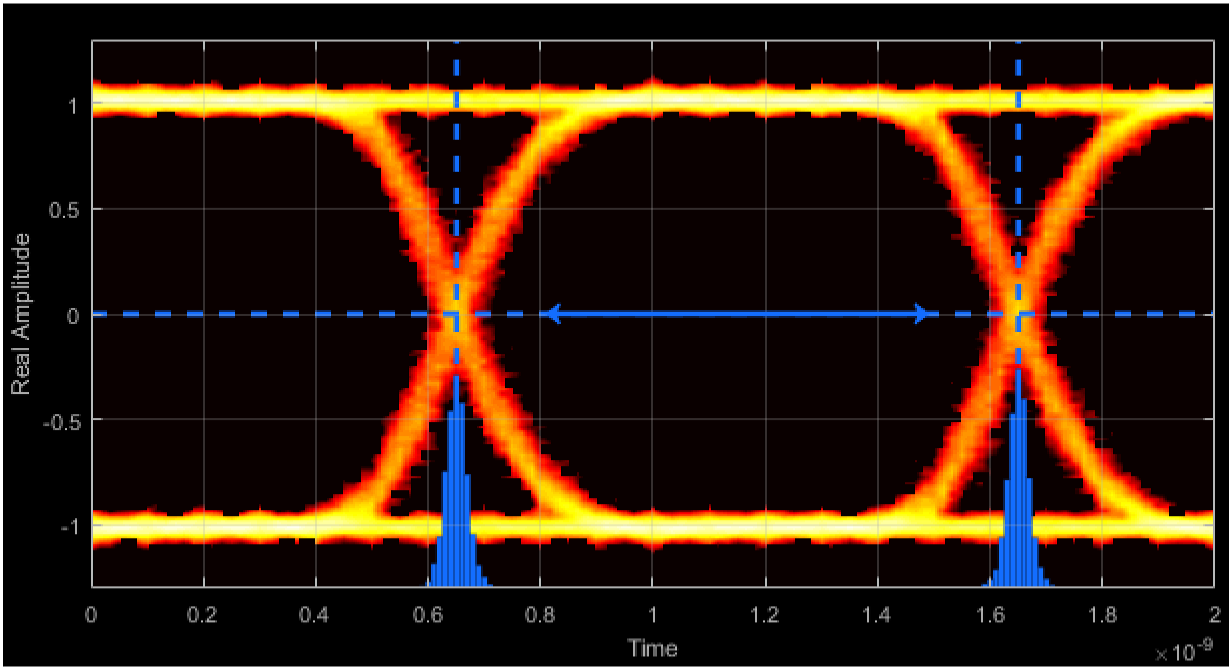
Eye Width — Statistical minimum time between eye crossings

Eye width is the horizontal distance between μ + 3σ of the left crossing time and μ – 3σ of the right crossing time. μ is the mean of the jitter histogram and σ is the standard deviation.



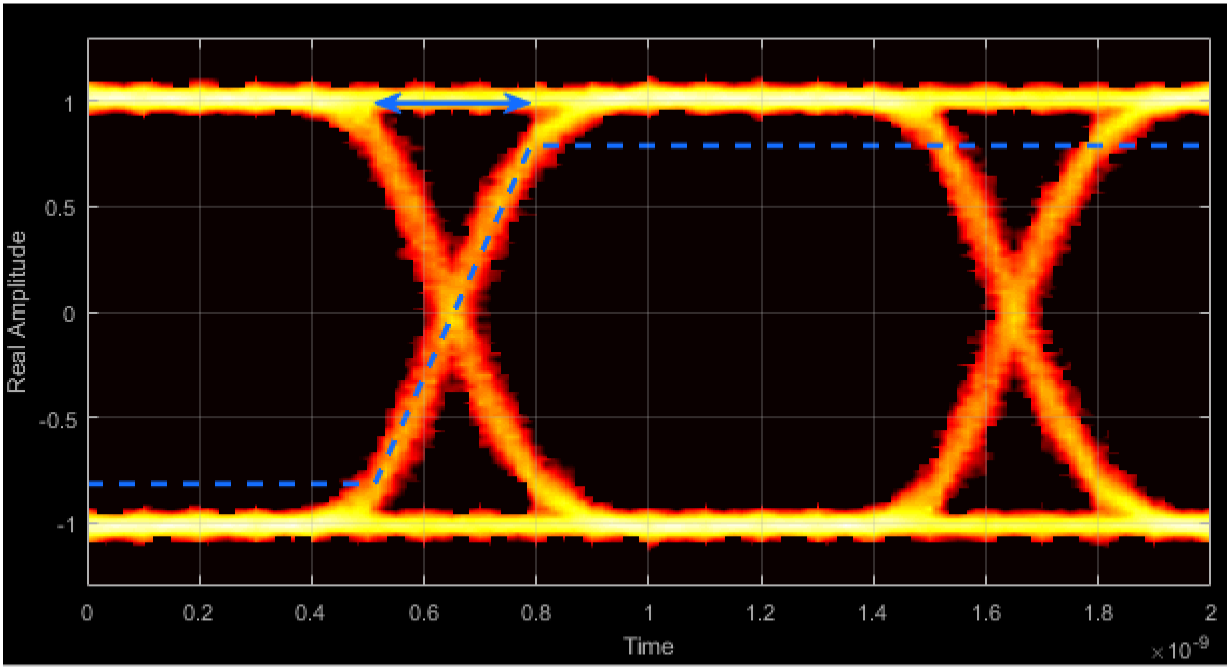
`Horizontal Opening` — Time between BER threshold points

The horizontal opening is the distance between the two points that correspond to the BER threshold. For example, for a 10^{-12} BER, these two points correspond to the 7σ distance from each crossing time.



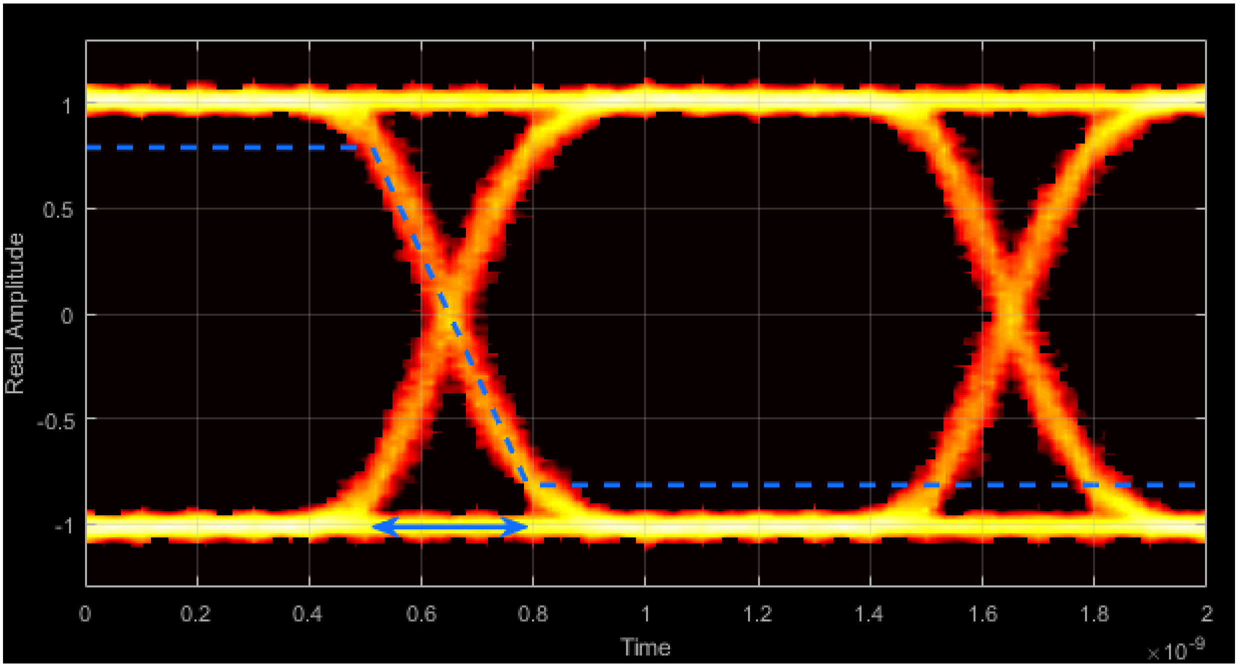
Rise Time — Time to transition from low to high

Rise time is the mean time between the low and high thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



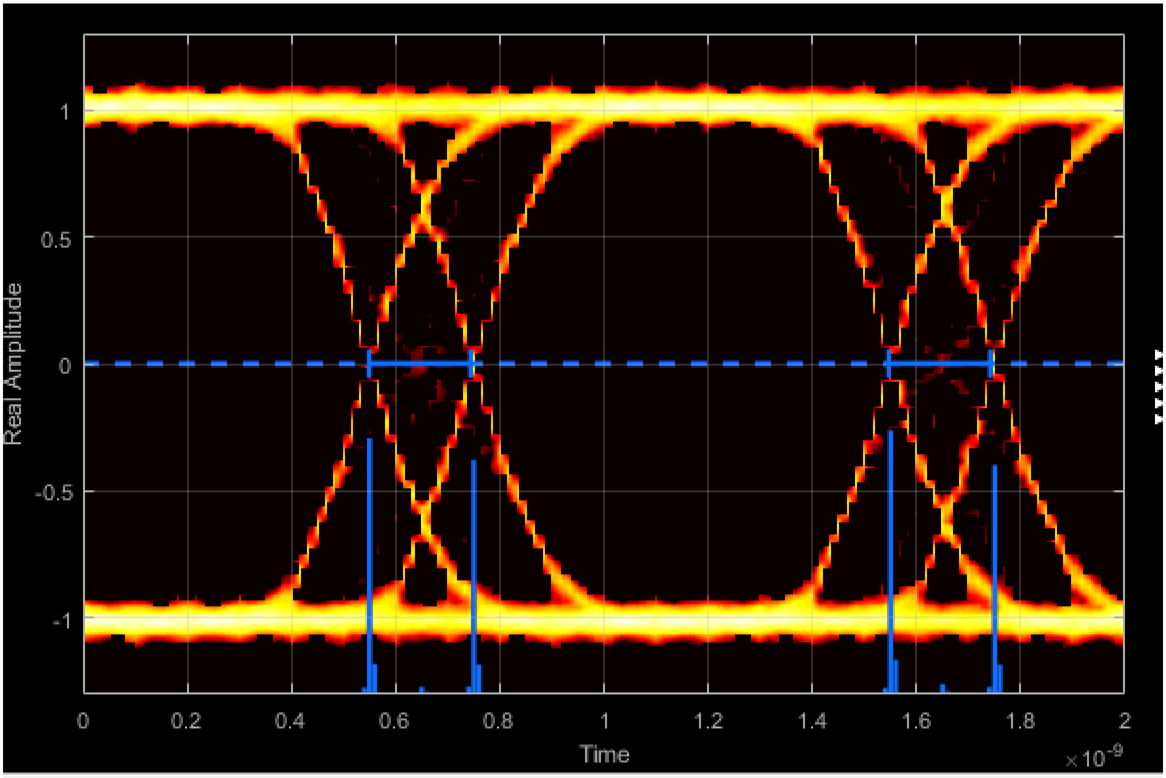
`Fall Time` — Time to transition from high to low

Fall time is the mean time between the high and low thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



Deterministic Jitter — Deterministic deviation from ideal signal timing

The deterministic jitter (DJ) is the distance between the two peaks of the dual-Dirac histograms. The probability density function (PDF) of DJ is composed of two delta functions.



`Random Jitter` — Random deviation from ideal signal timing

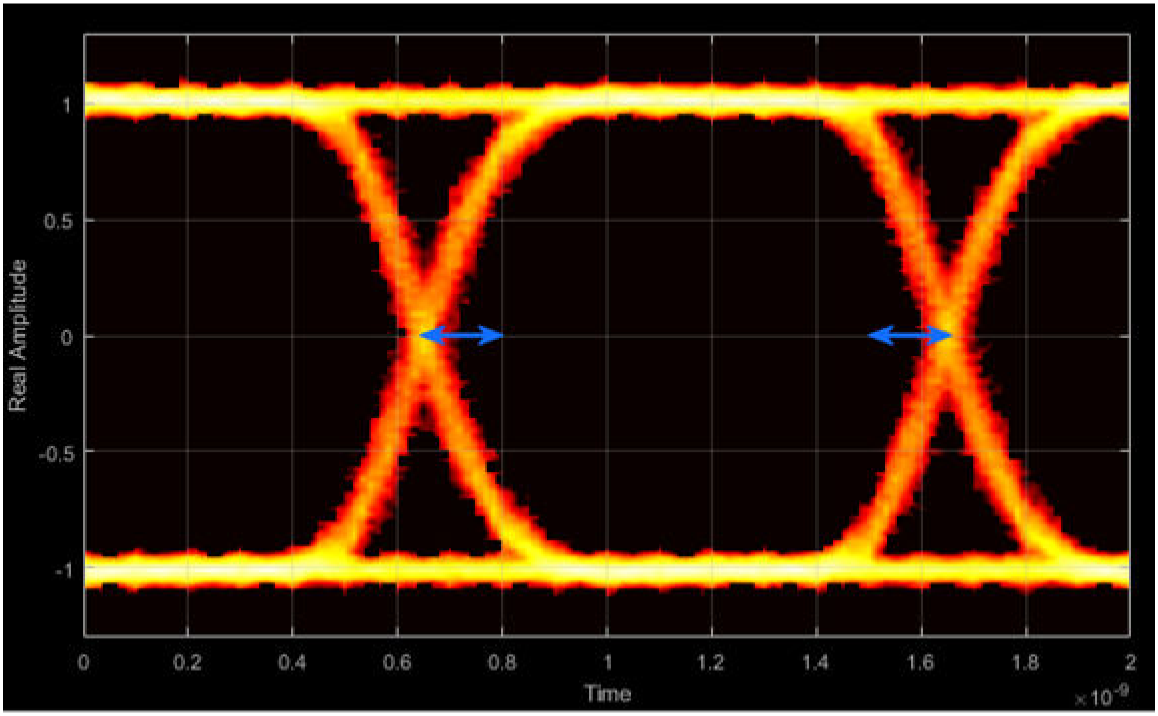
The random jitter (RJ) is the Gaussian unbounded jitter component. The random component of jitter is modeled as a zero-mean Gaussian random variable with a specified standard-deviation, σ. The random jitter is computed as:

$$\mathrm{RJ} = (Q_L + Q_R)\sigma\ ,$$

where

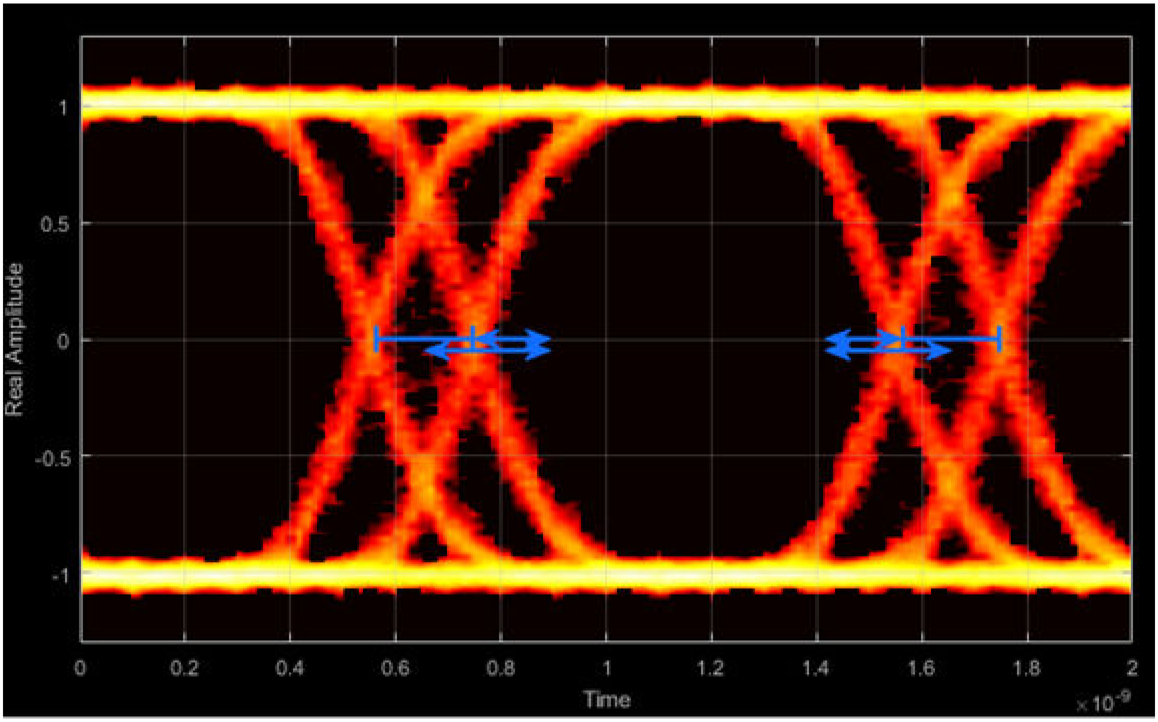
$$Q = \sqrt{2}\,\mathrm{erfc}^{-1}\left(2\frac{BER}{\rho}\right).$$

BER is the specified BER threshold. ρ is the amplitude of the left and right Dirac function, which is determined from the bin counts of the jitter histograms.

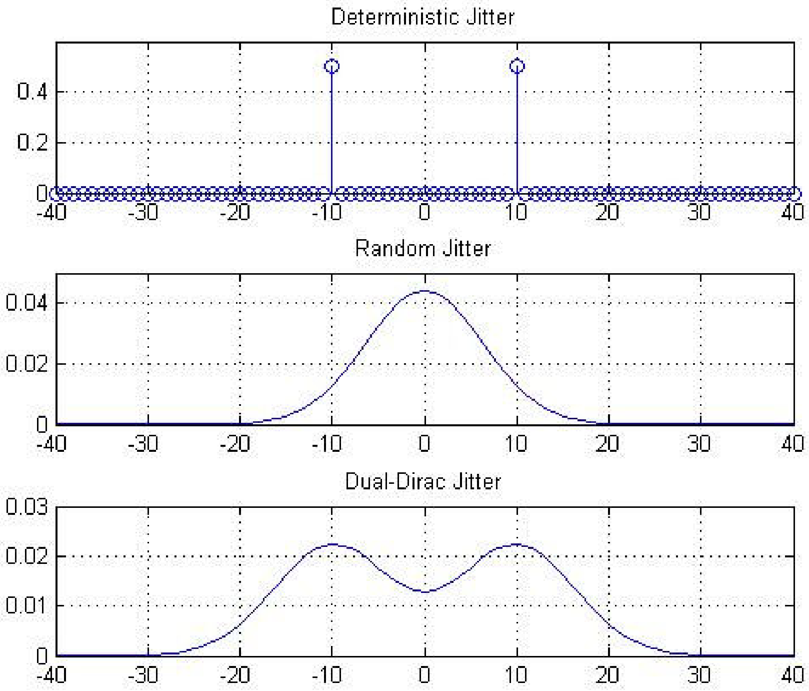


`Total Jitter` — Deviation from ideal signal timing

Total jitter (TJ) is the sum of the deterministic and random jitter, such that TJ = DJ + RJ.

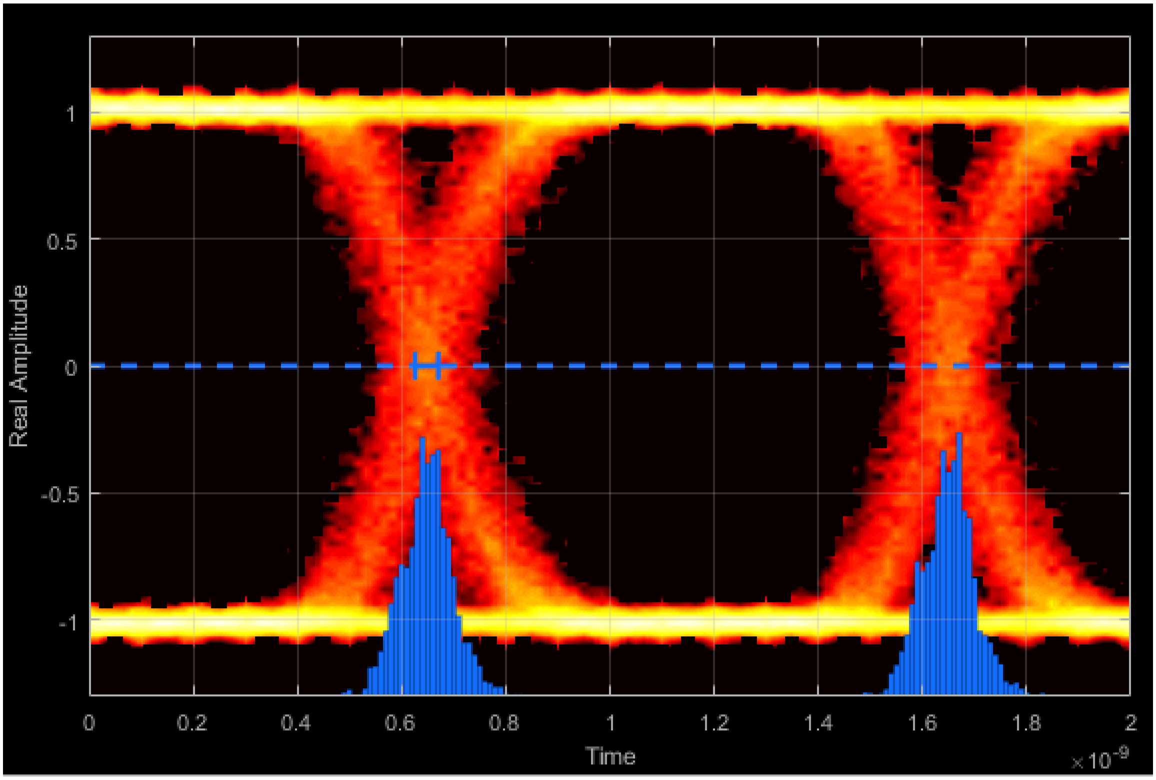


The total jitter PDF is the convolution of the DJ PDF and the RJ PDF.



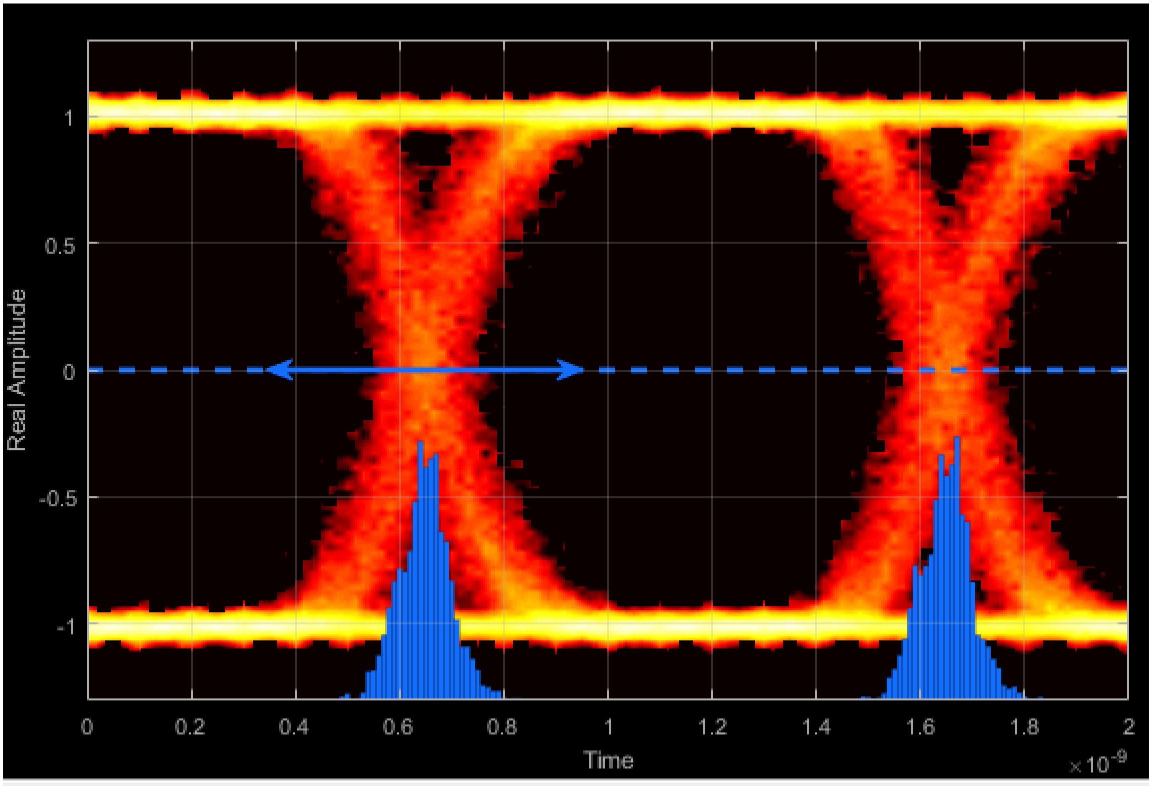
`RMS Jitter` — Standard deviation of jitter

RMS jitter is the standard deviation of the jitter calculated in the horizontal (jitter) histogram at the decision boundary.



`Peak-to-Peak Jitter` — Distance between extreme data points of histogram

Peak-to-peak jitter is the maximum horizontal distance between the left and right nonzero values in the horizontal histogram of each crossing time.



Measurement Settings

To change measurement settings, first select **Enable measurements**. Then, in the **Eye Measurements** pane, click the arrow next to **Settings**. You can control these measurement settings.

`Eye level boundaries` — Time range for calculating eye levels

`[40 60]` (default) | two-element vector

Time range for calculating eye levels, specified as a two-element vector. These values are expressed as a percentage of the symbol duration. Tunable.

`Decision boundary` — Amplitude level threshold

`0` (default) | scalar

Amplitude level threshold in V, specified as a scalar. This parameter separates the different signaling regions for horizontal (jitter) histograms. This parameter is tunable, but the jitter histograms reset when the parameter changes.

For non-return-to-zero (NRZ) signals, set **Decision boundary** to 0. For return-to-zero (RZ) signals, set **Decision boundary** to half the maximum amplitude.

`Rise/Fall Thresholds` — Amplitude levels of the rise and fall transitions

`[10 90]` (default) | two-element vector

Amplitude levels of the rise and fall transitions, specified as a two-element vector. These values are expressed as a percentage of the eye amplitude. This parameter is tunable, but the crossing histograms of the rise and fall thresholds reset when the parameter changes.

`Hysteresis` — Amplitude tolerance of the horizontal crossings

`0` (default) | scalar

Amplitude tolerance of the horizontal crossings in V, specified as a scalar. Increase hysteresis to provide more tolerance to spurious crossings due to noise. This parameter is tunable, but the jitter and the rise and fall histograms reset when the parameter changes.

`BER threshold` — BER used for eye measurements

`1e-12` (default) | nonnegative scalar from 0 to 0.5

BER used for eye measurements, specified as a nonnegative scalar from 0 to 0.5. The value is used to make measurements of random jitter, total jitter, horizontal eye openings, and vertical eye openings. Tunable.

`Bathtub BERs` — BER values used to calculate openings of bathtub curves

`[0.5 0.1 0.01 0.001 0.0001 1e-05 1e-06 1e-07 1e-08 1e-09 1e-10 1e-11 1e-12]` (default) | vector

BER values used to calculate openings of bathtub curves, specified as a vector whose elements range from 0 to 0.5. Horizontal and vertical eye openings are calculated for each of the values specified by this parameter. To enable this parameter, select **Show horizontal bathtub curve**, **Show vertical bathtub curve**, or both. Tunable.

`Measurement delay` — Duration of initial data discarded from measurements

`0` (default) | nonnegative scalar

Duration of initial data discarded from measurements, in seconds, specified as a nonnegative scalar.

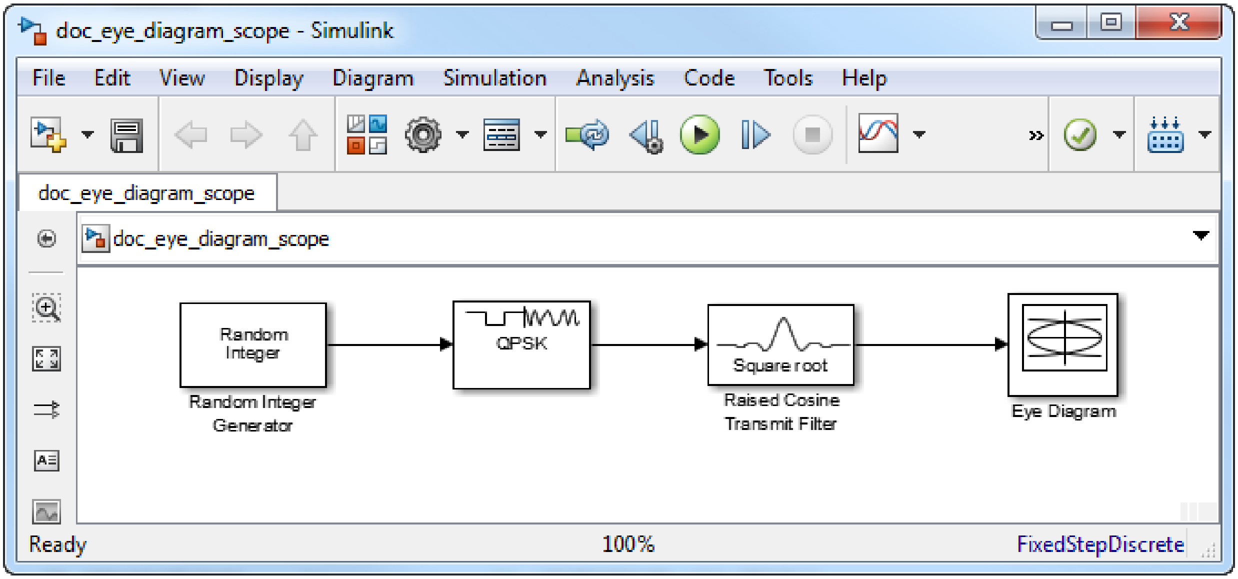
Examples

View Eye Diagram

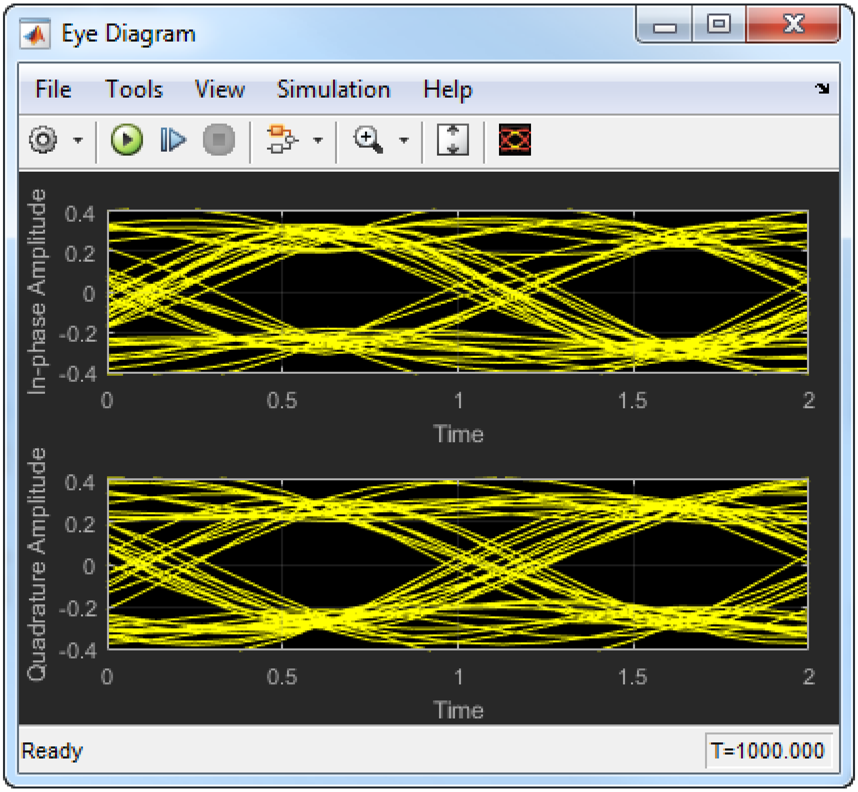
Display the eye diagram of a filtered QPSK signal using the Eye Diagram block.

Load the doc_eye_diagram_scope model from the MATLAB command prompt.

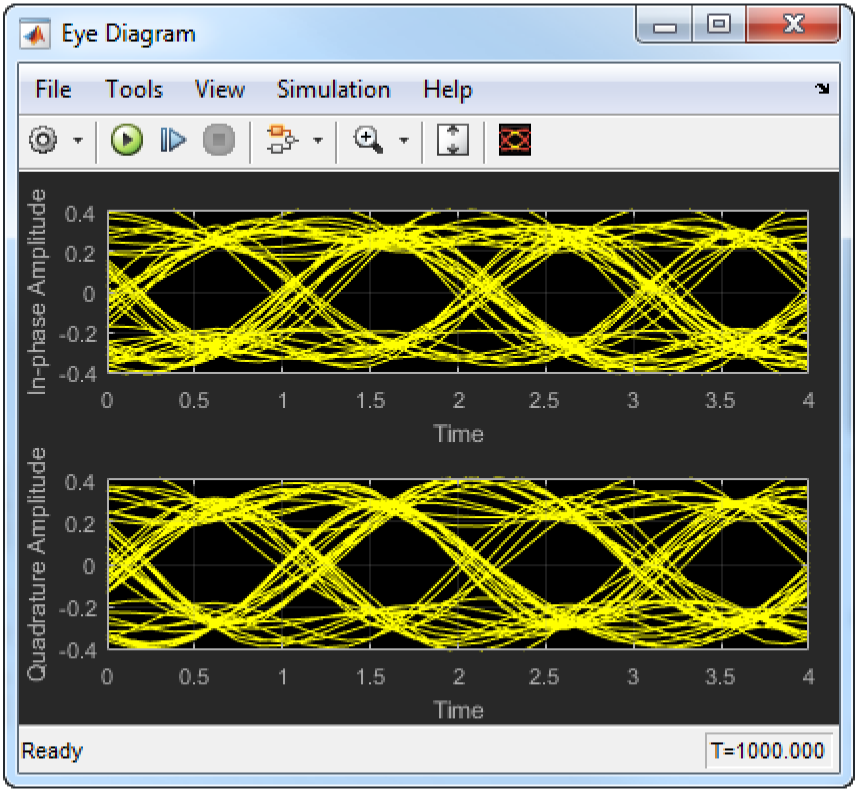
```
doc_eye_diagram_scope
```



Run the model and observe that two symbols are displayed.



Open the configuration parameters dialog box. Change the **Symbols per trace** parameter to 4. Run the simulation and observe that four symbols are displayed.



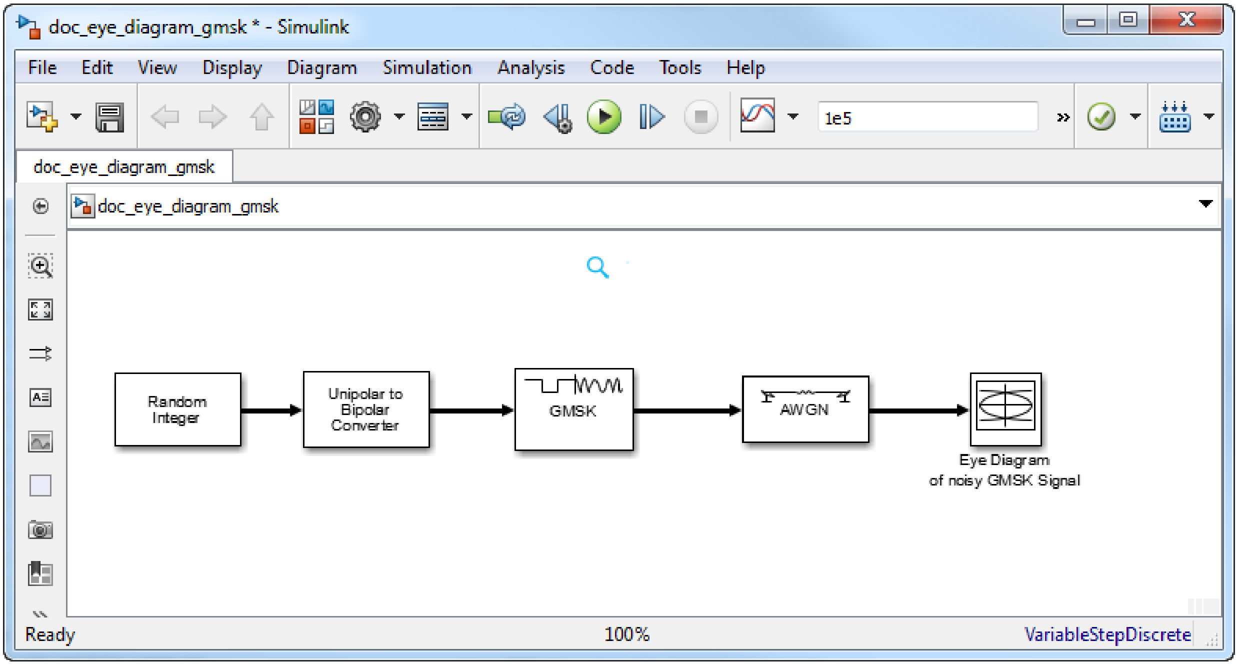
Try changing the Raised Cosine Transmit Filter parameters or changing additional Eye Diagram parameters to see their effects on the eye diagram.

Histogram Plots

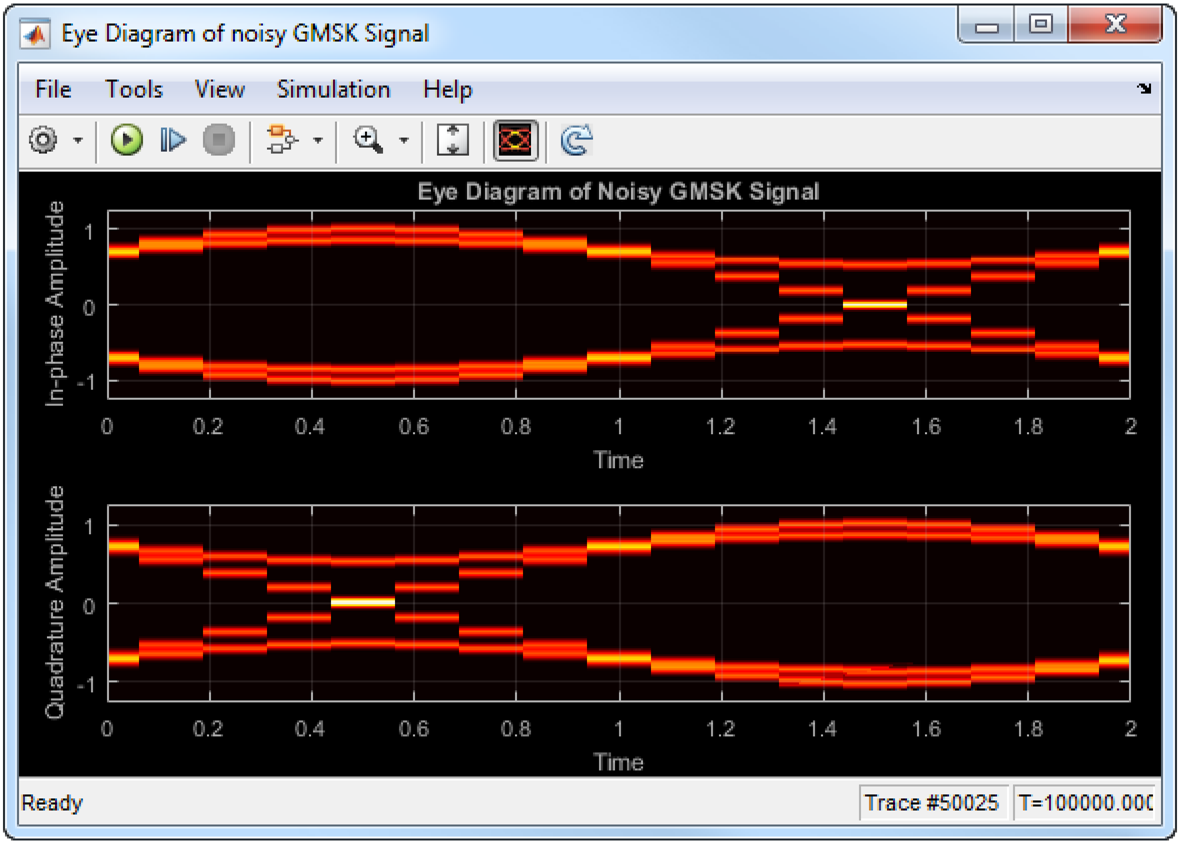
Display histogram plots of a noisy GMSK signal.

Load the doc_eye_diagram_gmsk model from the MATLAB command prompt.

```
doc_eye_diagram_gmsk
```

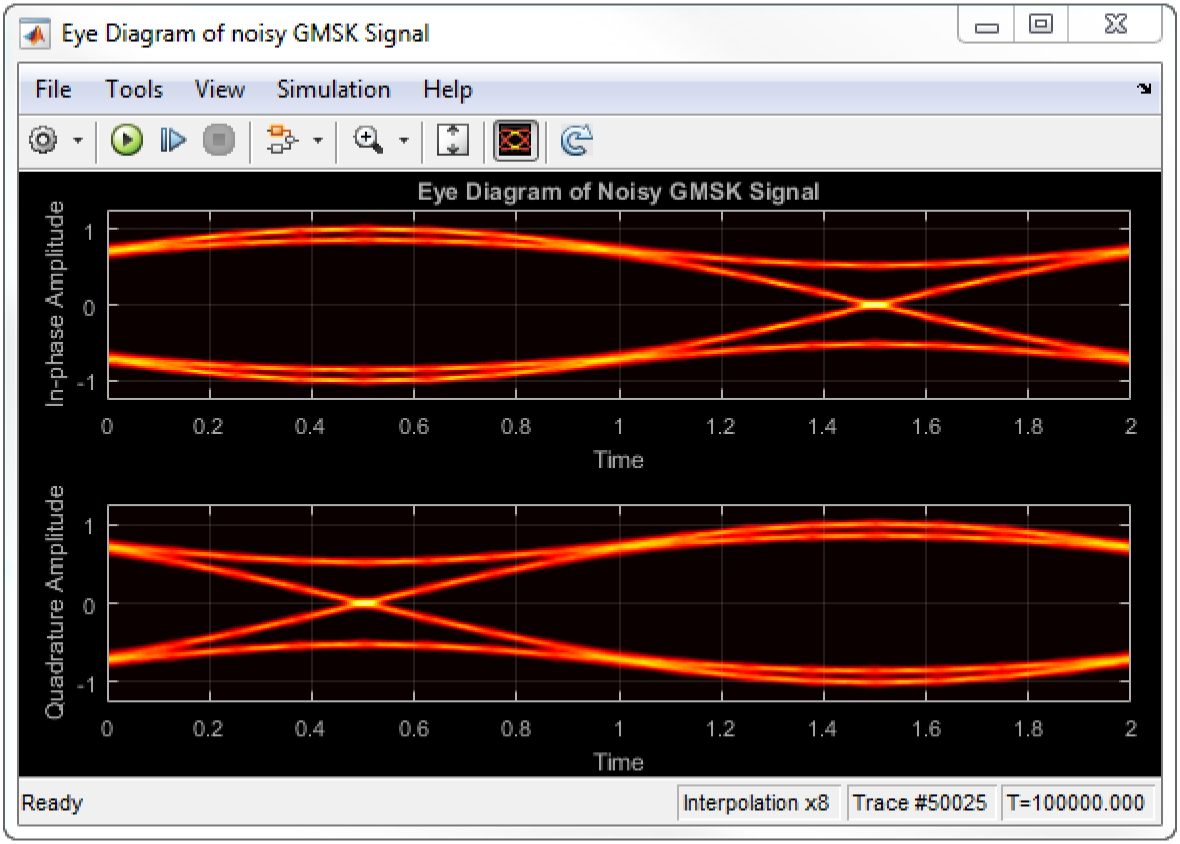


Run the model. The eye diagram is configured to show a histogram without interpolation.



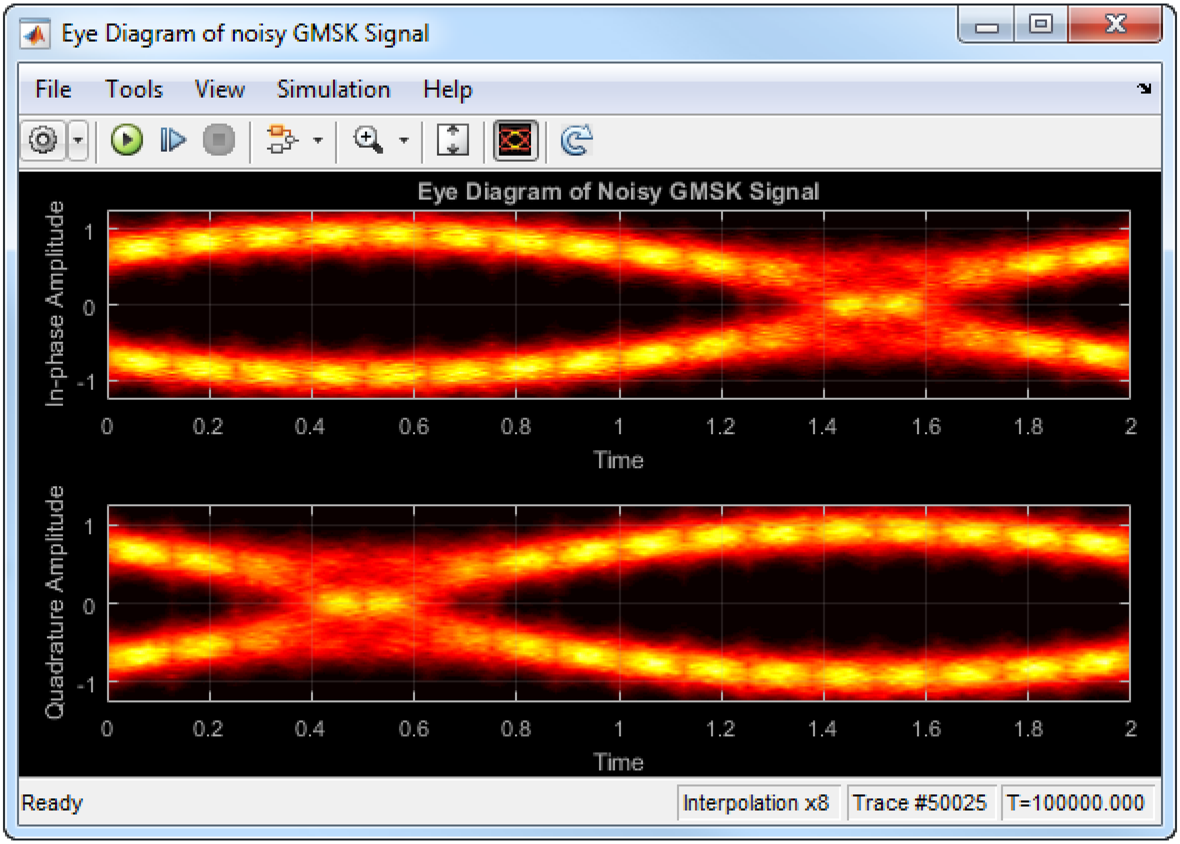
The lack of interpolation results in a plot having piecewise-continuous behavior.

Open the **2D Histogram** tab of the Configuration Properties dialog box. Set the **Oversampling method** to `Input interpolation`. Run the model.



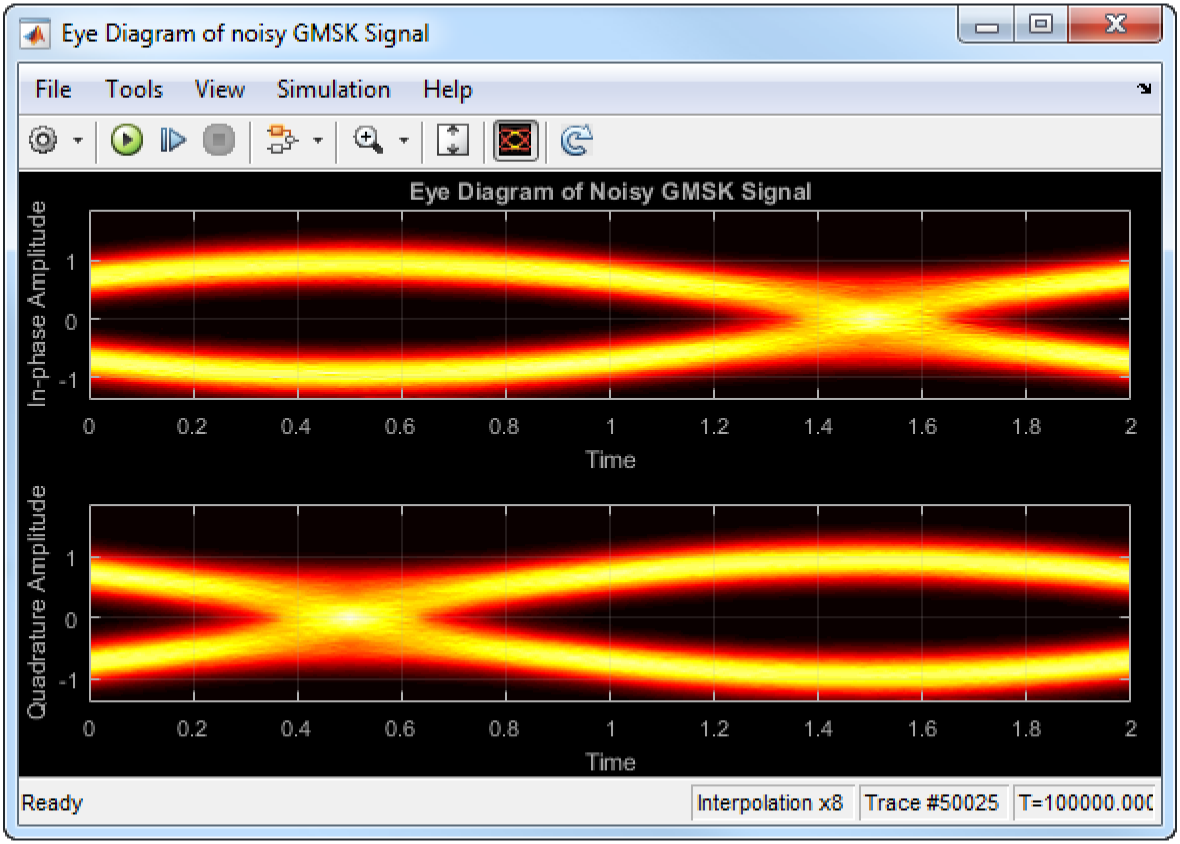
The interpolation smooths the eye diagram.

On the AWGN Channel block, change **SNR (dB)** from 25 to 10. Run the model.



Observe that vertical striping is present in the eye diagram. This striping is the result of input interpolation, which has limited accuracy in low-SNR conditions.

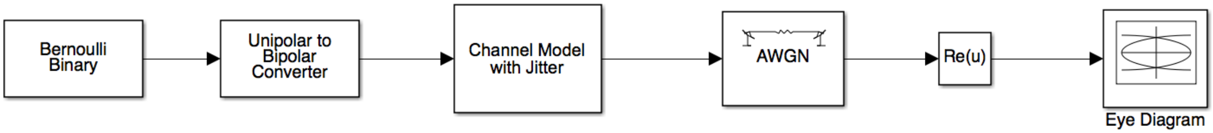
Set the **Oversampling method** to `Histogram interpolation`. Run the model.



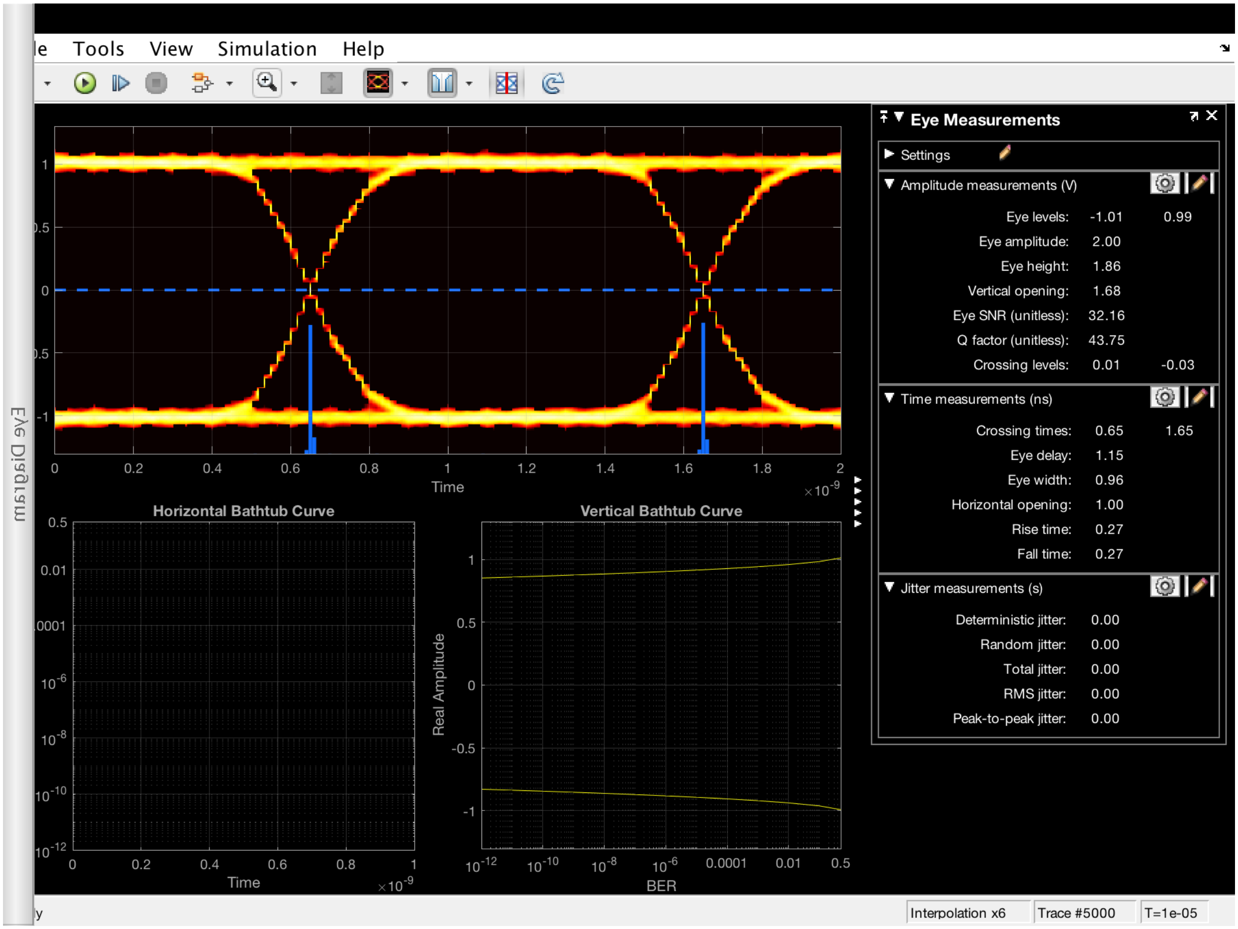
The eye diagram plot now renders accurately because the histogram interpolation method works for all SNR values. This method is not as fast as the other techniques and results in increased execution time.

Visualize Random and Deterministic Jitter

Open the model. The model generates bipolar data, adds deterministic and random jitter, applies white noise, displays the resulting eye diagram.



Run the model.



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

Eye levels:	-1.01	0.99
Eye amplitude:	2.00	
Eye height:	1.86	
Vertical opening:	1.68	
Eye SNR (unitless):	32.16	
Q factor (unitless):	43.75	
Crossing levels:	0.01	-0.03

▼ Time measurements (ns)

Crossing times:	0.65	1.65
Eye delay:	1.15	
Eye width:	0.96	
Horizontal opening:	1.00	
Rise time:	0.27	
Fall time:	0.27	

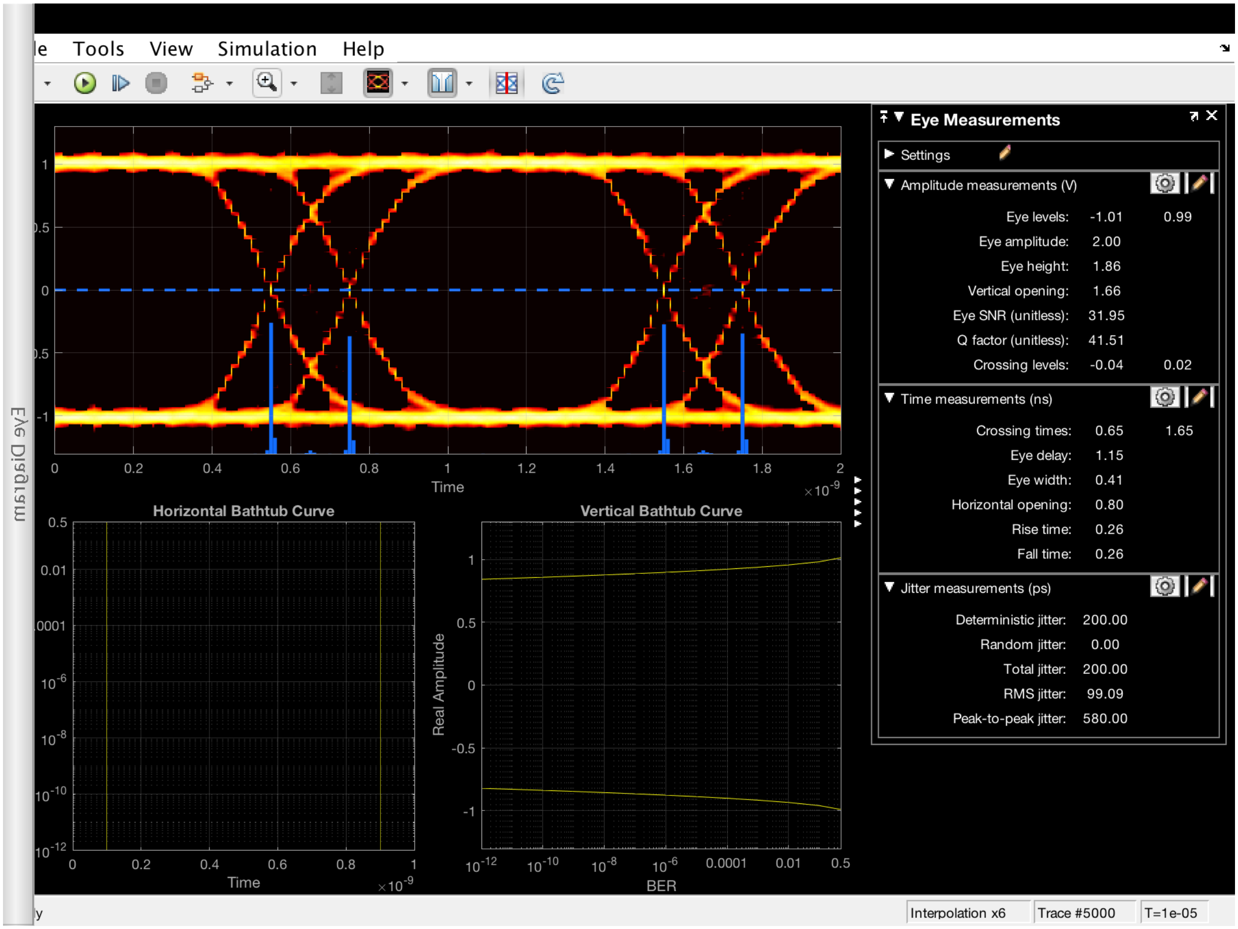
▼ Jitter measurements (s)

Deterministic jitter:	0.00
Random jitter:	0.00
Total jitter:	0.00
RMS jitter:	0.00
Peak-to-peak jitter:	0.00

y Interpolation x6 Trace #5000 T=1e-05

The signal shows clean crossings as there is no jitter.

To show the effect of the deterministic jitter, set the **Deterministic jitter** parameter to `100e-12` in the `Channel Model with Jitter` block. Run the model.



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.86
Vertical opening: 1.66
Eye SNR (unitless): 31.95
Q factor (unitless): 41.51
Crossing levels: -0.04 0.02

Time measurements (ns)

Crossing times: 0.65 1.65
Eye delay: 1.15
Eye width: 0.41
Horizontal opening: 0.80
Rise time: 0.26
Fall time: 0.26

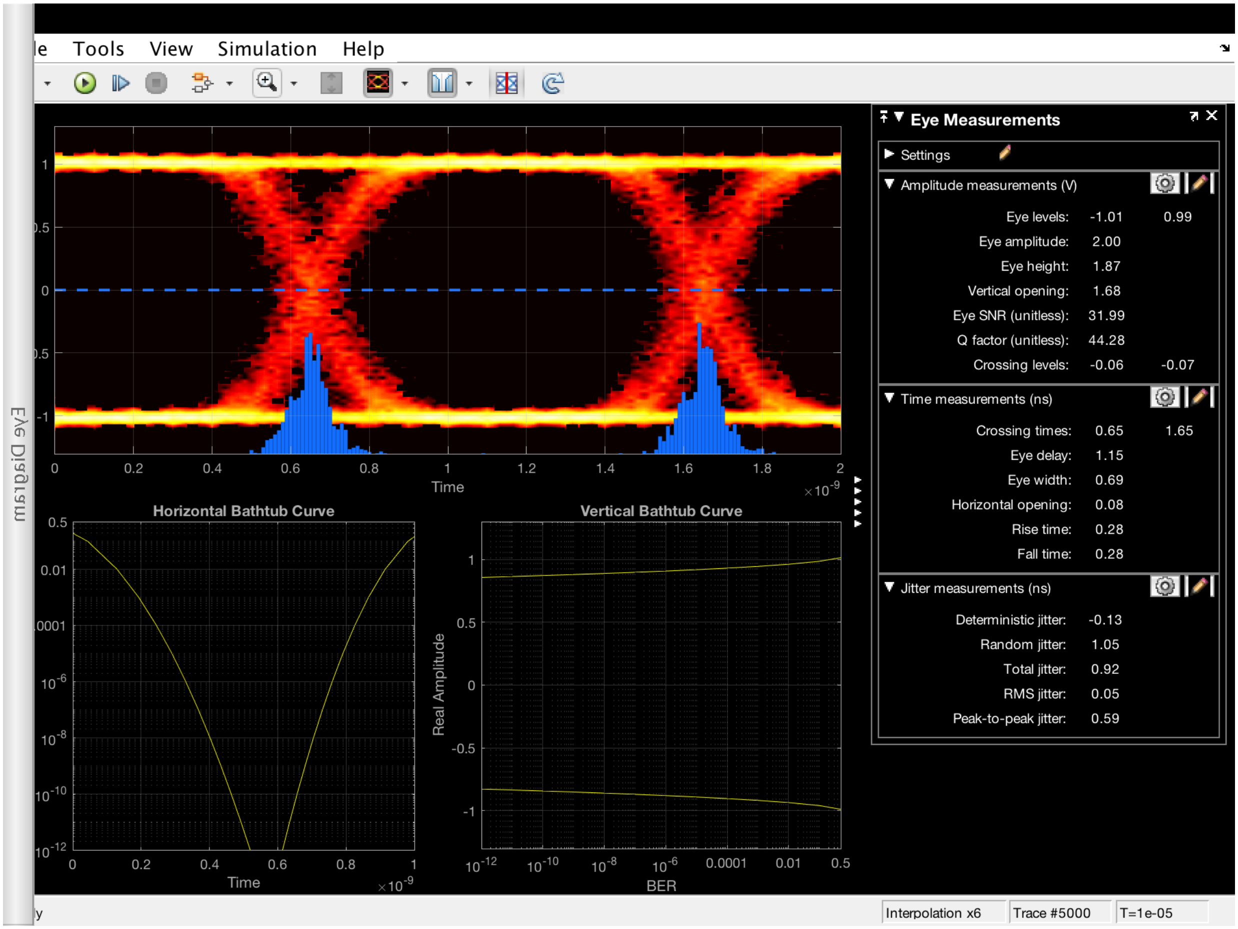
Jitter measurements (ps)

Deterministic jitter: 200.00
Random jitter: 0.00
Total jitter: 200.00
RMS jitter: 99.09
Peak-to-peak jitter: 580.00

y | Interpolation x6 | Trace #5000 | T=1e-05

The deterministic jitter is shown by the separation between the two peaks in the jitter histogram.

To show the effect of the RMS jitter, set the **Deterministic jitter** parameter to `0` and set the **RMS jitter** parameter to `50e-12`. Run the model.



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.87
Vertical opening: 1.68
Eye SNR (unitless): 31.99
Q factor (unitless): 44.28
Crossing levels: -0.06 -0.07

Time measurements (ns)

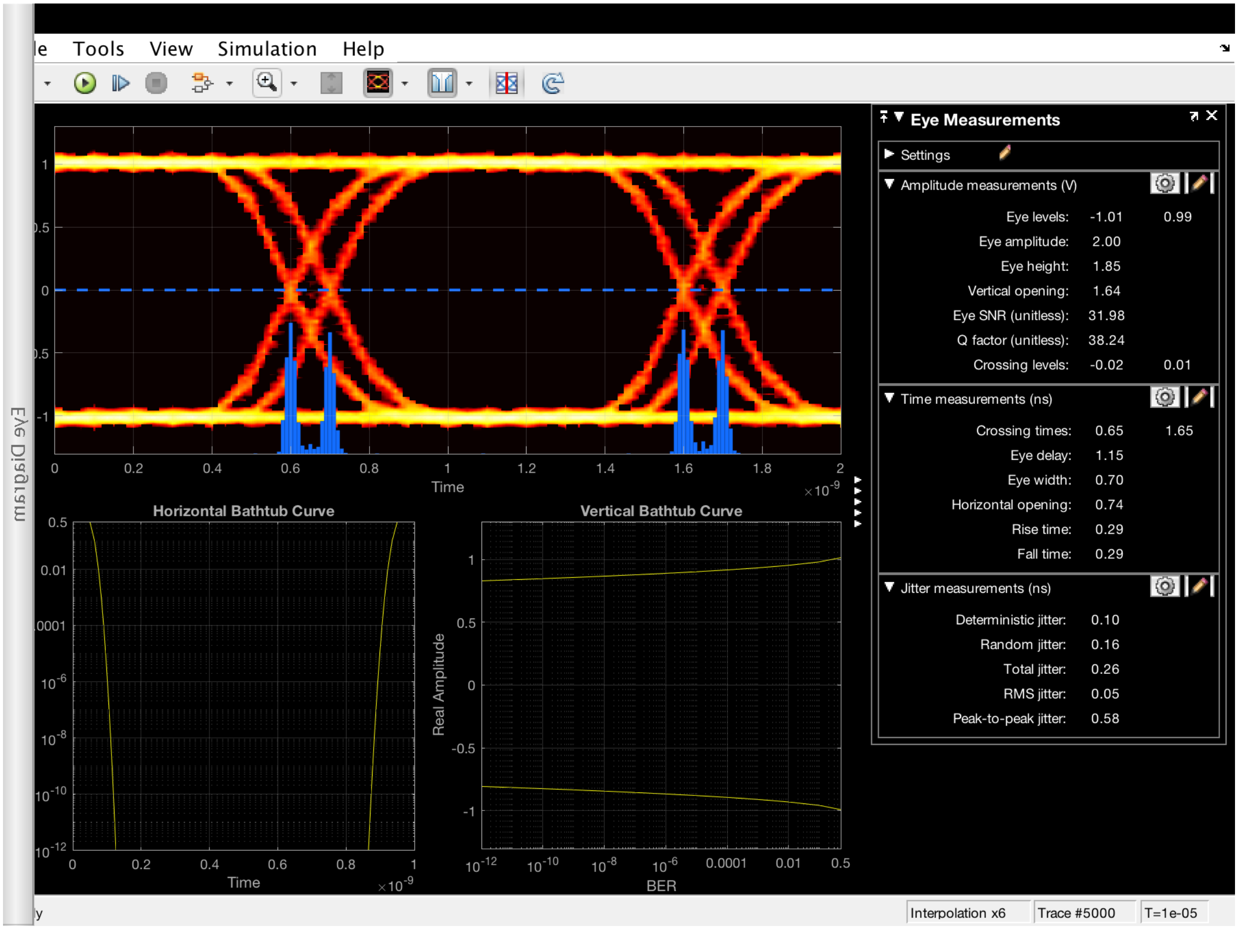
Crossing times: 0.65 1.65
Eye delay: 1.15
Eye width: 0.69
Horizontal opening: 0.08
Rise time: 0.28
Fall time: 0.28

Jitter measurements (ns)

Deterministic jitter: -0.13
Random jitter: 1.05
Total jitter: 0.92
RMS jitter: 0.05
Peak-to-peak jitter: 0.59

The RMS jitter is shown by the fuzziness around each of the crossings.

Set the RMS jitter to `10e-12` and the deterministic jitter to `50e-12`. Run the model.



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

```
               Eye levels:   -1.01     0.99
            Eye amplitude:    2.00
               Eye height:    1.85
         Vertical opening:    1.64
      Eye SNR (unitless):    31.98
     Q factor (unitless):    38.24
          Crossing levels:   -0.02     0.01
```

▼ Time measurements (ns)

```
           Crossing times:    0.65     1.65
                Eye delay:    1.15
                Eye width:    0.70
       Horizontal opening:    0.74
                Rise time:    0.29
                Fall time:    0.29
```

▼ Jitter measurements (ns)

```
     Deterministic jitter:    0.10
            Random jitter:    0.16
             Total jitter:    0.26
               RMS jitter:    0.05
       Peak-to-peak jitter:   0.58
```

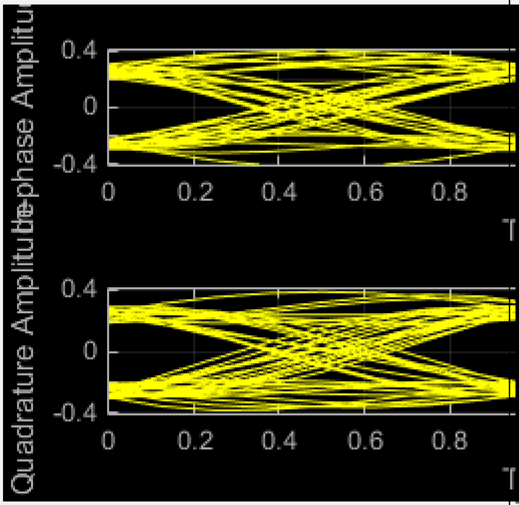
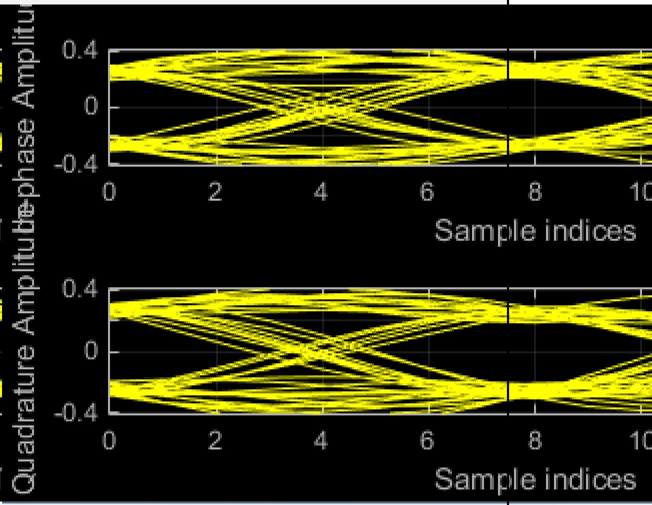
The signal shows the effects of both jitter types.

Definitions

Using Eye Diagram in Conditionally Executed Subsystems

When an Eye Diagram block is placed in a conditionally executed subsystem, for example in a triggered or enabled subsystem:

- Input size must be an integer multiple of `SamplesPerSymbol` * `SymbolsPerTrace`
- Sample offset must be zero
- The rightmost part of the display is intentionally omitted. This figure compares typical eye diagram display when placed in a normal system versus one placed in a conditionally executed subsystem.

Eye Diagram Plot in Normal System	**Eye Diagram Plot in Conditionally Executed Subsystem**
	
In a regular Eye Diagram, the rightmost part is a line between the last sample of a trace and the first sample of the next trace.	In conditionally executed subsystems, these traces may be non-contiguous, thus this rightmost segment could corrupt the display and is omitted.

See Also

Blocks
Constellation Diagram

System Objects
`comm.EyeDiagram`

Introduced in R2014a

Find Delay

Find delay between two signals

sRef
sDel
Find Delay
delay

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.

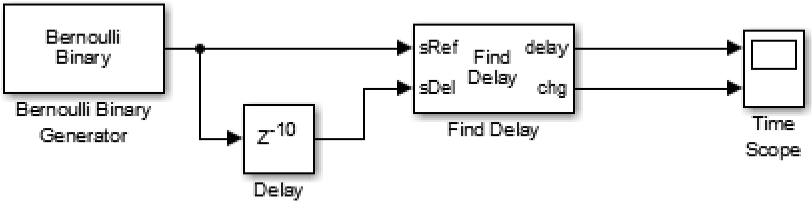
Another example of this usage is in “Delays”.

Finding the Delay to Help Align Words

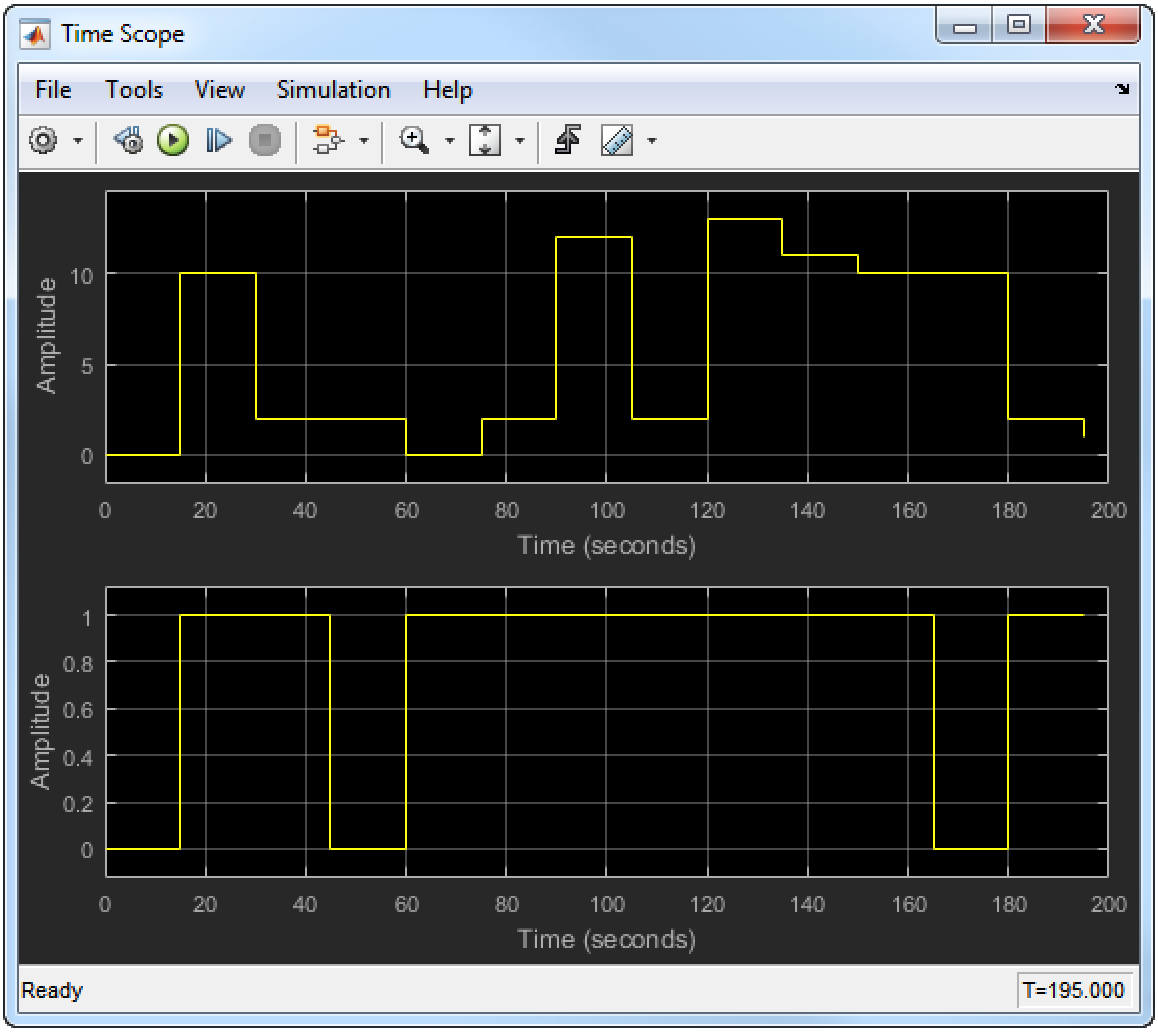
Another typical use of this block is to determine how to align the boundaries of frames with the boundaries of codewords or other types of data blocks. “Delays” describes when such alignment is necessary and also illustrates, in the “Aligning Words of a Block Code” discussion, how to use the Find Delay block to solve the problem.

Setting the Correlation Window Length

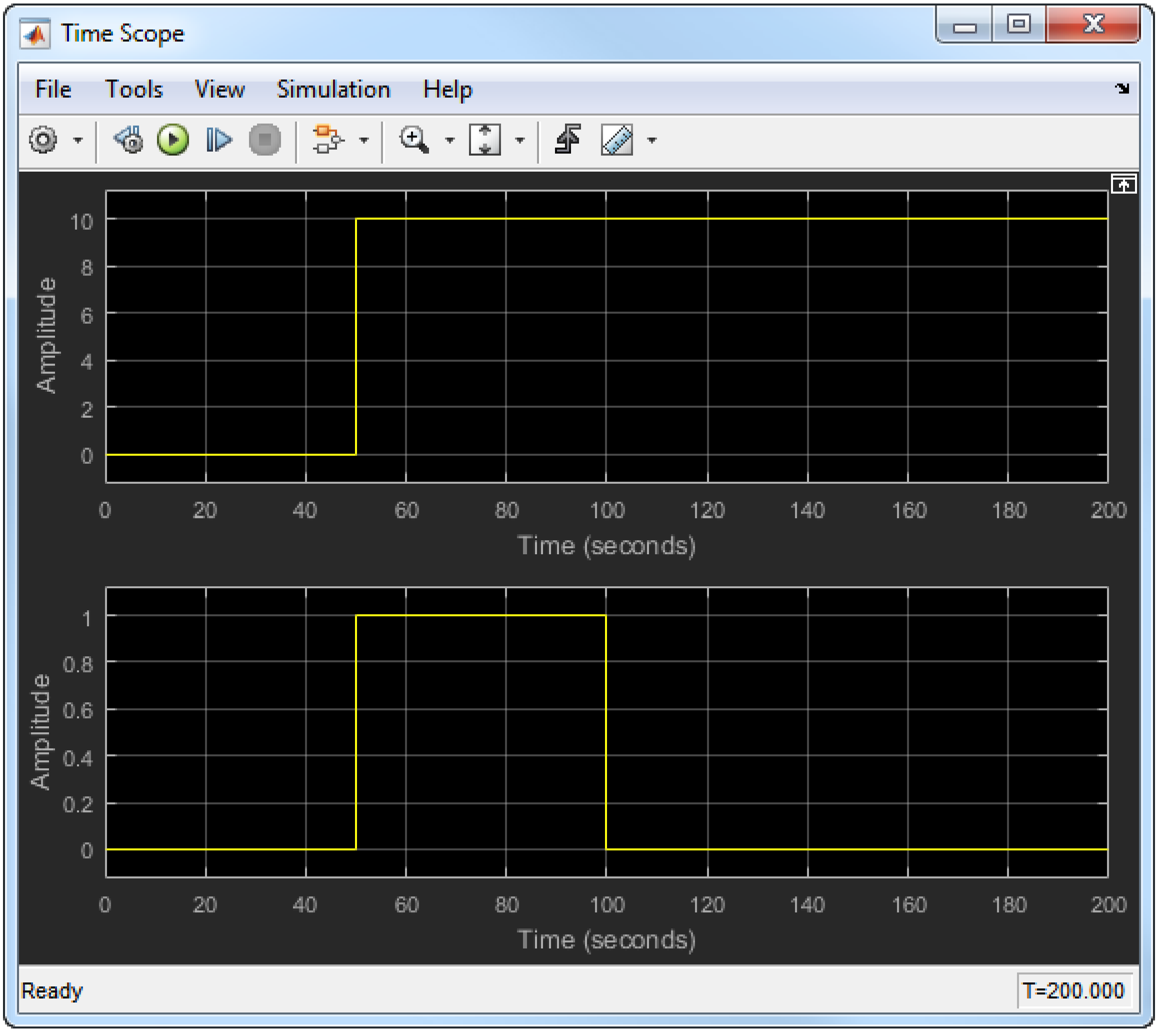
The next example illustrates how to tell when the **Correlation window length** is not sufficiently large. Load this model by typing doc_find_delay_example at the command prompt.



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.



Parameters

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.

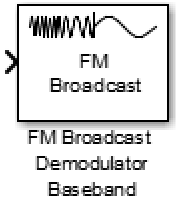
See Also

Align Signals | Error Rate Calculation | `finddelay`

Introduced in R2012a

FM Broadcast Demodulator Baseband

Demodulate using broadcast FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Broadcast Demodulator Baseband block demodulates a complex baseband FM signal by using the conjugate delay method, and filters the signal by using a de-emphasis filter. To demodulate stereo audio using 38 kHz, enable stereo demodulation. To demodulate RBDS signals from the 57 kHz band, enable RBDS demodulation.

Parameters

Sample rate (Hz)

Specify the input signal sample rate as a positive real scalar.

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe.

De-emphasis filter time constant (s)

Specify the de-emphasis lowpass filter time constant in seconds as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe.

Output audio sample rate (Hz)

Specify the output audio sample rate as a positive real scalar.

Play audio device

Select this check box to play sound from a default audio device.

Buffer size (samples)

Specify the buffer size the block uses to communicate with an audio device as a positive integer scalar. This parameter is available only when the **Play audio device** check box is selected.

Stereo audio

Select this check box to enable demodulation of a stereo audio signal. If not selected, the audio signal is assumed to be monophonic.

RBDS demodulation

Select this check box to demodulate the RBDS signal from the input complex baseband FM signal. By default, this check box is not selected.

Number of samples per RBDS symbol

Specify the number of samples of the RBDS output as a positive integer. The RBDS sample rate is given by **Number of samples per RBDS symbol** × 1187.5 Hz. According to the RBDS standard, the sample rate of each bit is 1187.5 Hz.

This parameter appears when you select the **RBDS demodulation** check box.

The default is 10.

RBDS Costas loop

Specify whether a Costas loop is used to recover the phase of the RBDS signal. Select this check box for radio stations that do not lock the 57 kHz RBDS signal in phase with the third harmonic of the 19 kHz pilot tone.

This parameter appears when you select the **RBDS demodulation** check box.

By default, this check box is not selected.

Simulate using

Select the type of simulation to run.

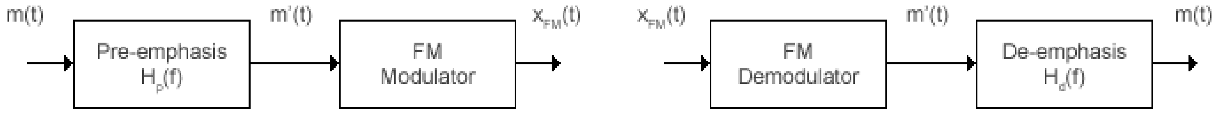
- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

The FM Broadcast demodulator includes the functionality of the baseband FM demodulator, de-emphasis filtering, and the ability to receive stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMDemodulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



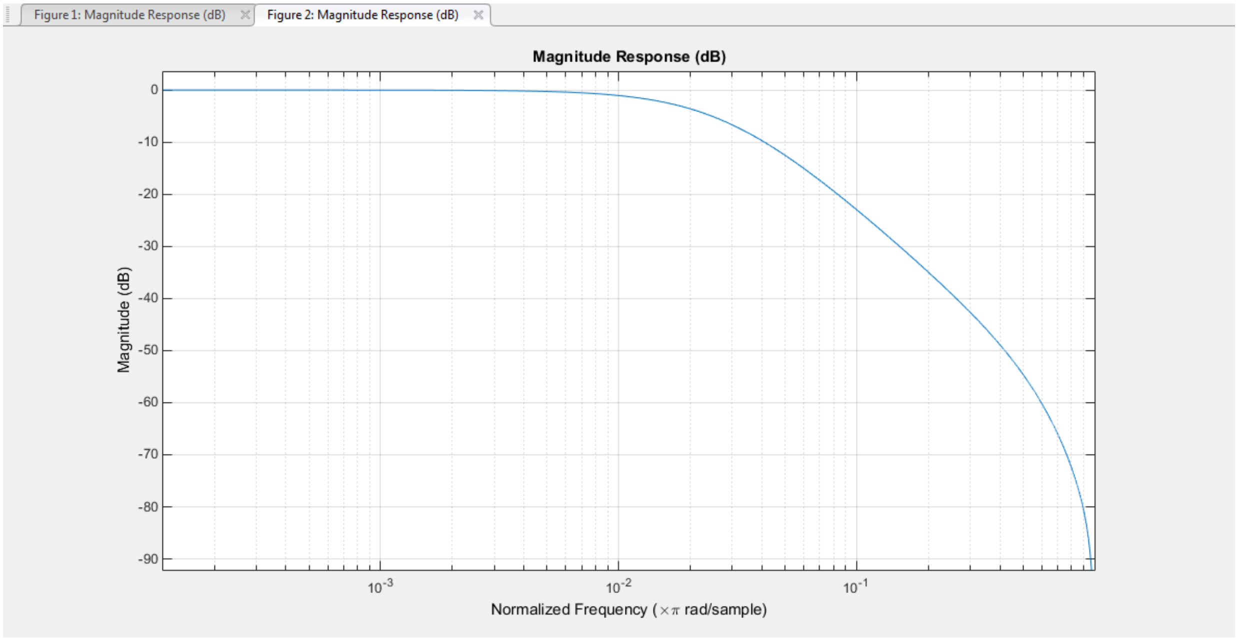
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f \tau_s \ ,$$

where τ_s is the filter time constant. The time constant is 50 μs in Europe and 75 μs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s}.$$

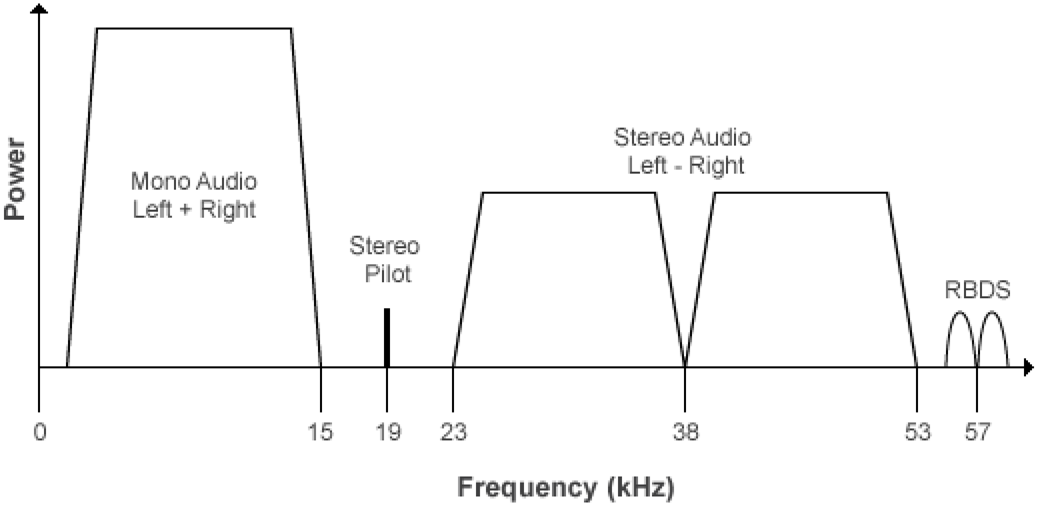
For an audio sample rate of 44.1 kHz, the de-emphasis filter has the following response.



Stereo and RDS/RBDS FM — Multiplex Signal

The FM broadcast demodulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals.

Here is the spectrum of the multiplex baseband signal, *m(t)*.



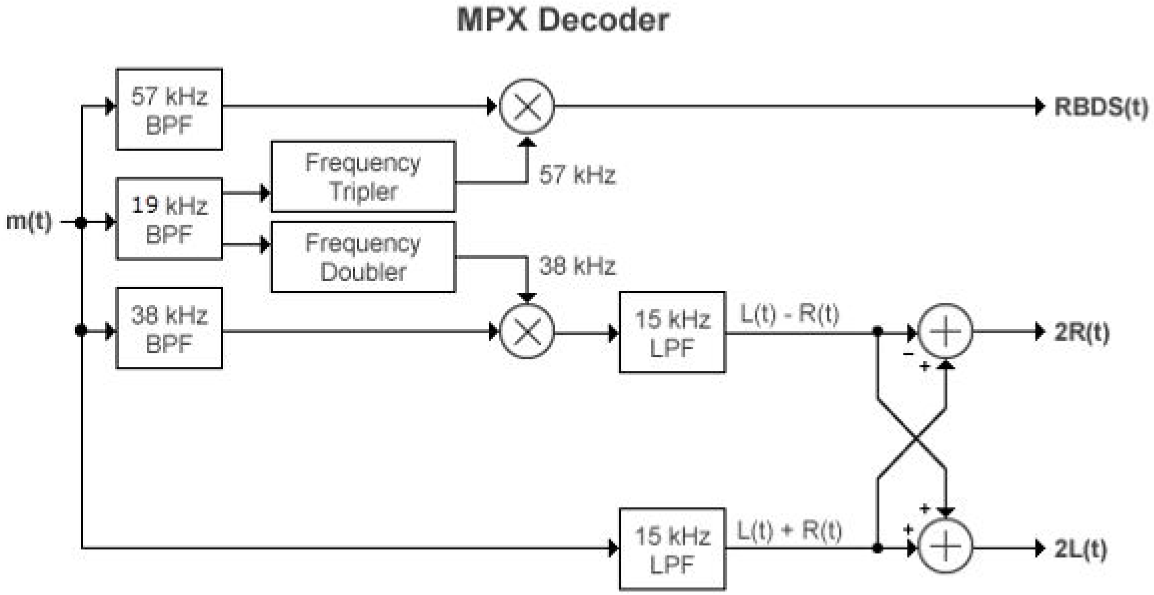
m(t) is given by

$$m(t) = C_0\left[L(t) + R(t)\right] + C_1\cos(2\pi \times 19kHz \times t) + C_0\left[L(t) - R(t)\right]\cos(2\pi \times 38kHz \times t) + C_2 RBDS(t)\cos(2$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

The demodulator applies *m(t)* to three bandpass filters with center frequencies at 19, 38, and 57 kHz, and to a lowpass filter with a 3-dB cutoff frequency of 15 kHz. The 19 kHz bandpass filter extracts the pilot tone from the modulated signal. The recovered pilot tone is doubled and tripled in frequency to produce the 38 kHz and 57 kHz signals, which demodulate the (*L* – *R*) and RDS/RBDS signals, respectively. To generate a scaled version of the left and right channels that produce the stereo sound, the (*L* + *R*) and (*L* – *R*) signals are added and subtracted. The RDS/RBDS signal is recovered by mixing with the 57 kHz signal.

Here is the block diagram of the FM broadcast demodulator.

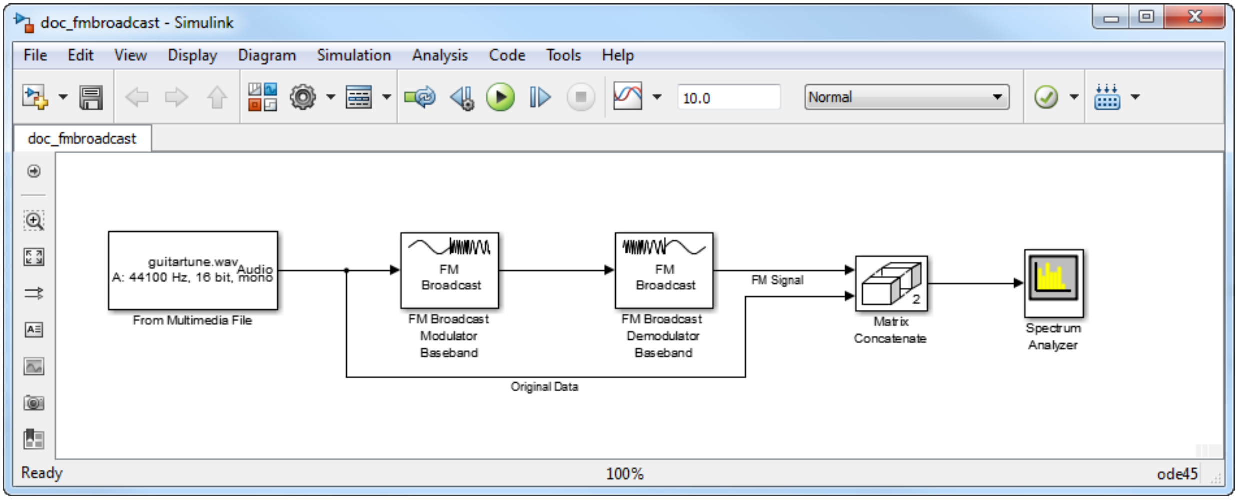


Examples

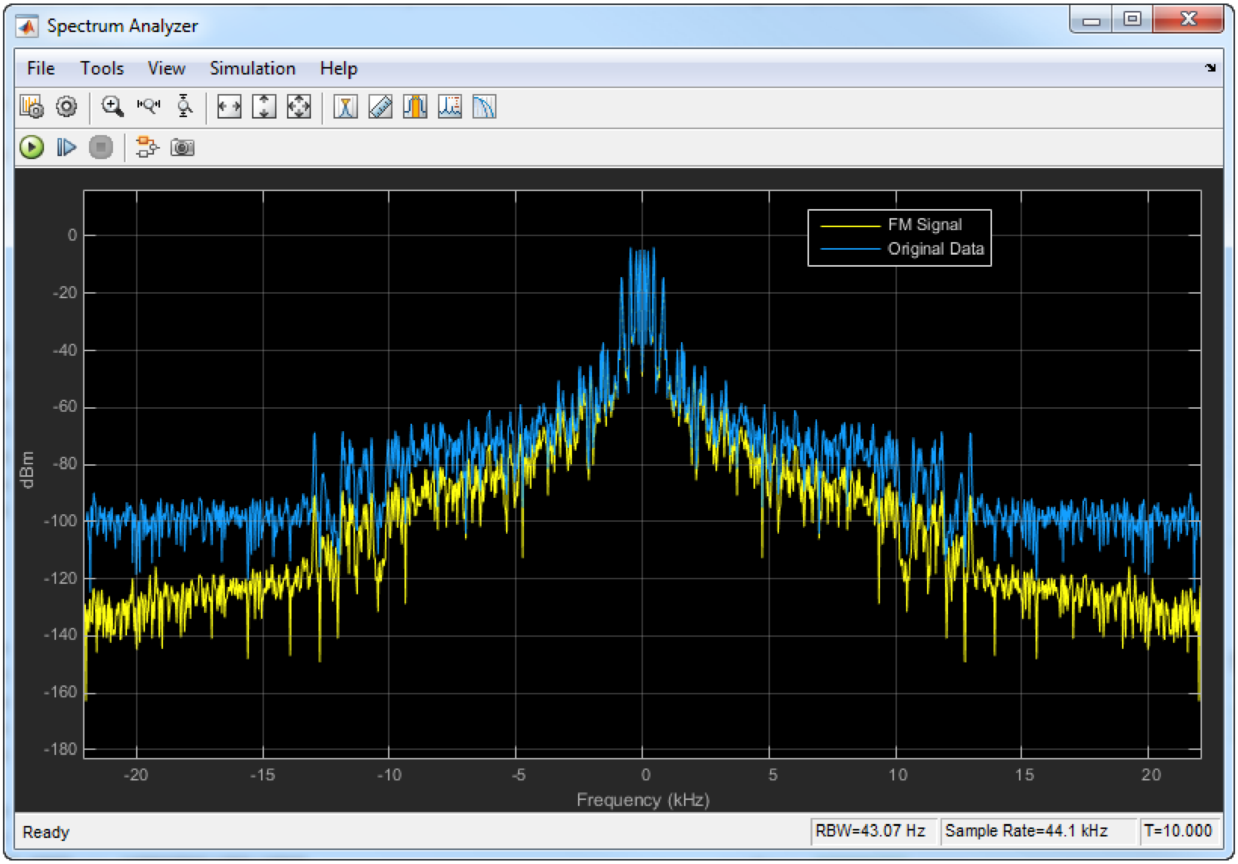
Modulate and Demodulate an Audio Signal

Load an audio input file, modulate and demodulate using the FM broadcast blocks. Compare the input signal spectrum with the demodulated signal spectrum.

Open the doc_fmbroadcast model.



Run the model. The spectrum of the baseband FM signal is attenuated at the higher frequencies relative to the original waveform.



Experiment with the model by changing the **Frequency deviation (Hz)** and the **Pre-emphasis filter time constant (s)** parameters on the modulator and demodulator and observe the impact on the FM signal spectrum.

Limitations

The input length must be an integer multiple of the audio decimation factor. If the **RBDS demodulation** check box is selected, the input length in addition must be an integer multiple of the RBDS decimation factor.

Supported Data Types

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

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

See Also

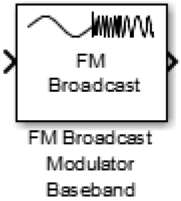
System Objects
`comm.FMBroadcastDemodulator` | `comm.FMDemodulator` | `comm.RBDSWaveformGenerator`

Blocks
FM Broadcast Modulator Baseband

Introduced in R2015a

FM Broadcast Modulator Baseband

Modulate using broadcast FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Broadcast Modulator Baseband block pre-emphasizes an audio signal and modulates it onto a baseband FM signal. If you select the **Stereo audio** check box, the block modulates the stereo audio (*L*–*R*) at the 38 kHz band, in addition to the baseband (*L* +*R*). If you select the **RBDS modulation** check box, the block also modulates a baseband RBDS signal at 57 kHz. For more details, see “Algorithms” on page 2-326.

Parameters

Sample rate (Hz)

Specify the output signal sample rate as a positive real scalar.

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe.

Pre-emphasis filter time constant (s)

Specify the pre-emphasis highpass filter time constant as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe.

Sample rate of audio input signal (Hz)

Specify the input audio sample rate as a positive real scalar.

Stereo audio

Select this check box if the input signal is a stereophonic audio signal.

RBDS modulation

Select this check box to modulate a baseband RBDS signal at 57 kHz. By default, this check box is not selected.

Oversampling factor of RBDS input

Specify the number of samples per RBDS symbol as a positive integer. The RBDS sample rate is given by **Oversampling factor of RBDS input** × `1187.5` Hz. According to the RBDS standard, the sample rate of each bit is 1187.5 Hz.

This parameter appears when you select the **RBDS modulation** check box.

The default is 10.

Simulate using

Select the type of simulation to run.

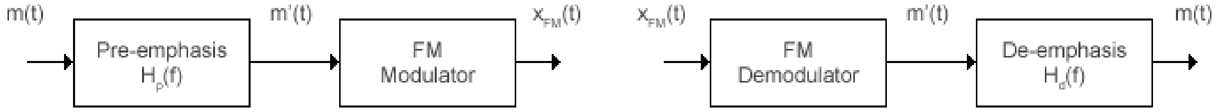
- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

The FM Broadcast modulator includes the functionality of the baseband FM modulator, pre-emphasis filtering, and the ability to transmit stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMModulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



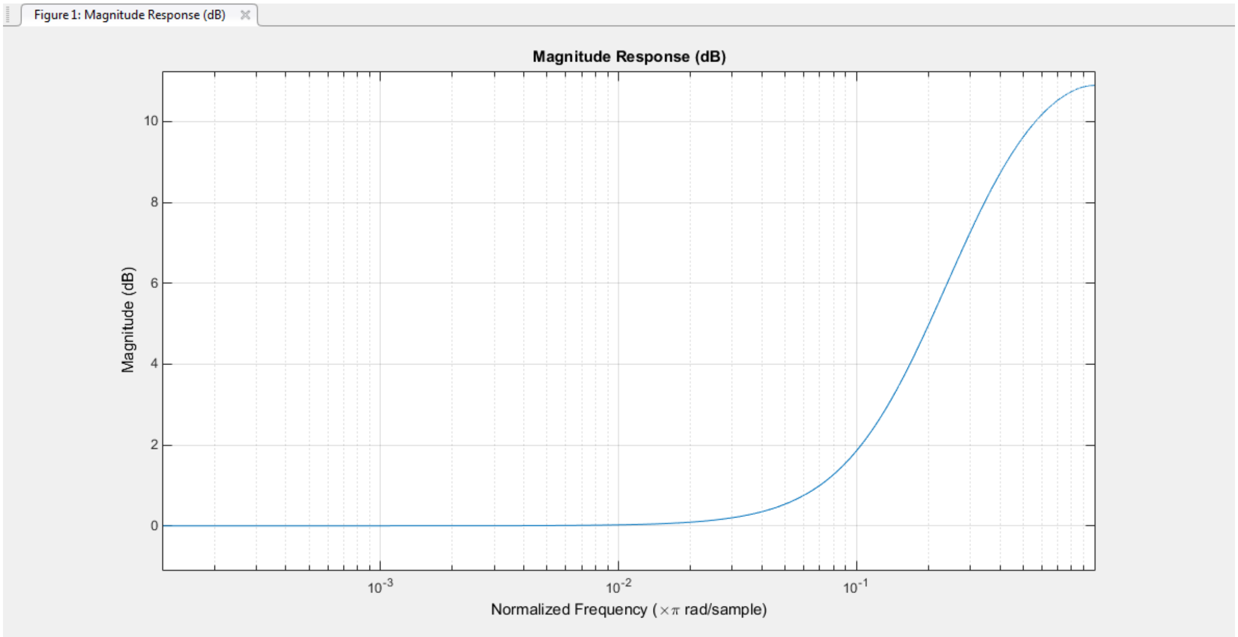
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s \ ,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

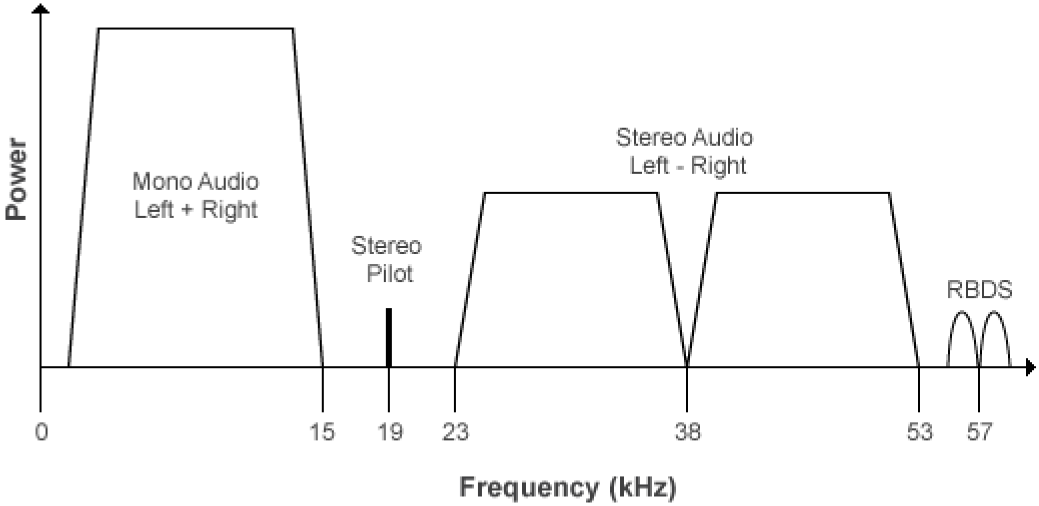
$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s} \ .$$

Irrespective of the audio sampling rate, the signal is converted to a 152 kHz output sampling rate. For an audio sample rate of 44.1 kHz, the pre-emphasis filter has the following response.

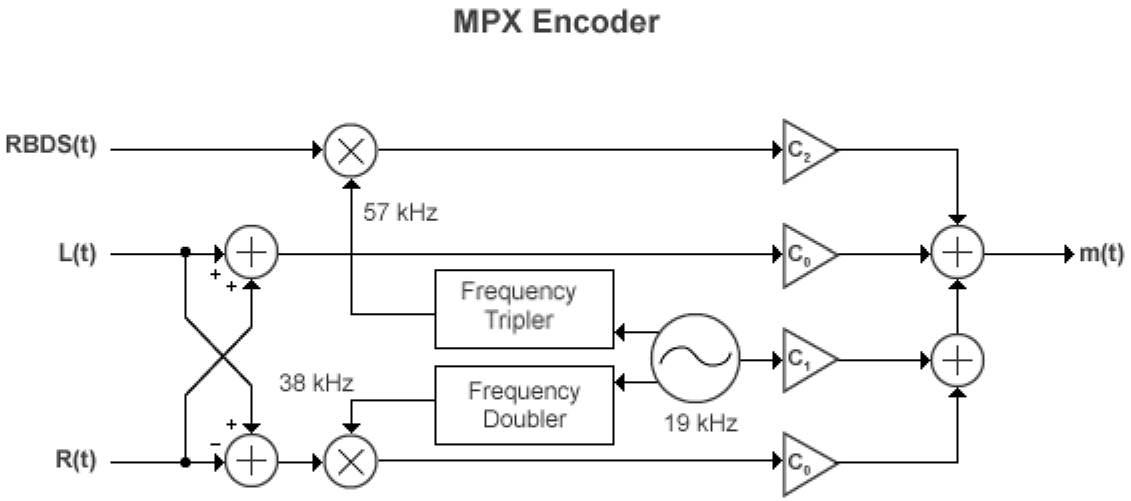


Stereo and RDS/RBDS FM – Multiplex Signal

The FM broadcast modulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals. Here is the spectrum of the multiplex baseband signal.



Here is the block diagram of the FM broadcast modulator, which is used to generate the multiplex baseband signal. *L(t)* and *R(t)* denote the time-domain waveforms from the left and right channels. *RBDS(t)* denotes the time-domain waveform of the RDS/RBDS signal.



The multiplex message signal, *m(t)* is given by

$$m(t) = C_0\left[L(t)+R(t)\right]+C_1\cos(2\pi\times19kHz\times t)+C_0\left[L(t)-R(t)\right]\cos(2\pi\times38kHz\times t)+C_2RBDS(t)\cos(2$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

Limitations

- If you select the **RBDS modulation** check box, both the audio and RBDS inputs must satisfy the following equation:

$$\frac{audioLength}{audioSampleRate}=\frac{RBDSLength}{RBDSSampleRate}$$

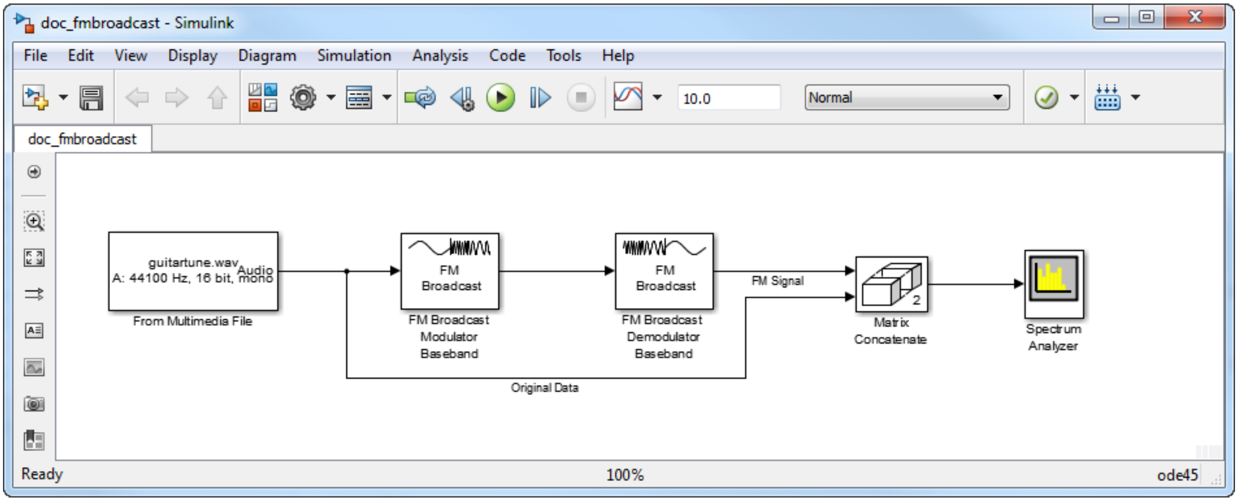
- The input length of the audio signal must be an integer multiple of the audio decimation factor. The input length of the RBDS signal must be an integer multiple of the RBDS decimation factor.

Examples

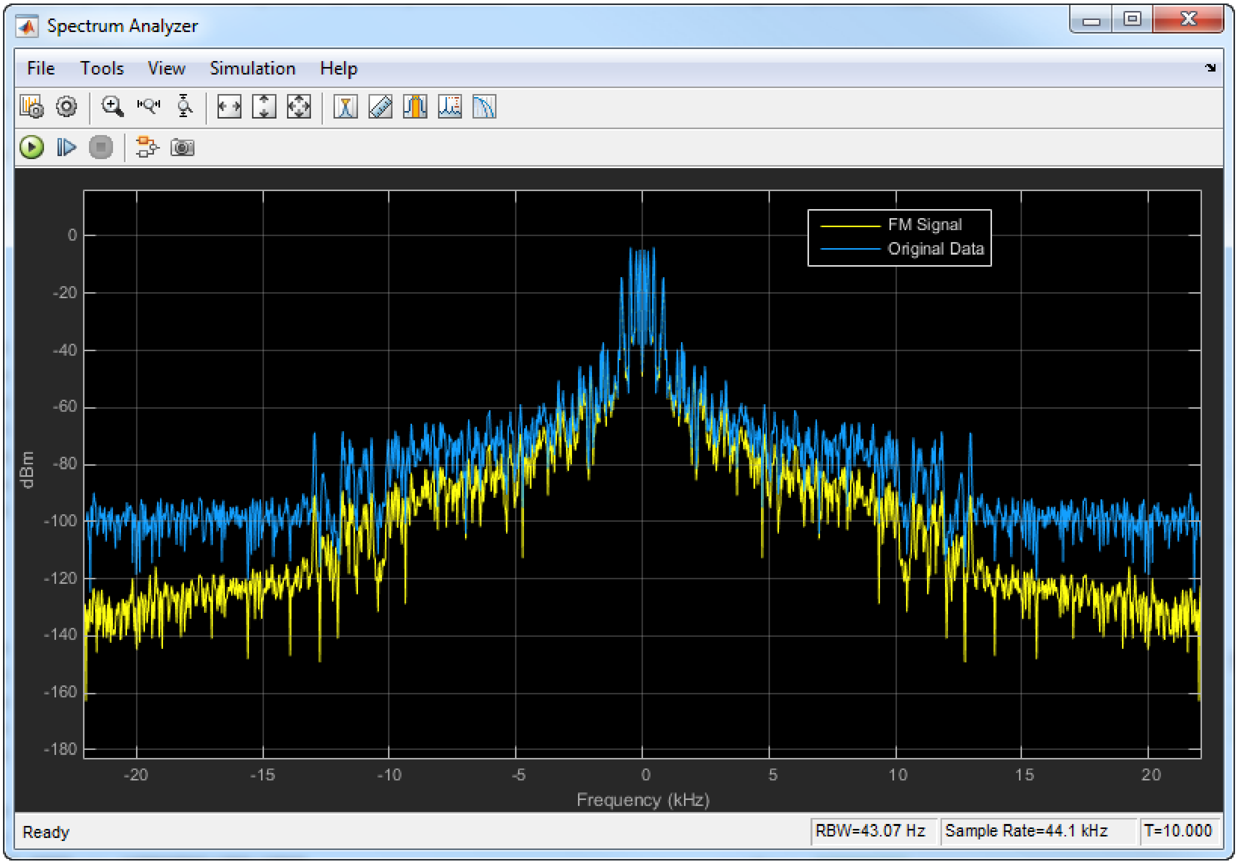
Modulate and Demodulate an Audio Signal

Load an audio input file, modulate and demodulate using the FM broadcast blocks. Compare the input signal spectrum with the demodulated signal spectrum.

Open the doc_fmbroadcast model.



Run the model. The spectrum of the baseband FM signal is attenuated at the higher frequencies relative to the original waveform.



Experiment with the model by changing the **Frequency deviation (Hz)** and the **Pre-emphasis filter time constant (s)** parameters on the modulator and demodulator and observe the impact on the FM signal spectrum.

Supported Data Types

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

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

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

See Also

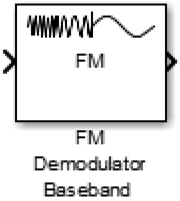
System Objects
`comm.FMBroadcastModulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Blocks
FM Broadcast Demodulator Baseband

Introduced in R2015a

FM Demodulator Baseband

Demodulate using FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Demodulator Baseband block demodulates a sample- or frame-based complex input signal. The block returns a real output signal.

Parameters

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth.

Simulate using

Select the type of simulation to run.

- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

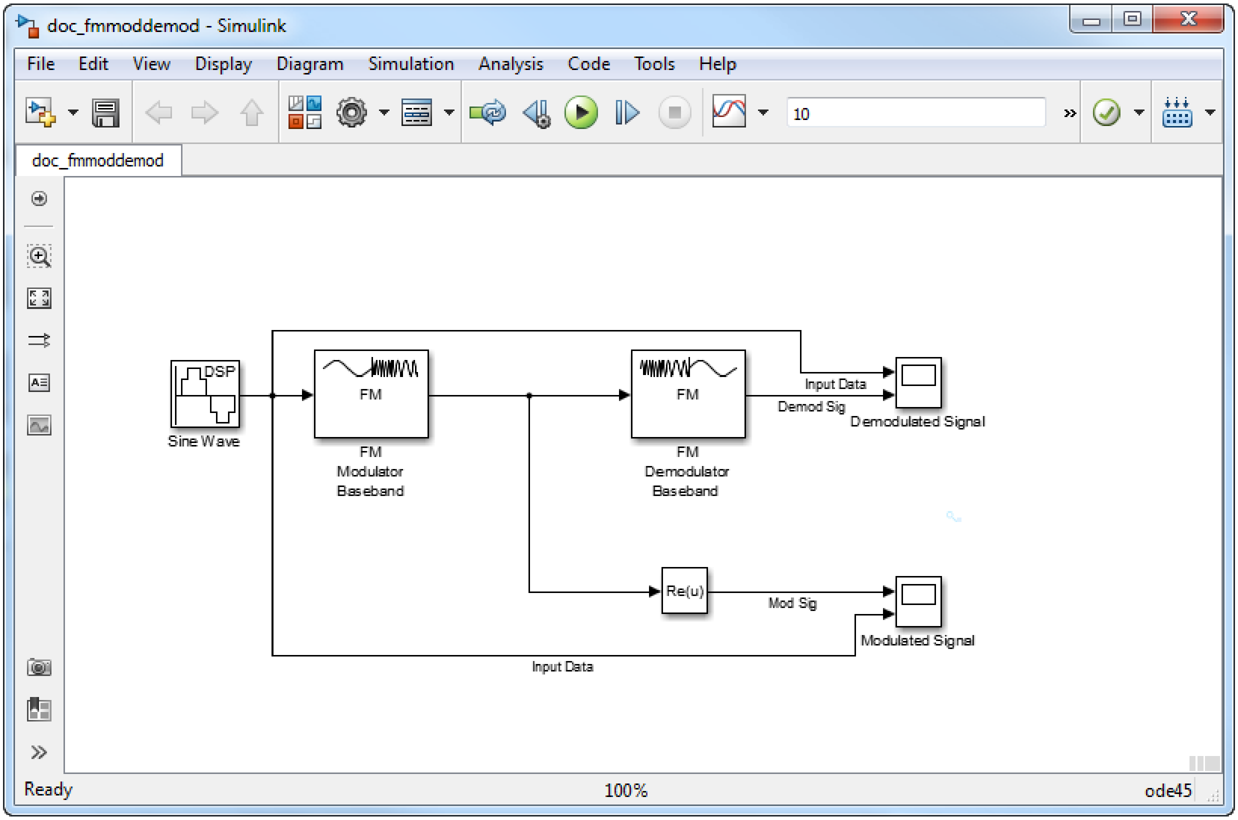
Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.FMDemodulator` reference page. The object properties correspond to the block parameters.

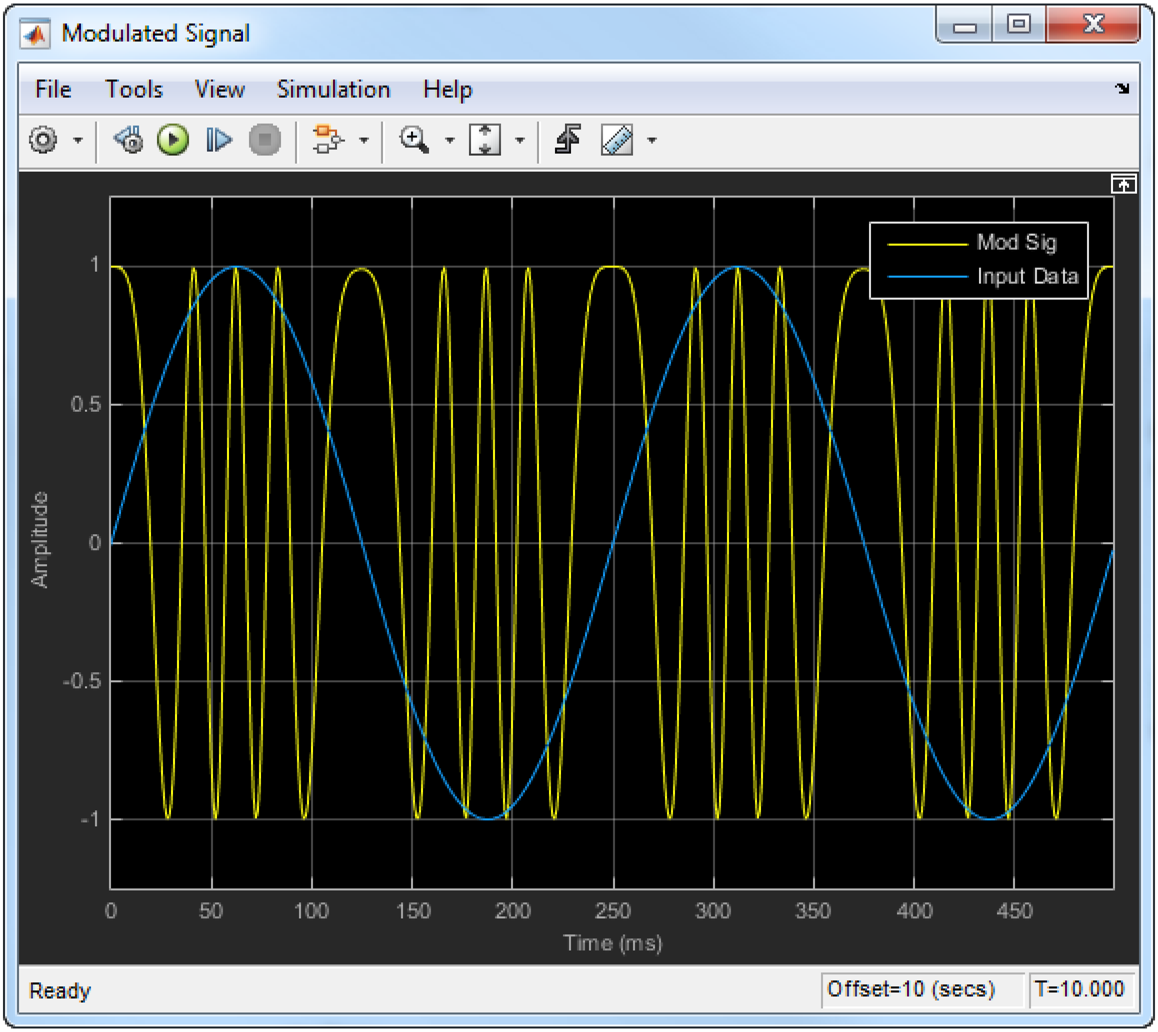
Examples

FM Modulation and Demodulation

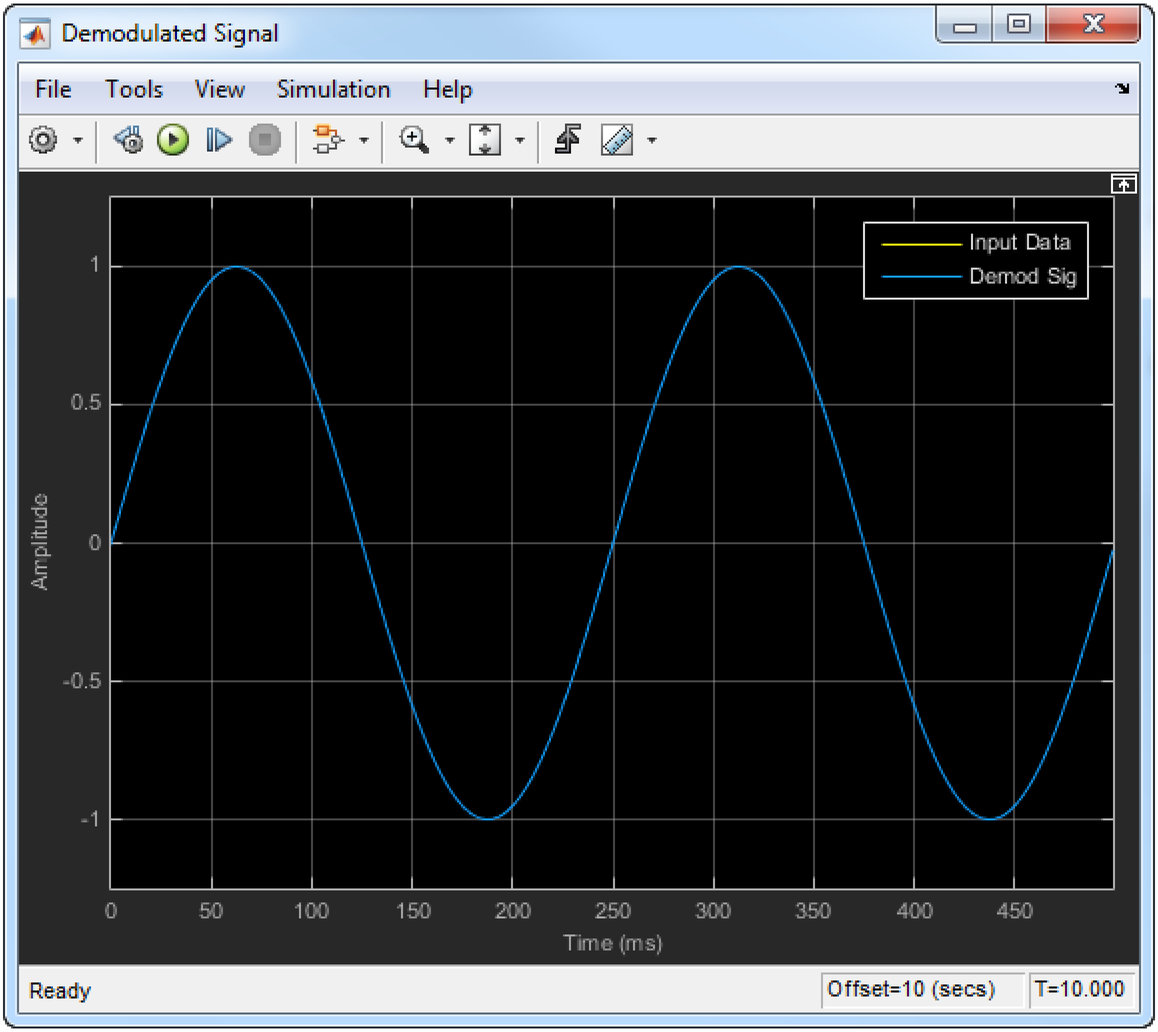
The example, doc_fmmoddemod, shows how the FM Modulator Baseband and FM Demodulator Baseband blocks are used to modulate and demodulate a sinusoidal signal.



The input data is a sine wave of frequency 4 Hz and amplitude 1 V. The frequency deviation is set to 50 Hz. The Modulated Signal scope illustrates that the frequency of the modulator output, `Mod Sig`, varies with the amplitude of the input data.



The Demodulated Signal scope demonstrates that the output of the demodulator, `Demod Sig`, is perfectly aligned with the input data.


Demodulated Signal
File
Tools
View
Simulation
Help
Input Data
Demod Sig
Amplitude
1
0.5
0
-0.5
-1
0
50
100
150
200
250
300
350
400
450
Time (ms)
Ready
Offset=10 (secs)
T=10.000

Supported Data Types

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

Pair Block

FM Modulator Baseband

See Also

`comm.FMDemodulator`

Selected Bibliography

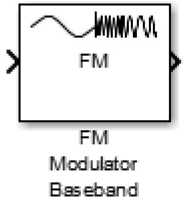
[1] Chakrabarti, I. H., and Hatai, I. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

Introduced in R2015a

FM Modulator Baseband

Modulate using FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Modulator Baseband block applies frequency modulation to a sample- or frame-based real input signal. The block returns a complex output signal.

Parameters

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth.

Simulate using

Select the type of simulation to run.

- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

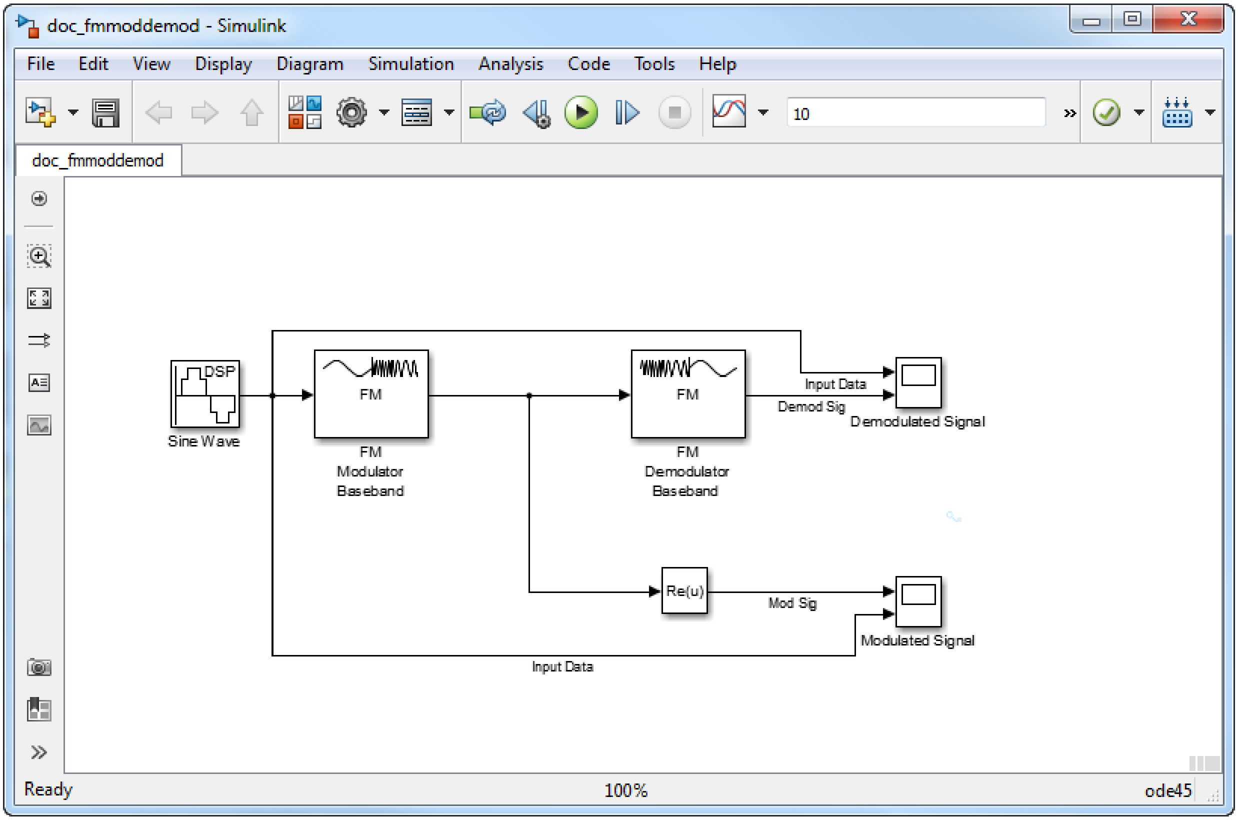
Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.FMModulator` reference page. The object properties correspond to the block parameters.

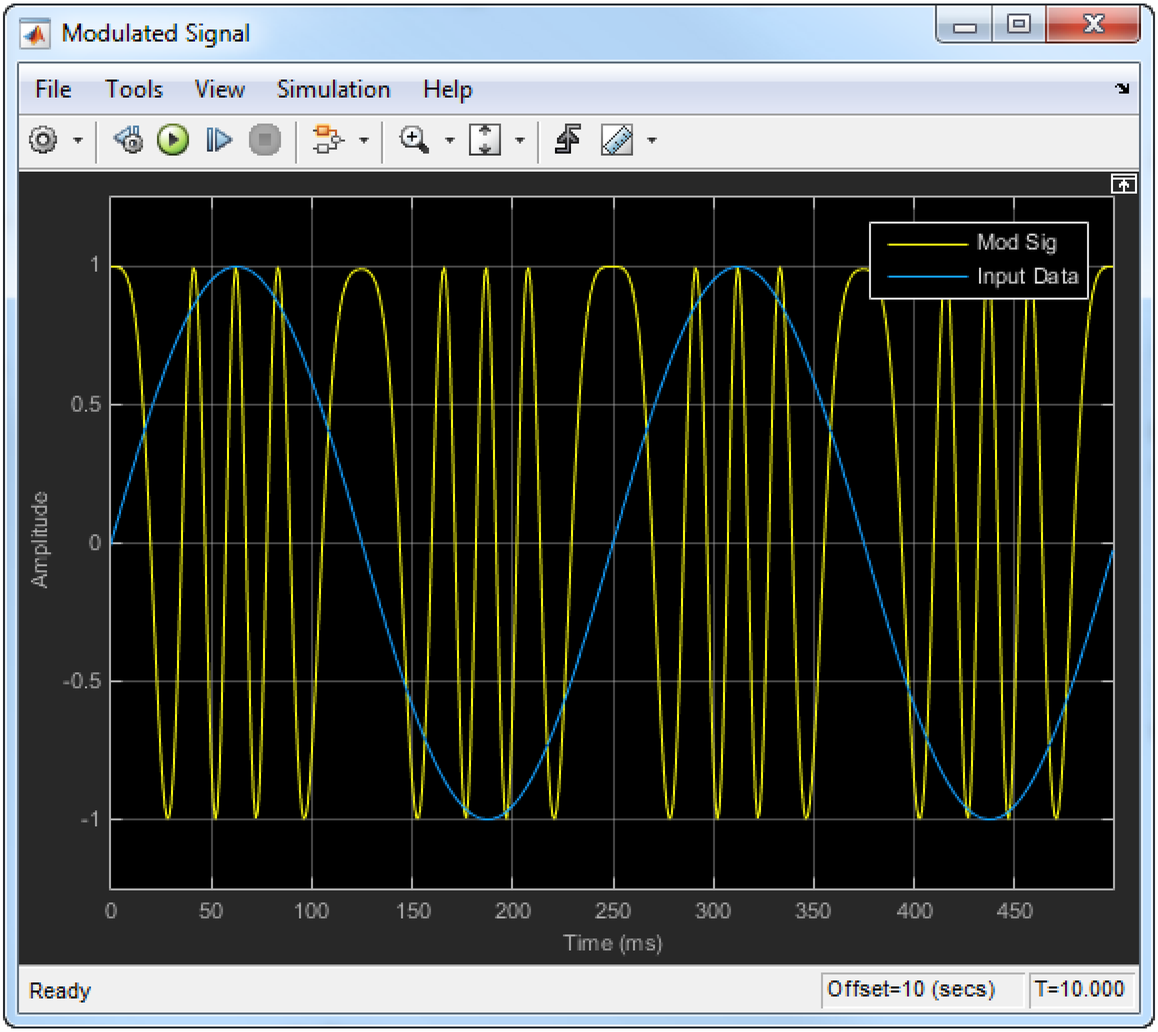
Examples

FM Modulation and Demodulation

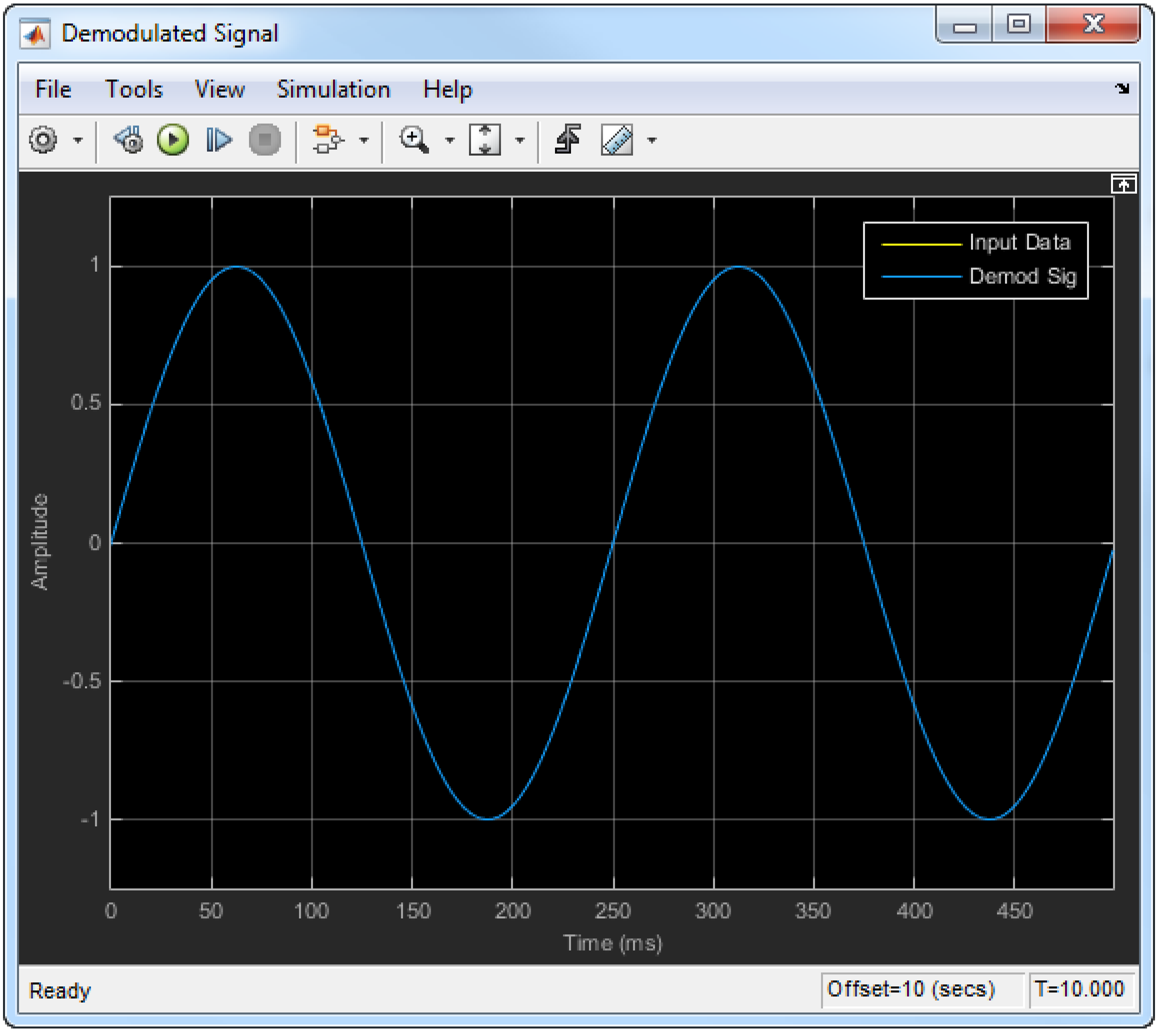
The example, doc_fmmoddemod, shows how the FM Modulator Baseband and FM Demodulator Baseband blocks are used to modulate and demodulate a sinusoidal signal.



The input data is a sine wave of frequency 4 Hz and amplitude 1 V. The frequency deviation is set to 50 Hz. The Modulated Signal scope illustrates that the frequency of the modulator output, `Mod Sig`, varies with the amplitude of the input data.



The Demodulated Signal scope demonstrates that the output of the demodulator, `Demod Sig`, is perfectly aligned with the input data.


Demodulated Signal
File
Tools
View
Simulation
Help
Input Data
Demod Sig
1
0.5
0
-0.5
-1
Amplitude
0
50
100
150
200
250
300
350
400
450
Time (ms)
Ready
Offset=10 (secs)
T=10.000

Supported Data Types

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

Pair Block

FM Demodulator Baseband

See Also

`comm.FMModulator`

Selected Bibliography

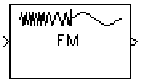
[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems*. New York: McGraw-Hill, 1971, pp. 142–155.

Introduced in R2015a

FM Demodulator Passband

Demodulate FM-modulated data



Library

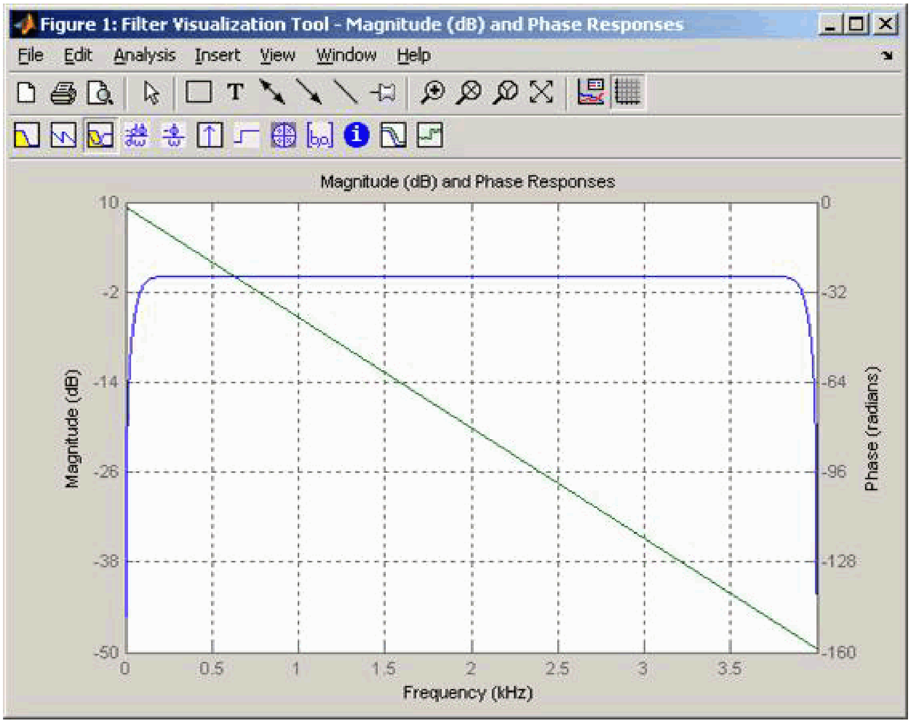
Analog Passband Modulation, in Modulation

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.

Parameters

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.

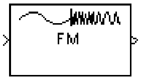
Pair Block

FM Modulator Passband

Introduced before R2006a

FM Modulator Passband

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.

Parameters

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

Introduced before R2006a

Free Space Path Loss

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.

Parameters

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`.

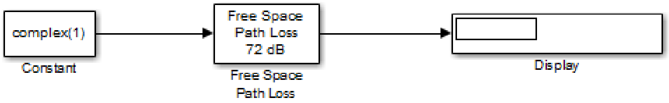
Carrier frequency (MHz)

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

Introduced before R2006a

Gardner Timing Recovery

Recover symbol timing phase using Gardner's method

Note Gardner Timing Recovery has been removed. Use the Symbol Synchronizer block instead.

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.

Parameters

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 (Simulink) 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

$$
\begin{aligned}
e(k) &= a_I(k) + a_Q(k) \\
a_I(k) &= \left\{ y_I((k-1)T + d_{k-1}) - y_I(kT + d_k) \right\} y_I(kT - T/2 + d_{k-1}) \\
a_Q(k) &= \left\{ y_Q((k-1)T + d_{k-1}) - y_Q(kT + d_k) \right\} y_Q(kT - T/2 + d_{k-1})
\end{aligned}
$$

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

Notice from the expressions in curly braces above that the timing error detector approximates the derivative of y using finite differences.

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

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

Symbol Synchronizer

Introduced before R2006a

Gaussian Filter

Filter input signal, possibly downsampling, using Gaussian FIR filter

Library

Comm Filters

Description

Note The Gaussian Filter block is not recommended. Use the `gaussdesign` function and either the Discrete FIR Filter, FIR Interpolation, or FIR Decimation block instead.

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

$$h(t) = \frac{\exp\left(\frac{-t^2}{2\delta^2}\right)}{\sqrt{2\pi}\cdot\delta}$$

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.

Parameters

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.

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

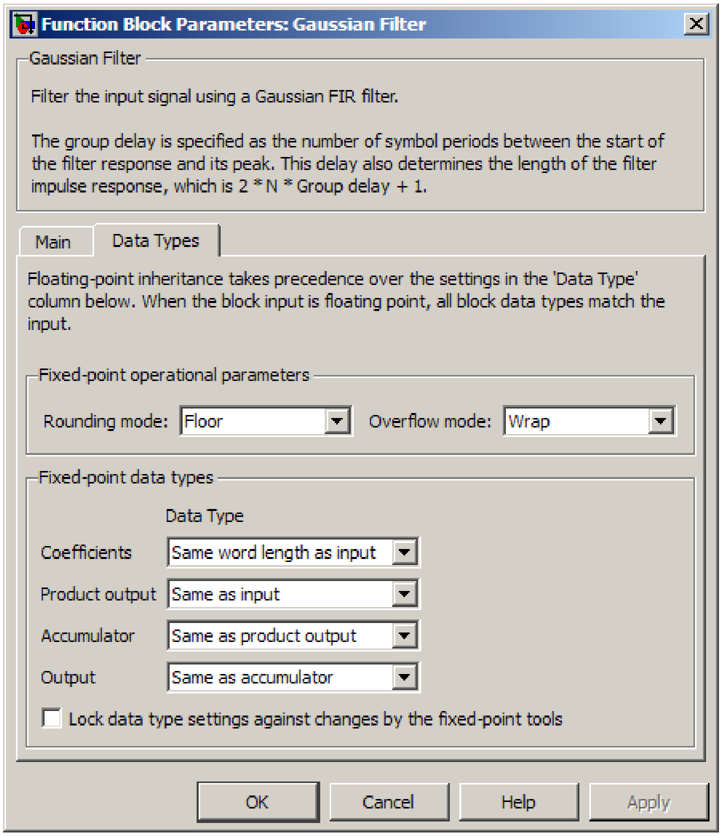
If you check this box, then the block creates a variable in the MATLAB workspace that contains the filter coefficients.

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.



Function Block Parameters: Gaussian Filter

Gaussian Filter

Filter the input signal using a Gaussian FIR filter.

The group delay is specified as the number of symbol periods between the start of the filter response and its peak. This delay also determines the length of the filter impulse response, which is 2 * N * Group delay + 1.

Main | Data Types

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

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Fixed-point data types

	Data Type
Coefficients	Same word length as input
Product output	Same as input
Accumulator	Same as product output
Output	Same as accumulator

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

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 (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

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” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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, `gaussdesign`

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)

Introduced before R2006a

Gaussian Noise Generator

Generate Gaussian distributed noise with given mean and variance values

Note

Note Gaussian Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

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.

Parameters

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.

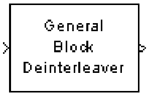
See Also

Random Source, AWGN Channel, `rand`, `randseed`, MATLAB Function

Introduced before R2006a

General Block Deinterleaver

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 **Permutation vector** 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*,

Output(**Permutation vector**(*k*)) = Input(*k*)

The **Permutation vector** parameter must contain unique integers between 1 and *N*.

Both the input and the **Permutation vector** parameter must be column vector signals.

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” (Simulink).

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 **Permutation vector** 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.

Parameters

Permutation vector source

A selection that specifies the source of the permutation vector. The source can be either `Dialog` or `Input port`. The default value is `Dialog`.

Permutation vector

A vector of length N that lists the indices of the output elements that came from the input vector. This parameter is available only when **Permutation vector source** is set to `Dialog`.

Examples

This example reverses the operation in the example on the General Block Interleaver block reference page. If you set **Permutation vector** 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

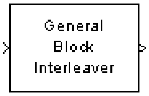
See Also

`perms` (MATLAB function)

Introduced before R2006a

General Block Interleaver

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 **Permutation vector** 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(**Permutation vector**(k))

for each integer *k* between 1 and *N*. The contents of **Permutation vector** must be integers between 1 and N, and must have no repetitions.

Both the input and the **Permutation vector** parameter must be column vector signals.

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” (Simulink).

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.

Parameters

Permutation vector source

A selection that specifies the source of the permutation vector. The source can be either `Dialog` or `Input port`. The default value is `Dialog`.

Permutation vector

A vector of length N that lists the indices of the output elements that came from the input vector. This parameter is available only when **Permutation vector source** is set to `Dialog`.

Examples

If **Permutation vector** 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 **Permutation vector** is a permutation of the vector `[1:4]'`.

Pair Block

General Block Deinterleaver

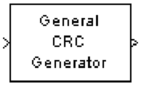
See Also

`perms` (MATLAB function)

Introduced before R2006a

General CRC Generator

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 polynomial character vector such as `'x^3 + x^2 + 1'`.
- 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:

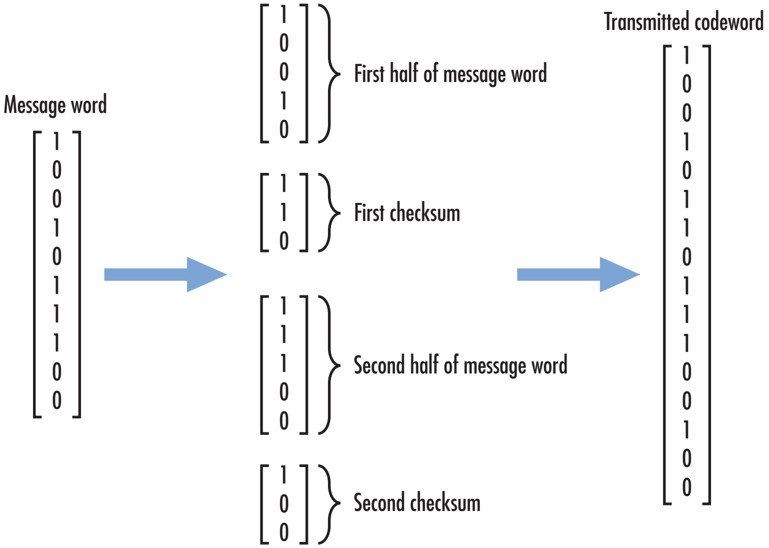
1. Divides each input frame into *k* subframes of equal size
2. Prefixes the **Initial states** vector to each of the *k* subframes
3. Applies the CRC algorithm to each augmented subframe
4. Appends the resulting checksums at the end of each subframe
5. Outputs concatenated subframes

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.

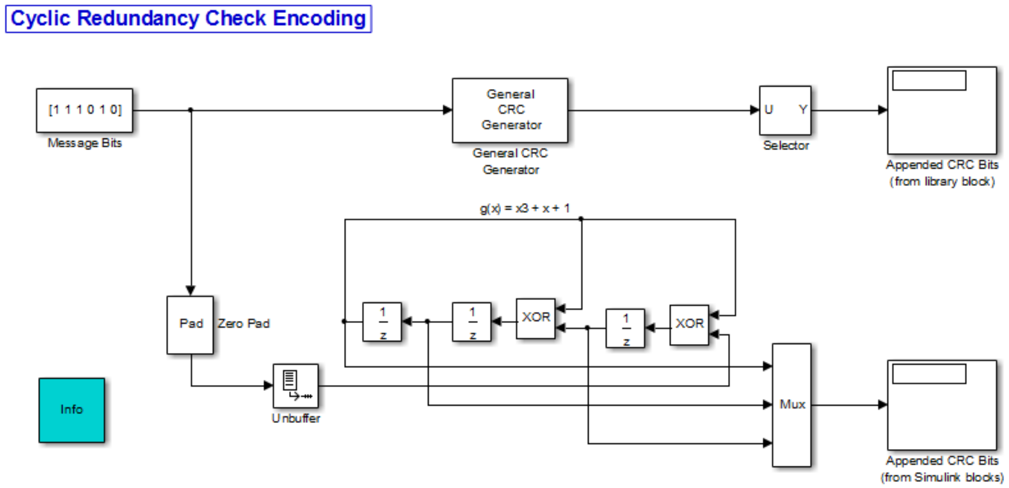
Example

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.

Parameters

Generator polynomial

A polynomial character vector, a binary row vector in descending order of powers, or an integer row vector 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

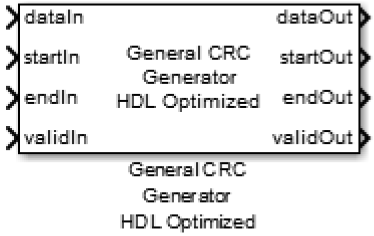
See Also

CRC-N Generator, CRC-N Syndrome Detector

Introduced before R2006a

General CRC Generator HDL Optimized

Generate CRC code bits and append to input data, optimized for HDL code generation


dataIn
startIn
endIn
validIn
General CRC Generator HDL Optimized
dataOut
startOut
endOut
validOut
General CRC Generator HDL Optimized

Library

Communications System Toolbox > Error Correction and Detection > CRC (`commcrc2`)

Communications System Toolbox HDL Support > Error Correction and Detection > CRC (`commhdlcrc`)

Description

This hardware-friendly CRC generator block, like the General CRC Generator block, generates a cyclic redundancy check (CRC) checksum and appends it to the input 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 accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame. To achieve higher throughput, the block accepts vector data up to the CRC length, and implements a parallel architecture.

Signal Attributes

Port	Direction	Description	Data Type
dataIn	Input	Message data. Data can be a vector of binary values, or a scalar integer representing several bits. For example, vector input [0 0 0 1 0 0 1 1] is equivalent to uint8 input 19. The data width must be less than or equal to the CRC length, and the CRC length must be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.	Vector: double or Boolean Scalar: unsigned integer (uint8/16/32) or fixdt(0,N, 0)
startIn	Input	Indicates the start of a frame of data.	Boolean
endIn	Input	Indicates the end of a frame of data.	Boolean
validIn	Input	Indicates that input data is valid.	Boolean
dataOut	Output	Message data with the checksum appended. The output data has the same vector size as the input data.	Same as dataIn
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

Parameters

Polynomial

A double or Boolean 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 or Boolean scalar or vector of length equal to the CRC length, specifying the initial state of the internal shift register. The default value is 0.

Direct method

- When this parameter is selected, the block uses the direct algorithm for CRC checksum calculations.
- When this parameter is not selected, the block uses the nondirect algorithm for CRC checksum calculations.

The parameter is cleared by default.

To learn about the direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

Reflect input

The input data width must be a multiple of 8.

- When this parameter is selected, each input byte is flipped before entering the shift register.
- When this parameter is not selected, the message data is passed to the shift register unchanged.

The parameter is cleared by default.

Reflect CRC checksum

The CRC length must be a multiple of 8.

- When this parameter is selected, each checksum byte is flipped before it is passed to the final XOR stage.
- When this parameter is not selected, the checksum byte is passed to the final XOR stage unchanged.

The parameter is cleared by default.

Final XOR value

The value that the CRC checksum is XORed with before it is appended to the input data. This parameter can be a double or Boolean scalar or vector of length equal to the CRC length. The default value is 0.

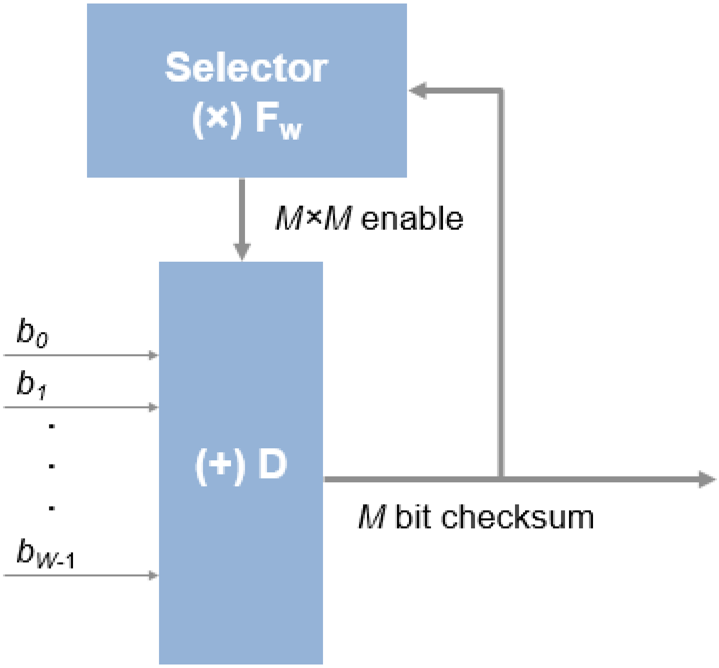
Algorithm

When you use vector or integer input, the block implements a parallel CRC algorithm [1].

To provide high throughput for modern communications systems, the CRC algorithm is implemented with a parallel architecture. This architecture recursively calculates *M* bits of CRC checksum for each *W* input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of *M*, the recursive checksum calculation for *W* bits in parallel is:

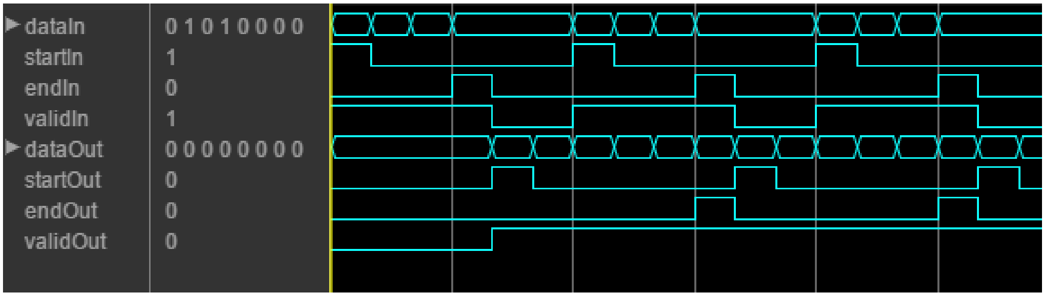
$$X^{'} = F_W (\times) X (+) D$$

F_W is an *M*-by-*M* matrix that selects elements of the current state for the polynomial calculation with the new input bits. *D* is an *M*-sample vector that provides the new input bits, ordered in relation to the polynomial and padded with zeroes. (×) is implemented with logical AND, and (+) is implemented with logical XOR.



Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. There must be enough space between the input frames to insert the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported. The output valid signal matches the input valid pattern.

Initial Delay

The General CRC Generator HDL Optimized block introduces a latency on the output. This latency can be computed with the following equation, assuming the input data is continuous:

```
initialdelay = (CRC length/input data width) + 2
```

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see General CRC Generator HDL Optimized in the HDL Coder documentation.

References

[1] Campobello, Giuseppe, Giuseppe Patane, and Marco Russo. "Parallel CRC Realization." *IEEE Transactions on Computers*. Vol. 52, No. 10, October 2003, pp. 1312–1319.

See Also

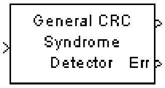
Blocks
General CRC Generator | General CRC Syndrome Detector HDL Optimized

System Objects
`comm.HDLCRCGenerator`

Introduced in R2012a

General CRC Syndrome Detector

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.

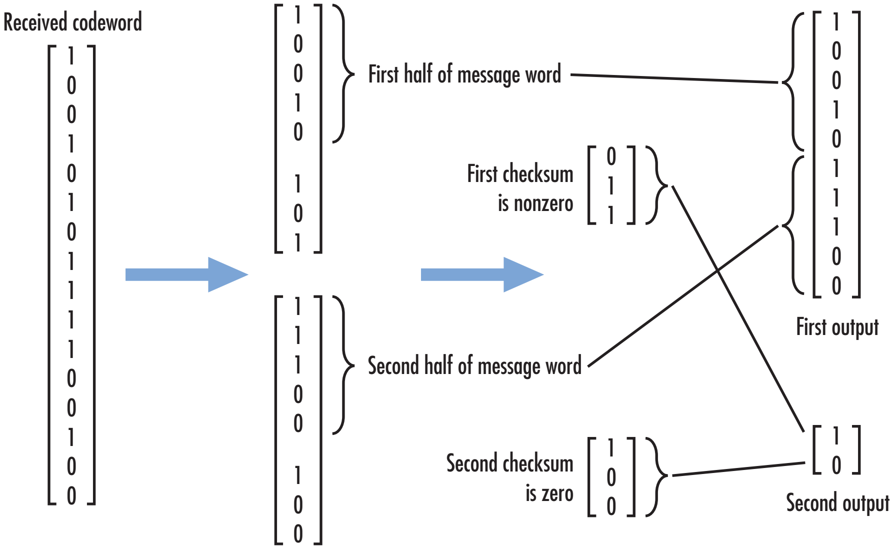
The block's parameter settings should agree with those in the General CRC Generator block.

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.

Parameters

Generator polynomial

A character vector, a binary row vector in descending order of powers, or an integer row vector 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

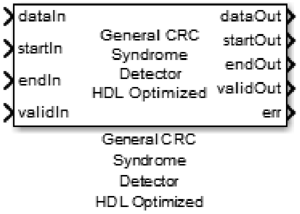
See Also

CRC-N Generator, CRC-N Syndrome Detector

Introduced before R2006a

General CRC Syndrome Detector HDL Optimized

Detect errors in input data using CRC, optimized for HDL code generation


dataIn
startIn
endIn
validIn
General CRC Syndrome Detector HDL Optimized
dataOut
startOut
endOut
validOut
err
General CRC Syndrome Detector HDL Optimized

Library

Communications System Toolbox > Error Correction and Detection > CRC (`commcrc2`)

Communications System Toolbox HDL Support > Error Correction and Detection > CRC (`commhdlcrc`)

Description

This hardware-friendly CRC detector block performs a cyclic redundancy check (CRC) on data and compares the resulting checksum with the appended checksum. If the two checksums do not match, the block reports an error. Instead of processing an entire frame at once, the block accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame. To achieve higher throughput, the block accepts vector data up to the CRC length, and implements a parallel architecture.

Signal Attributes

Port	Direction	Description	Data Type
`dataIn`	Input	Message data plus checksum. Data can be a vector of binary values, or a scalar integer representing several bits. For example, vector input [0 0 0 1 0 0 1 1] is equivalent to uint8 input 19. The data width must be less than or equal to the CRC length, and the CRC length must be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.	Vector: double or Boolean Scalar: unsigned integer (uint8/16/32) or fixdt(0,N, 0)
`startIn`	Input	Indicates the start of a frame of data, including checksum.	Boolean
`endIn`	Input	Indicates the end of a frame of data.	Boolean
`validIn`	Input	Indicates that input data is valid.	Boolean
`dataOut`	Output	Message data. The output data has the same width as the input data.	Same as `dataIn`
`startOut`	Output	Indicates the start of a frame of data.	Boolean
`endOut`	Output	Indicates the end of a frame of data.	Boolean
`validOut`	Output	Indicates that output data is valid.	Boolean
`err`	Output	Indicates the corruption of the received data, when `err` is high(1).	Boolean

Parameters

Polynomial

A double or Boolean vector specifying the polynomial, in descending order of powers. The 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 or Boolean scalar or vector of length equal to the CRC length, specifying the initial state of the internal shift register. The default value is 0.

Direct method

- When this parameter is selected, the block uses the direct algorithm for CRC checksum calculations.
- When this parameter is not selected, the block uses the nondirect algorithm for CRC checksum calculations.

The parameter is cleared by default.

To learn about the direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

Reflect input

The input data width must be a multiple of 8.

- When this parameter is selected, each input byte is flipped before entering the shift register.
- When this parameter is not selected, the message data is passed to the shift register unchanged.

The parameter is cleared by default.

Reflect CRC checksum

The CRC length must be a multiple of 8.

- When this parameter is selected, each checksum byte is flipped before it is passed to the final XOR stage.
- When this parameter is not selected, the checksum byte is passed to the final XOR stage unchanged.

The parameter is cleared by default.

Final XOR value

The value that the CRC checksum is XORed with before it is appended to the input data. This parameter can be a double or Boolean scalar or vector of length equal to the CRC length. The default value is 0.

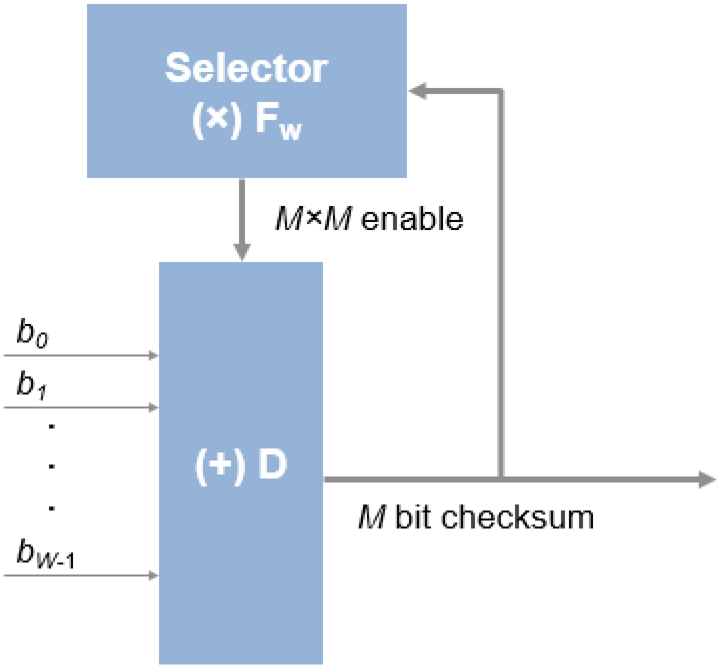
Algorithm

When you use vector or integer input, the block implements a parallel CRC algorithm [1].

To provide high throughput for modern communications systems, the CRC algorithm is implemented with a parallel architecture. This architecture recursively calculates *M* bits of CRC checksum for each *W* input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of *M*, the recursive checksum calculation for *W* bits in parallel is:

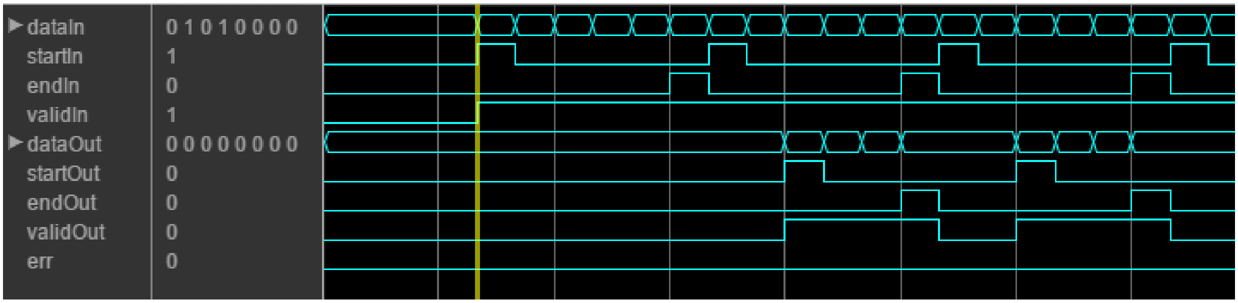
$$X^{'} = F_W (\times) X (+) D$$

F_W is an *M*-by-*M* matrix that selects elements of the current state for the polynomial calculation with the new input bits. *D* is an *M*-sample vector that provides the new input bits, ordered in relation to the polynomial and padded with zeroes. (×) is implemented with logical AND, and (+) is implemented with logical XOR.



Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. The input frames are contiguous, and the output frames show space between them because the detector block removes the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported.

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, assuming the input data is continuous:

```
initialdelay = 3 * (CRC length/input data width) + 2
```

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see General CRC Syndrome Detector HDL Optimized in the HDL Coder documentation.

References

[1] Campobello, Giuseppe, Giuseppe Patane, and Marco Russo. "Parallel CRC Realization." *IEEE Transactions on Computers*. Vol. 52, No. 10, October 2003, pp. 1312–1319.

See Also

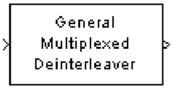
Blocks
General CRC Generator HDL Optimized | General CRC Syndrome Detector

System Objects
`comm.HDLCRCDetector`

Introduced in R2012b

General Multiplexed Deinterleaver

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 data type of the output will be the same as that of the input signal.

Parameters

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.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see General Multiplexed Deinterleaver in the HDL Coder documentation.

Pair Block

General Multiplexed Interleaver

See Also

Convolutional Deinterleaver, Helical Deinterleaver

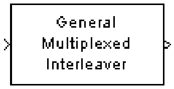
References

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

Introduced before R2006a

General Multiplexed Interleaver

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.

Parameters

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.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see General Multiplexed Interleaver in the HDL Coder documentation.

Pair Block

General Multiplexed Deinterleaver

See Also

Convolutional Interleaver, Helical Interleaver

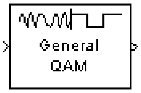
References

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

Introduced before R2006a

General QAM Demodulator Baseband

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 2-407 table on this page.

Parameters

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.

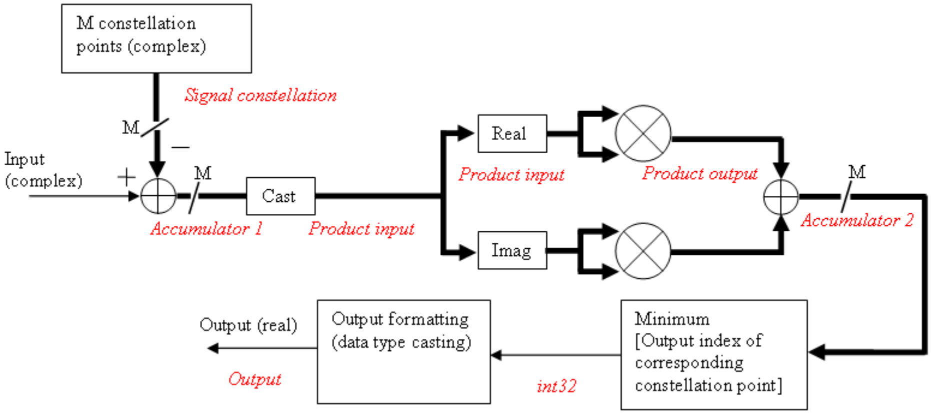
If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of 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

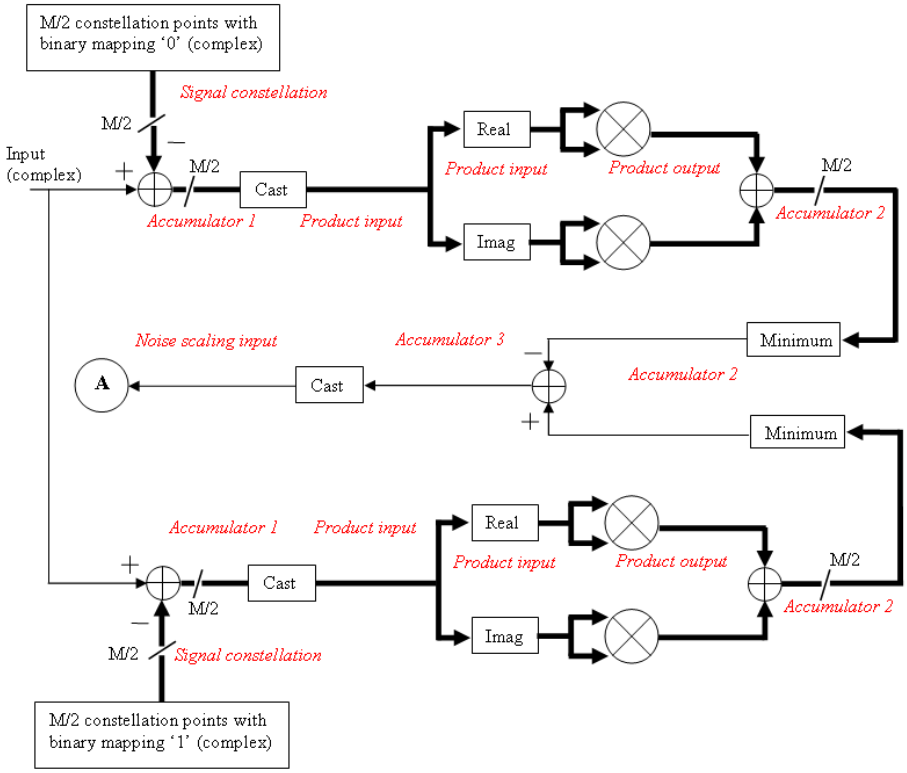


Fixed-Point Signal Flow Diagram for Hard Decision Mode

Note In the figure above, *M* represents the size of the **Signal constellation** .

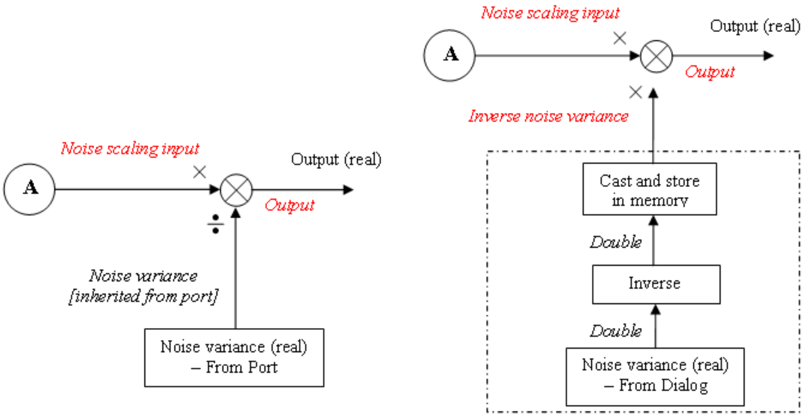
The general QAM Demodulator Baseband block supports fixed-point operations for computing Hard Decision (**Output** type set to `Bit` and **Decision type** is set to `Hard decision`) and Approximate LLR (**Output type** is set to `Bit` and **Decision type** is set to `Approximate Log-Likelihood ratio`) 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**.



(a) Noise variance from Port

(b) Noise variance from Dialog

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

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 $ufix\left(\lceil \log_2 M \rceil\right)$ 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” (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

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 (Fixed-Point Designer) for more information.

For more information, see the **Saturate on integer overflow** parameter subsection of “Specify Fixed-Point Attributes for Blocks” (DSP System Toolbox).

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" (DSP System Toolbox) 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` (DSP System Toolbox) 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 (DSP System Toolbox) 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.

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 (DSP System Toolbox) 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 2-402. In this case, it follows the internal rule for Product data types specified in the Inherit via Internal Rule (DSP System Toolbox) 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" (DSP System Toolbox) 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" (DSP System Toolbox).

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` • $ufix\left(\lceil \log_2 M \rceil\right)$ 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

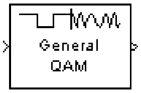
See Also

Rectangular QAM Demodulator Baseband

Introduced before R2006a

General QAM Modulator Baseband

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 2-410 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*.

Parameters

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 • $ufix\left(\left\lceil \log_2 M \right\rceil\right)$
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

General QAM Demodulator Baseband

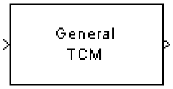
See Also

Rectangular QAM Modulator Baseband

Introduced before R2006a

General TCM Decoder

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 2-413.

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**.

Parameters

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

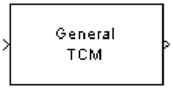
[1] Biglieri, E., D. Divsalar, P. J. McLane, and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

Introduced before R2006a

General TCM Encoder

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 2-418.

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”. 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

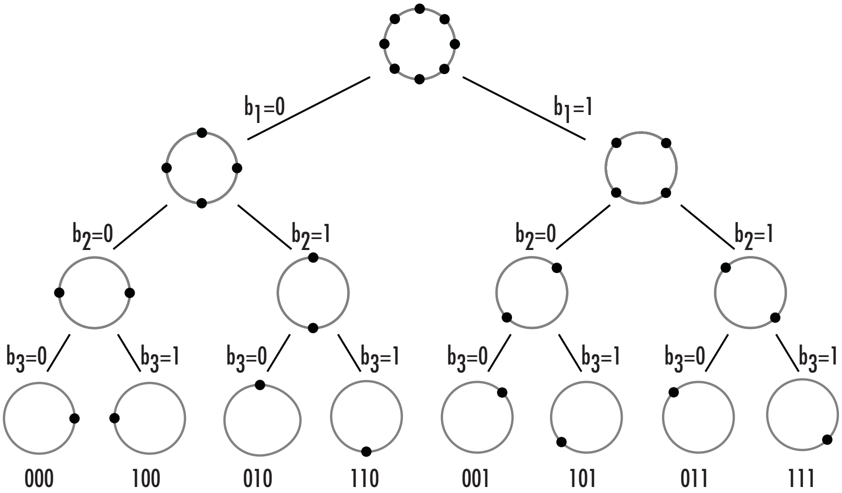
The trellis-coded modulation technique partitions the constellation into subsets called cosets so as to maximize the minimum distance between pairs of points in each coset.

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].

Parameters

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.

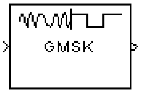
[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Introduced before R2006a

GMSK Demodulator Baseband

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

Parameters

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.

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

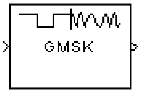
References

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

Introduced before R2006a

GMSK Modulator Baseband

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 on page 2-428 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.

Parameters

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.

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

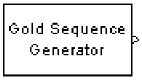
References

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

Introduced before R2006a

Gold Sequence Generator

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” (Simulink).

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 2-432 below. The set *G*(*u, v*) of Gold sequences is defined by

$$G(u,v) = \{u,v,u \oplus v,u \oplus Tv,u \oplus T^2v,\ldots,u \oplus T^{N-1}v\}$$

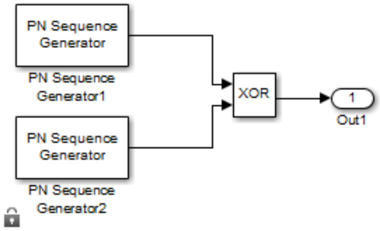
where *T* represents the operator that shifts vectors cyclically to the left by one place, and

$\oplus$ 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 these formats:

- A polynomial character vector that includes the number `1`, for example, `'z^4 + z + 1'`.
- 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 character vector `'z^5 + z^2 + 1'`, the vector `[5 2 0]`, and the vector `[1 0 0 1 0 1]` 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]`
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
0	$u \oplus v$
1	$u \oplus Tv$

Sequence Index	Output Sequence
2	$u \oplus T^2 v$
...	...
2ⁿ-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 2-764 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
- $$\gcd(n,k) = \begin{cases} 1 & n \equiv 1 \bmod 2 \\ 2 & n \equiv 2 \bmod 4 \end{cases}$$

Parameters

Preferred polynomial[1]

Character vector or 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]

Character vector or 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

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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 (Simulink) 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.

Introduced before R2006a

Hadamard Code Generator

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

$H_N H_N^T = N I_N$

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.

Parameters

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

The time between each sample of the output signal. Specify as a nonnegative real scalar.

Samples per frame

The number of samples in one column of the output signal. Specify as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

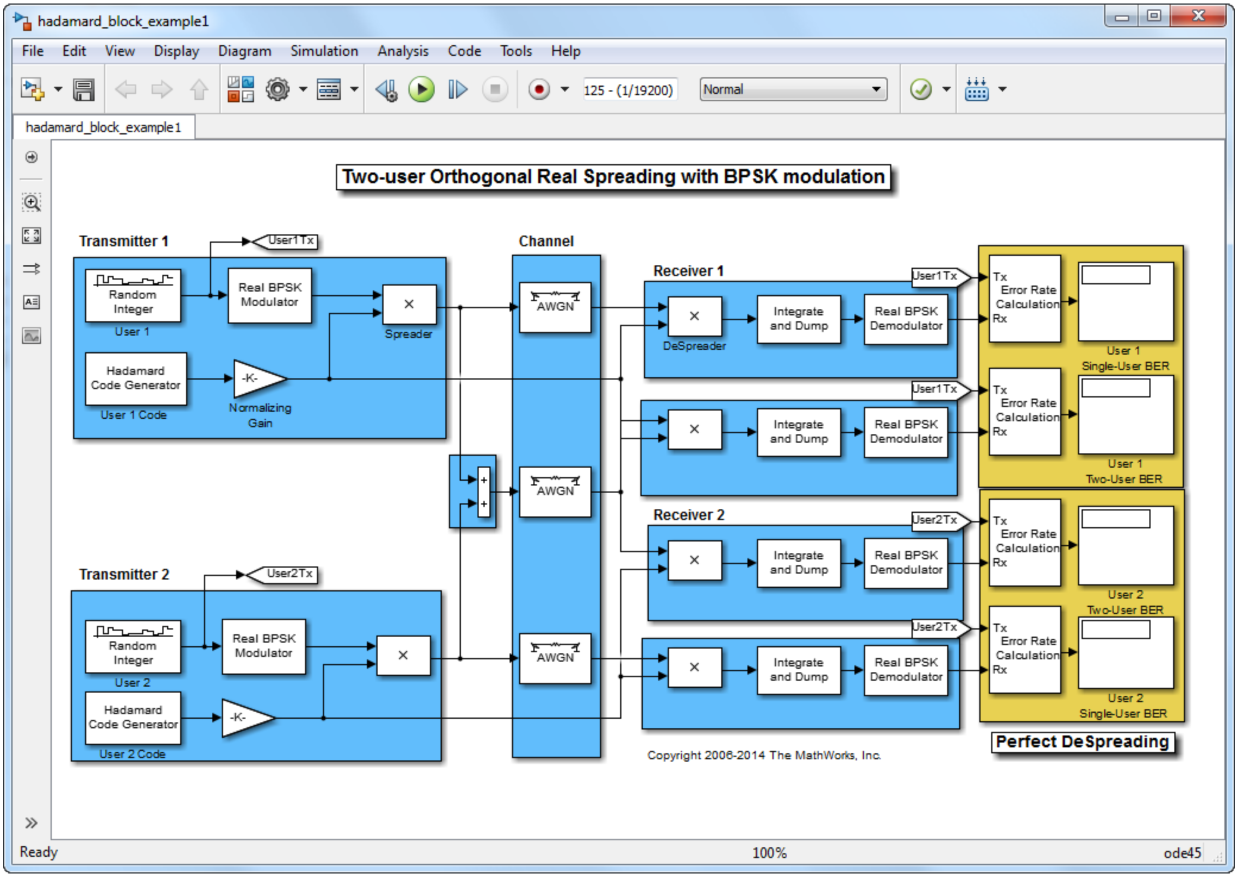
Examples

Orthogonal Spreading - Single-User vs. Two-User Comparison

This example model compares a single-user system vs. a two-user data transmission system with the two data streams being independently spread by different orthogonal codes.

The model uses random binary data which is BPSK modulated (real), spread by Hadamard codes of length 64 and then transmitted over an AWGN channel. The receiver consists of a despreader followed by a BPSK demodulator. Open the model here: hadamard_block_example1.

```
modelname = 'hadamard_block_example1';
open_system(modelname);
sim(modelname);
```



For the same data and channel settings, the model calculates the performance for one- and two-user transmissions.

Note that for the individual users, the error rates are exactly the same in both cases. This shows that perfect despreading is possible due to the ideal cross-correlation properties of the Hadamard codes.

To experiment with this model further, specify a different **Code length** or **Code index** for the individual users to examine the variations in relative performance.

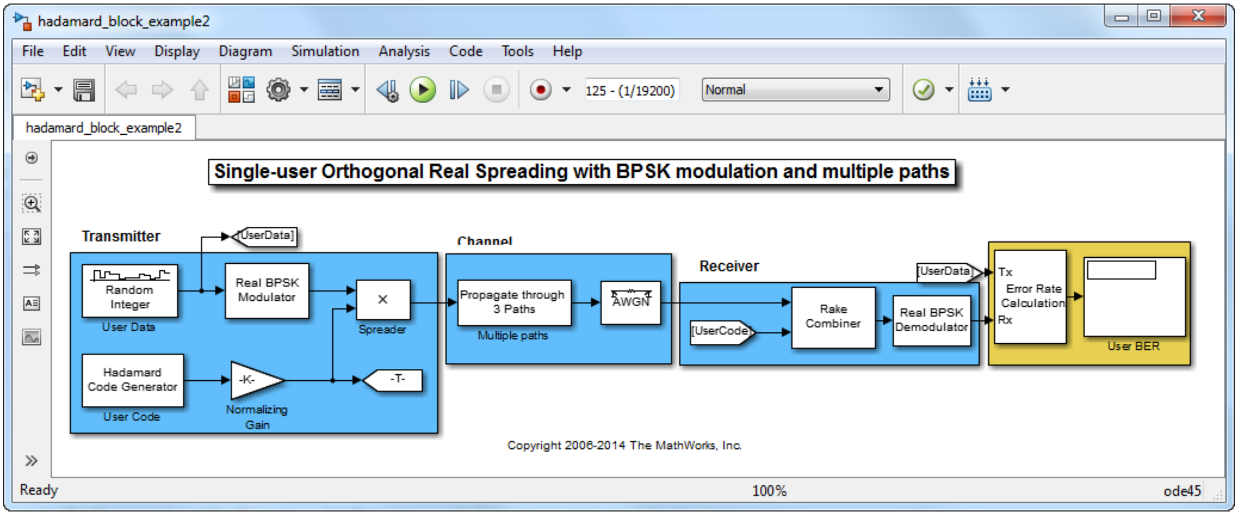
```
close_system(modelname, 0);
```

Orthogonal Spreading - Multipath Scenario

This example model considers a single-user system in which the signal is transmitted over multiple paths. This is similar to a mobile channel environment where the signals are received over multiple paths, each of which have different amplitudes and delays. To take advantage of the multipath transmission, the receiver employs diversity reception which combines the independent paths coherently.

Note, to keep the system simple, no shadowing effects are considered and the receiver has *a priori* knowledge of the number of paths and their respective delays. Open the model here: hadamard_block_example2.

```
modelname = 'hadamard_block_example2';
open_system(modelname);
sim(modelname);
```



For the data transmission with the same spreading code that was used in the first example, we now see deterioration in performance when compared with that example (compare the 180 errors with 81 in the previous case). This can be attributed to the non-ideal auto-correlation values of the orthogonal spreading codes chosen, which prevents perfect resolution of the individual paths. Consequently, we don't see the merits of diversity combining.

To experiment with this model further, try selecting other path delays to see how the performance varies for the same code. Also try different codes with the same delays.

```
close_system(modelname, 0);
```

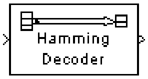
See Also

OVSF Code Generator, Walsh Code Generator

Introduced before R2006a

Hamming Decoder

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.
- In addition, you can specify the primitive polynomial as a character vector, for example, `'D^3 + D + 1'`.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-443 table on this page.

Parameters

Codeword length N

The codeword length *N*, which is also the input vector length.

Message length K, or M-degree primitive polynomial

The message length, which is also the input vector length or a binary vector that represents a primitive polynomial for GF(2^M) or a polynomial character vector.

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

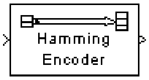
See Also

`hammgen` (Communications System Toolbox)

Introduced before R2006a

Hamming Encoder

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.
- In addition, you can specify the primitive polynomial as a character vector, for example, `'D^3 + D + 1'`.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-446 table on this page.

Parameters

Codeword length N

The codeword length, which is also the output vector length.

Message length K, or M-degree primitive polynomial

The message length, which is also the input vector length or a binary vector that represents a primitive polynomial for GF(2^M) or a polynomial character vector.

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 Decoder

See Also

`hammgen` (Communications System Toolbox)

Introduced before R2006a

Helical Deinterleaver

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*·*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*·*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 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.

Parameters

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.

Introduced before R2006a

Helical Interleaver

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 $1+(k\text{-}1){\cdot}\ s$, 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.

Parameters

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

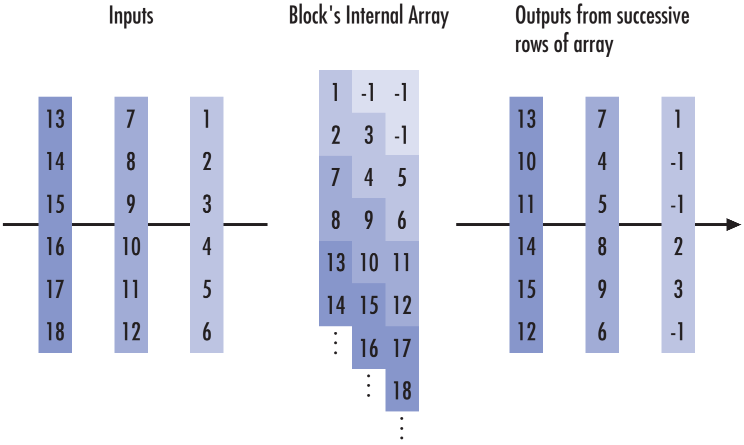
The number of rows of separation between consecutive input groups in their respective columns of the helical array.

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.

Introduced before R2006a

Ideal Rectangular Pulse Filter

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 2-459 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.

Parameters

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.

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 (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

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” (DSP System Toolbox) 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 **Saturate on integer overflow** 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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

Introduced before R2006a

Insert Zero

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.

Parameters

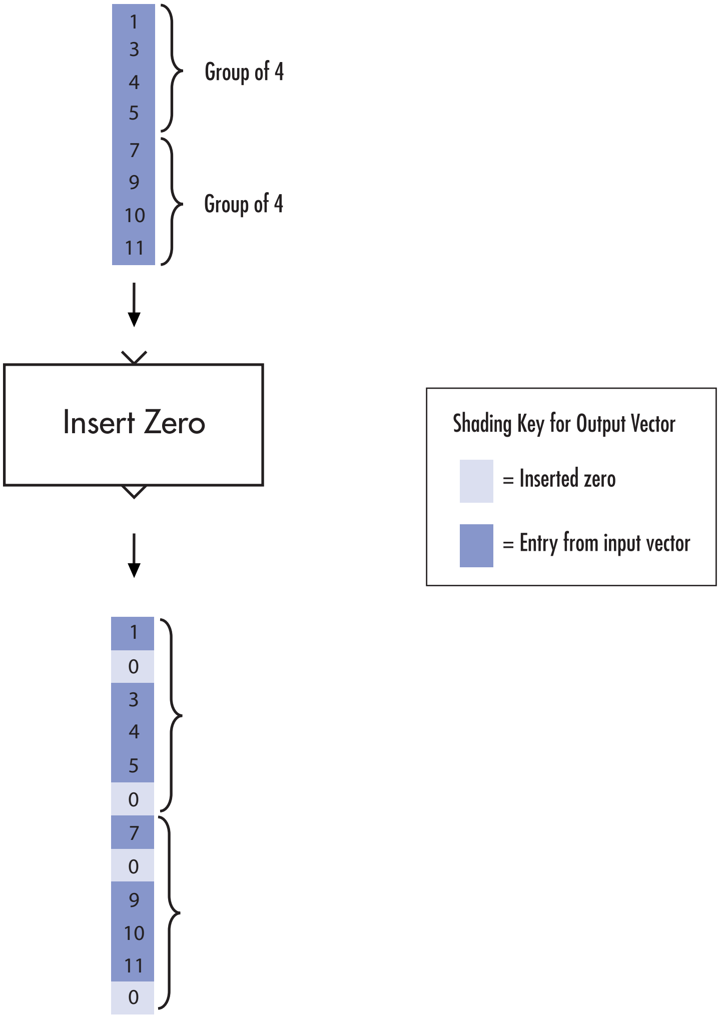
Insert zero vector

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.



Compare this example with that on the reference page for the Puncture block.

See Also

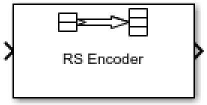
Puncture

Introduced before R2006a

Integer-Input RS Encoder

Create Reed-Solomon code from integer vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding



Description

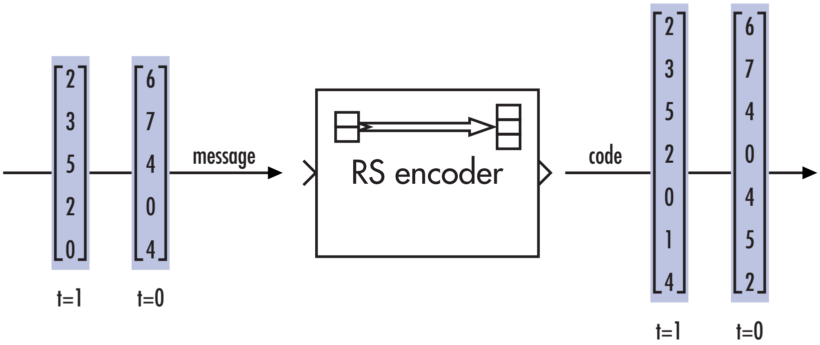
The Integer-Input RS Encoder block creates a Reed-Solomon code.

The symbols for the code are integers between 0 and 2^M-1, which represent elements of the finite field $GF(2^M)$. 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 default value of M by specifying the primitive polynomial for $GF(2^M)$, as described in “Specify the Primitive Polynomial” on page 2-471 below. Restrictions on M and N are described in “Restrictions on M and the Codeword Length N” on page 2-470.

The input and output are integer-valued signals that represent messages and codewords, respectively. For more information, see“Input and Output Signal Length in RS Blocks” on page 2-469.

An (N, K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (not bit errors) in each codeword.

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.



Ports

Input

In — Message

integer column vector

Message, specified as one of the following:

- When there is no message shortening, a (N_C×K)-by-1 integer column vector.
- When there is message shortening, a (N_C×S)-by-1 integer column vector.

N_C is the number of message words, K is the **Message length K**, and S is the **Shortened message length S**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-469.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

Out — Reed-Solomon codeword

integer column vector

Reed-Solomon codeword, returned as an (N_C×(N – K + S – P)-by-1 integer column vector. N_C is the number of codewords, N is the **Codeword length N**, K is the **Message length K**, S is the **Shortened message length S**, P is the number of punctures per codeword.

For more information, see “Input and Output Signal Length in RS Blocks” on page 2-469.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

For more information, see “Supported Data Types” on page 2-472.

Parameters

`Codeword length N` — Codeword length

7 (default) | integer

Codeword length, specified as an integer.

For more information, see“Restrictions on M and the Codeword Length N” on page 2-470 and “Input and Output Signal Length in RS Blocks” on page 2-469.

`Message length K` — Message word length

3 (default) | integer

Message word length, specified as an integer from 1 to N–2, where N is the codeword length.

`Shortened message length S` — Shortened message word length

3 (default) | integer

Shortened message word length, specified as an integer, such that $S \leq K$. When **Shortened message length S < Message length K**, the Reed-Solomon code is shortened.

You still specify N and K values for the full-length (N, K) code but the decoding is shortened to an (N–K+S, S) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values from 0 to 2^M–1, in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see “Specify the Generator Polynomial” on page 2-471.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Specify the Primitive Polynomial” on page 2-471.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (N–K)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 2-472.

Note If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

Dependencies

To enable this parameter, select **Puncture code**.

Definitions

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, K, or shortened message word length, S. The codeword length is $N – K + S – P$, where N is the full codeword length and P is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N – P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that y is an integer multiple of x.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Integer-Input RS Encoder	**Input Length (symbols):**
	$N_C \times K$
	Output Length (symbols):
	$N_C \times (N–P)$
		Input Length (symbols):
		$N_C \times S$
		Output Length (symbols):
		$N_C \times (N–K+S–P)$

Integer-Output RS Decoder	**Input Length (symbols):**
	$N_C \times (N–P)$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (symbols):
	$N_C \times K$
		Input Length (symbols):
		$N_C \times (N–K+S–P)$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (symbols):
		$N_C \times S$

- N is the codeword length.
- K is the message word length.
- S is the shortened message word length.
- N_C is the number of codewords (and message words).
- P is the number of punctures, and is equal to the number of zeros in the puncture vector.
- M is the degree of the primitive polynomial. Each group of M bits represents an integer between `0` and `2`^{M}`—1` that belongs to the finite Galois field `GF(2`^{M}`)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Restrictions on M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, N, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.

- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, *M*, are from 3 to 16. The valid values for *N* in this case are from 7 to 2^M–1. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify 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** text box, enter a binary row vector that represents a primitive polynomial over $GF(2^M)$, 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.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 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. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})\ldots(x+\alpha^{b+N-K-1})$$

α 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 running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

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

Port	Supported Data Types
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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

See Also

Blocks
Binary-Input RS Encoder

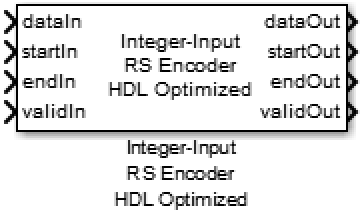
System Objects
`comm.RSEncoder`

Functions

Introduced before R2006a

Integer-Input RS Encoder HDL Optimized

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, one symbol at a time. The wordlength of each symbol must be ceil($\log_2$(*codewordLength*+1)).	Integer or fixdt with any binary point scaling. double is allowed for simulation but not for HDL code generation.
startIn	Input	Indicates the start of a frame of data.	Boolean
endIn	Input	Indicates the end of a frame of data.	Boolean
validIn	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.	Same as dataIn
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

Parameters

Codeword length

The length of the code word, N, must be equal to 2^M-1, where M is the input word length. M can be between 3 and 16 bits.

Message length

The message length, K. The number of parity symbols, N-K, must be a positive even integer, greater than or equal to the input word length, M.

Each input frame, i.e. the number of valid data samples between start and end, must contain more than N-K symbols, and fewer than or equal to K symbols. A shortened code is inferred anytime the number of input data samples in a frame is less than K.

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, `M`, must be an integer multiple of `K` times the order of the primitive polynomial.

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. When you select `Auto`, the block uses `B = 1`.

B value

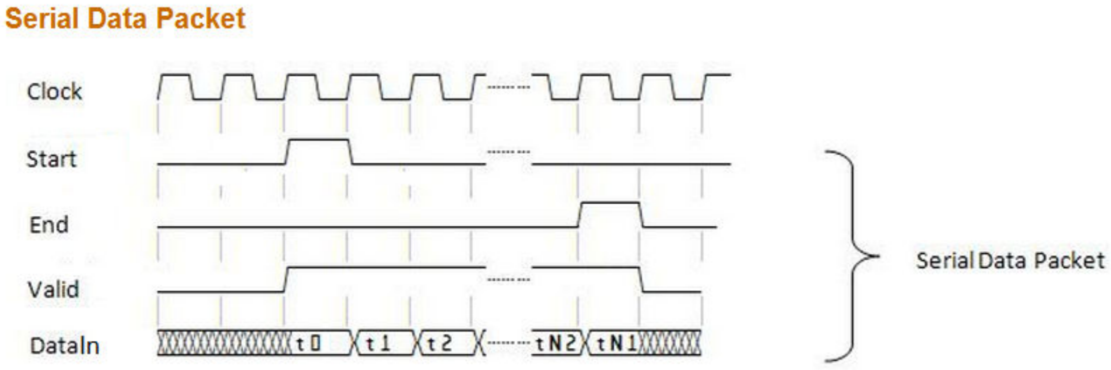
The starting exponent of the roots.

This field is available only when you select `Property` for **Source of B, the starting power for roots of the primitive polynomial**.

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. To get the value of B from a generator polynomial, use the `genpoly2b` function. The default value is 1.

Algorithm

Timing Diagram



HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Integer-Input RS Encoder HDL Optimized in the HDL Coder documentation.

Examples

“Using HDL Optimized RS Encoder/Decoder Library Blocks”

Pair Block

Integer-Output RS Decoder HDL Optimized

See Also

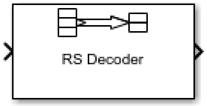
Integer-Input RS Encoder | `comm.HDLRSEncoder`

Introduced in R2012b

Integer-Output RS Decoder

Decode Reed-Solomon code to recover integer vector data

Library: Communications System Toolbox / Error Detection and Correction / Block Coding

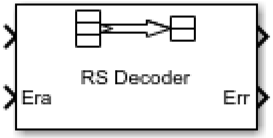


Description

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 must 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 *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 the M and the Codeword Length N” on page 2-483 below.

The block can output shortened codewords when the **Shortened message length S** is specified. In this case, the codeword length *N* and message length *K* should specify the full-length (*N*, *K*) code that is shortened to an (*N*–*K*+*S*, *S*) code.



This icon shows optional ports.

The input and output are integer-valued signals that represent codewords and messages, respectively. For more information, see “Input and Output Signal Length in RS Blocks” on page 2-482. The block inherits the output data type from the input data type. For information about the data types each block port supports, see “Supported Data Types” on page 2-485.

For more information on representing data for Reed-Solomon codes, see the section “Integer Format (Reed-Solomon Only)”.

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 “Specify the Primitive Polynomial” on page 2-484 below.

You can also specify the generator polynomial for the Reed-Solomon code, as described in “Specify the Generator Polynomial” on page 2-484.

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.

If decoding fails, 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.

Parameters

Codeword length N

The codeword length in symbols.

Message length K

The message length in symbols.

Specify shortened message length

Selecting this check box enables the **Shortened message length S** text box.

Shortened message length S

The shortened message length in symbols. When you specify this parameter, provide full-length *N* and *K* values to specify the (*N*, *K*) code that is shortened to an (*N*–*K*+*S*, *S*) code.

Specify generator polynomial

Selecting this check box enables the **Generator polynomial** text box.

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**.

Specify primitive polynomial

Selecting this check box enables the **Primitive polynomial** text box.

Primitive polynomial

This parameter applies only when you select **Specify primitive polynomial**.

Binary row vector representing the primitive polynomial in descending order of powers.

Puncture code

Selecting this check box enables the **Puncture vector** text box.

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 symbol 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.

Definitions

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, K, or shortened message word length, S. The codeword length is $N – K + S – P$, where N is the full codeword length and P is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N – P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that y is an integer multiple of x.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Integer-Input RS Encoder	**Input Length (symbols):**
	$N_C \times K$
	Output Length (symbols):
	$N_C \times (N–P)$
		Input Length (symbols):
		$N_C \times S$
		Output Length (symbols):
		$N_C \times (N–K+S–P)$

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Integer-Output RS Decoder	**Input Length (symbols):**
	$N_C \times (N–P)$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (symbols):
	$N_C \times K$
		Input Length (symbols):
		$N_C \times (N–K+S–P)$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (symbols):
		$N_C \times S$

- N is the codeword length.
- K is the message word length.
- S is the shortened message word length.
- N_C is the number of codewords (and message words).
- P is the number of punctures, and is equal to the number of zeros in the puncture vector.
- M is the degree of the primitive polynomial. Each group of M bits represents an integer between `0` and `2`^{M}`–1` that belongs to the finite Galois field `GF(2`^{M}`)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Restrictions on the M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, N, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, M, are from 3 to 16. The valid values for N in this case are from 7 to 2^M–1. Selecting **Specify primitive polynomial** enables you to specify the primitive

polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify 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** text box, enter a binary row vector that represents a primitive polynomial over $GF(2^M)$, 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.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 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. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$g(x) = (x+\alpha^b)(x+\alpha^{b+1})(x+\alpha^{b+2})\ldots(x+\alpha^{b+N-K-1})$

α 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 running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2^M-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

1s and 0s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- 1 means that the data symbol is passed through the block unaltered.
- 0 means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- 1 means that the data symbol is to be replaced with an erasure symbol.
- 0 means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

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
Era	• Double-precision floating point • Boolean

Port	Supported Data Types
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

Algorithms

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see "Algorithms for BCH and RS Errors-only Decoding".

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

Blocks
Binary-Output RS Decoder

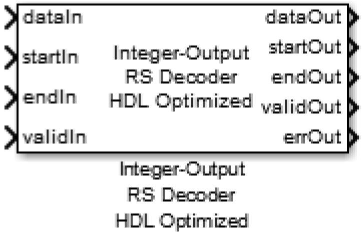
System Objects
`comm.RSDecoder`

Functions
`primpoly` | `rsdec` | `rsgenpoly`

Introduced before R2006a

Integer-Output RS Decoder HDL Optimized

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).

Port	Direction	Description	Data Type
dataIn	Input	Message data, one symbol at a time.	Integer or fixdt with any binary point scaling. double is allowed for simulation but not for HDL code generation.
startIn	Input	Indicates the start of a frame of data.	Boolean
endIn	Input	Indicates the end of a frame of data.	Boolean
validIn	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.	Same as dataIn
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
errOut	Output	Indicates the corruption of the received data when error is high.	Boolean
numErrors	Output (optional)	Count of detected errors.	uint8

Troubleshooting

- Each input frame must contain more than (N-K)*2 symbols, and fewer than or equal to N symbols. A shortened code is inferred when the number of valid data samples between startIn and endIn is less than N. A shortened code still requires N cycles to perform the Chien search. If the input is less than N symbols, leave a guard interval of at least N-size inactive cycles before starting the next frame.
- The decoder can operate on up to 4 messages at a time. If the block receives the start of a fifth message before completely decoding the first message, the block drops data samples from the first message. To avoid this issue, increase the number of inactive cycles between input messages.

- 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. To get the value of B from a generator polynomial, use the `genpoly2b` function.

Parameters

Codeword length

The length of the code word in symbols, `N`, must be equal to 2^{M}`-1`, where `M` is the input word length. `M` can be between 3 and 16 bits.

Message length

The message length in symbols, `K`. The number of parity symbols, `N-K`, must be a positive even integer, greater than or equal to the input word length, `M`.

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. When you select `Auto`, the block uses `B = 1`.

B value

The starting exponent of the roots.

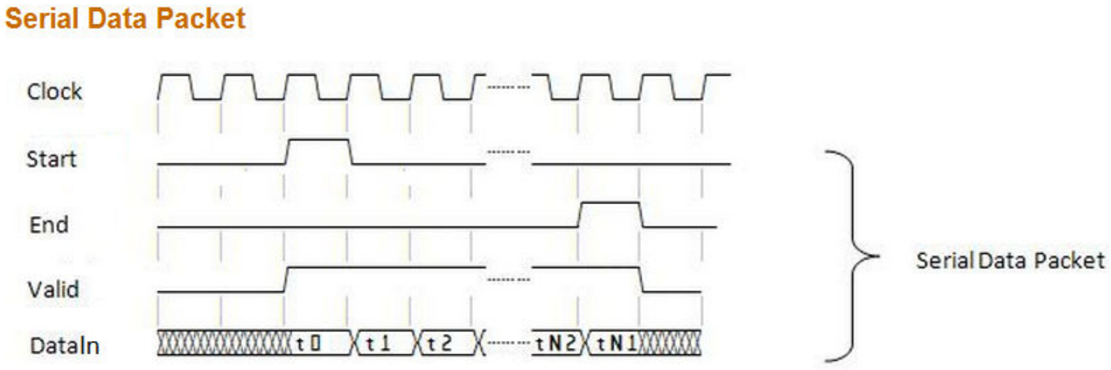
This field is available only when you select `Property` for **Source of B, the starting power for roots of the primitive polynomial**. The default is 1.

Enable number of errors output

Check this box to enable the `numErrors` output port. This port outputs the detected symbol error count.

Algorithm

Timing Diagram



HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Integer-Output RS Decoder HDL Optimized in the HDL Coder documentation.

Examples

“Using HDL Optimized RS Encoder/Decoder Library Blocks”

Pair Block

Integer-Input RS Encoder HDL Optimized

See Also

Integer-Output RS Decoder | `comm.HDLRSDecoder`

Introduced in R2012b

Integer to Bit Converter

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.

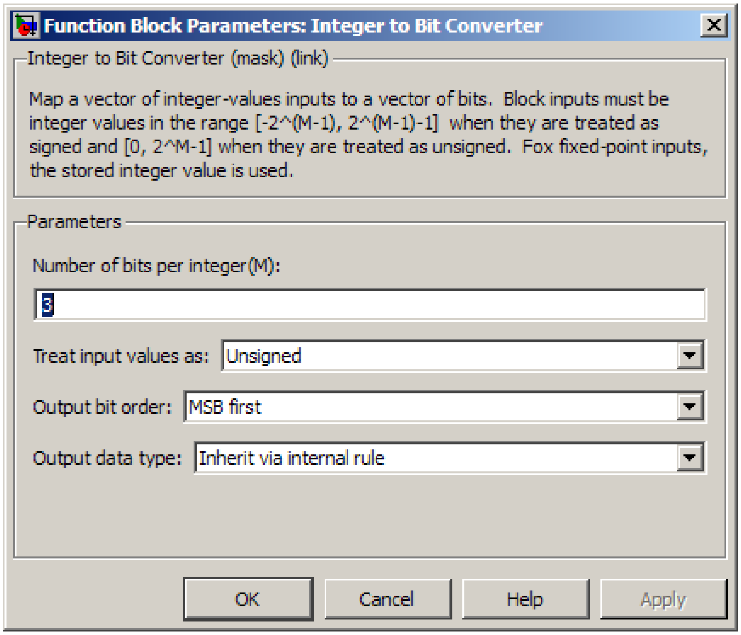
The block maps each integer value (or stored integer when you use a fixed point input) to a group of *M* bits, using the selection for the **Output bit order** to determine the most significant bit. The resulting output vector length is *M* times the input vector length.

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*·*M* column vector.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, `double`, and `fixed point`.

Dialog Box



Function Block Parameters: Integer to Bit Converter

Integer to Bit Converter (mask) (link)

Map a vector of integer-values inputs to a vector of bits. Block inputs must be integer values in the range [-2^(M-1), 2^(M-1)-1] when they are treated as signed and [0, 2^M-1] when they are treated as unsigned. Fox fixed-point inputs, the stored integer value is used.

Parameters

Number of bits per integer(M): 3

Treat input values as: Unsigned

Output bit order: MSB first

Output data type: Inherit via internal rule

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** (Simulink) 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)

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.

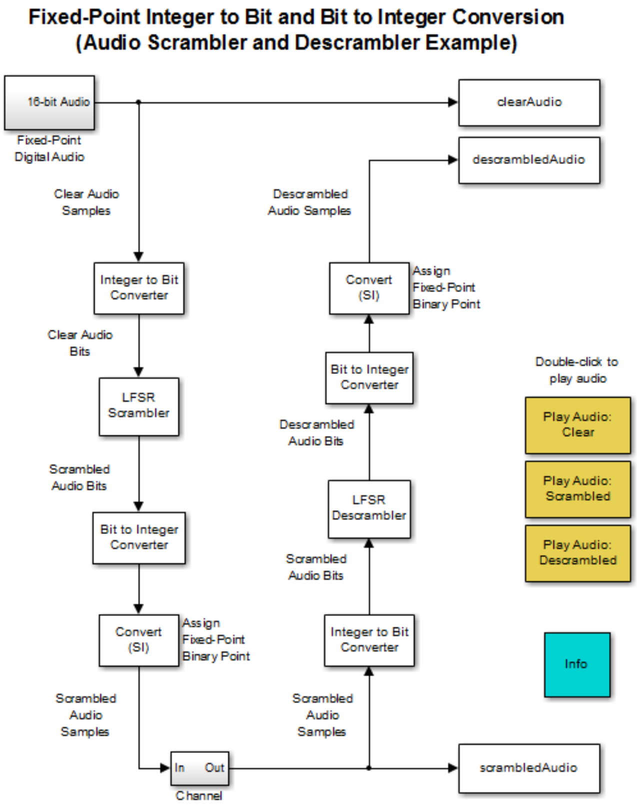
You can open the model by typing `doc_audioscrambler` at the MATLAB command line.

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.



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.

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 on page 2-96

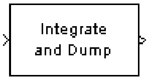
See Also

`de2bi` and `dec2bin`

Introduced before R2006a

Integrate and Dump

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*·*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 2-504.

Parameters

Integration period

The number of input samples between resets.

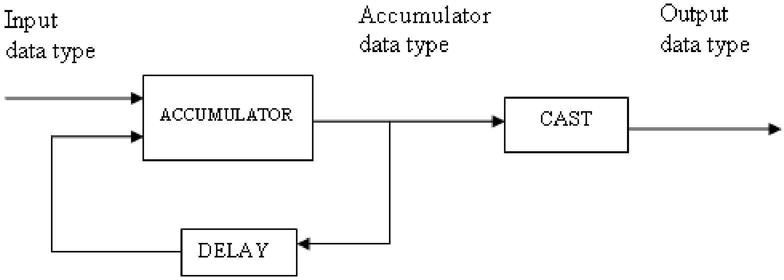
Offset

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” (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

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” (Fixed-Point Designer) 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” (DSP System Toolbox).

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.)

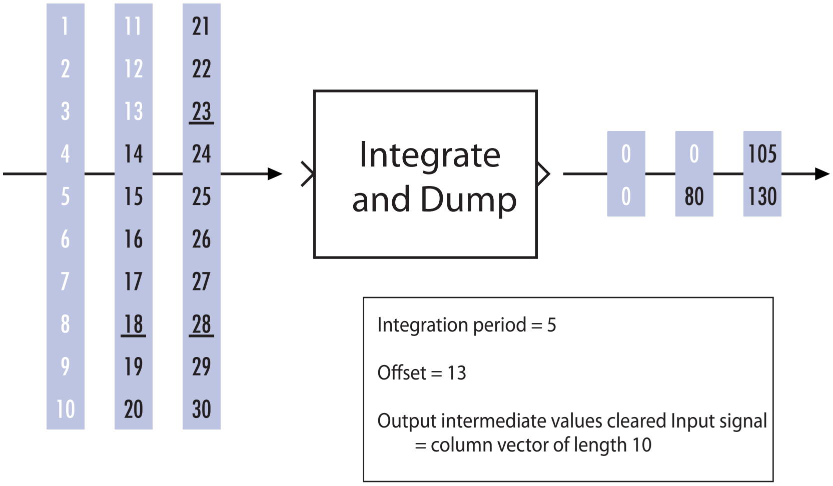
Output intermediate values Check Box	Input Signal Properties	First Several Output Values
Cleared	Scalar	0, 0, 4+5+6+7, and 8+9+10+11, where one 0 is an initial transient value and the other 0 is a delay value that results from the cleared check box and scalar value input.
Cleared	Column vector of length 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.

Output intermediate values Check Box	Input Signal Properties	First Several Output 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

Introduced before R2006a

Interlacer

Alternately select elements from two input vectors to generate output vector

O
Interlacer
E

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.

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)

Introduced before R2006a

I/Q Imbalance

Create complex baseband model of signal impairments caused by imbalances between in-phase and quadrature receiver components

Library

RF Impairments

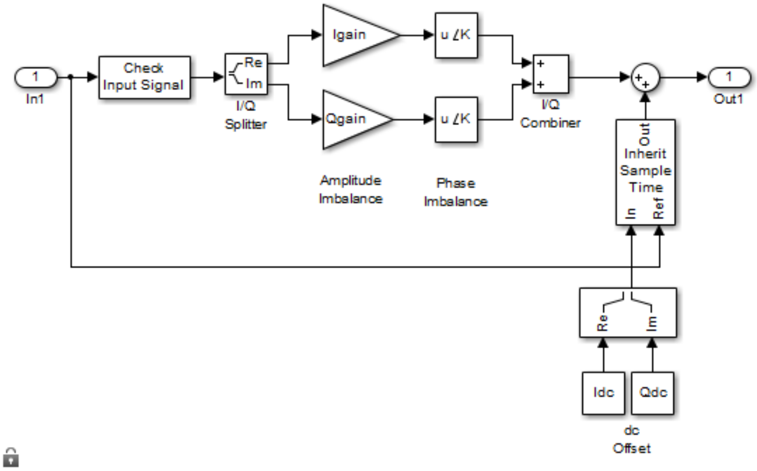
Description

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

$$y_{AmplitudeImbalance} = [10^{(0.5*\frac{I_a}{20})} * x_r] + j[10^{(-0.5*\frac{I_a}{20})} * x_i]$$

$$\triangleq y_{r_{AmplitudeImbalance}} + \text{j}*y_{i_{AmplitudeImbalance}}$$

For an I/Q phase imbalance I_p

$$y_{PhaseImbalance} =$$

$$[\exp(-0.5 * j * \pi * \frac{I_p}{180}) * y_{r_{AmplitudeImbalance}}] + \{\exp[j(\frac{\pi}{2} + 0.5 * \pi * \frac{I_p}{180})] * y_{i_{AmplitudeImbalance}}\}$$

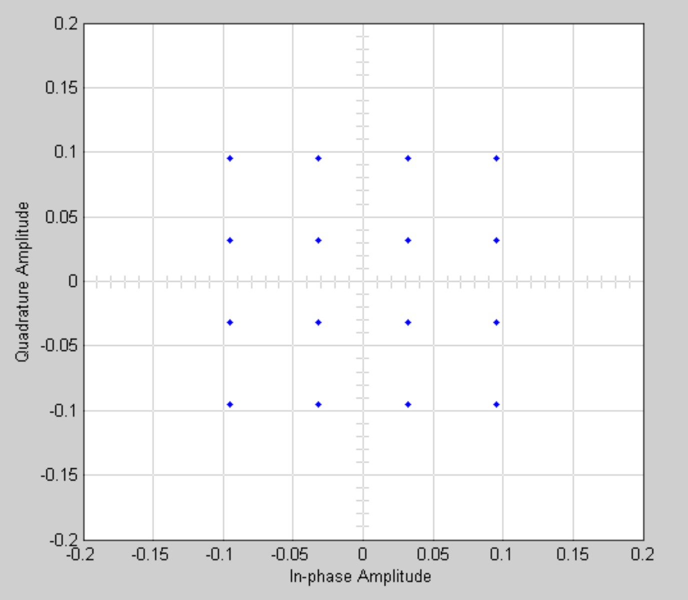
$$\triangleq \; y_{r_{PhaseImbalance}} + j * y_{i_{PhaseImbalance}}$$

For DC offsets I_{DC} and Q_{DC}

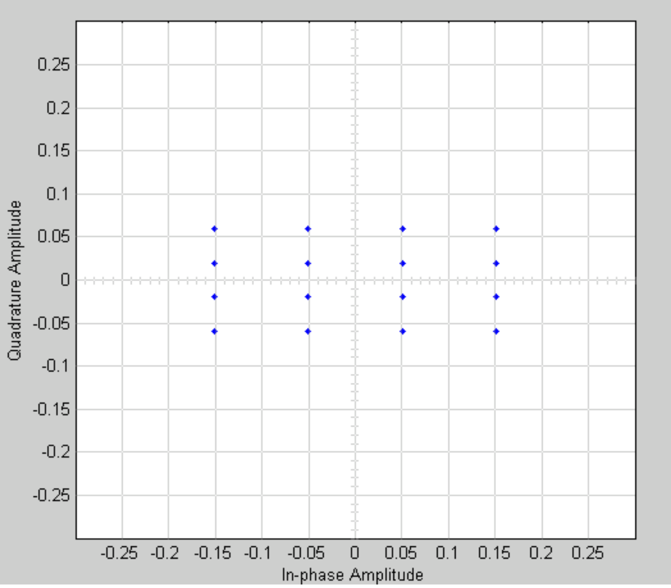
$$y = (y_{r_{PhaseImbalance}} + I_{DC}) + j * (y_{i_{PhaseImbalance}} + Q_{DC})$$

The value of the **I/Q amplitude imbalance (dB)** parameter is divided between the in-phase and quadrature components such that 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 where *X* can be positive or negative.

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:

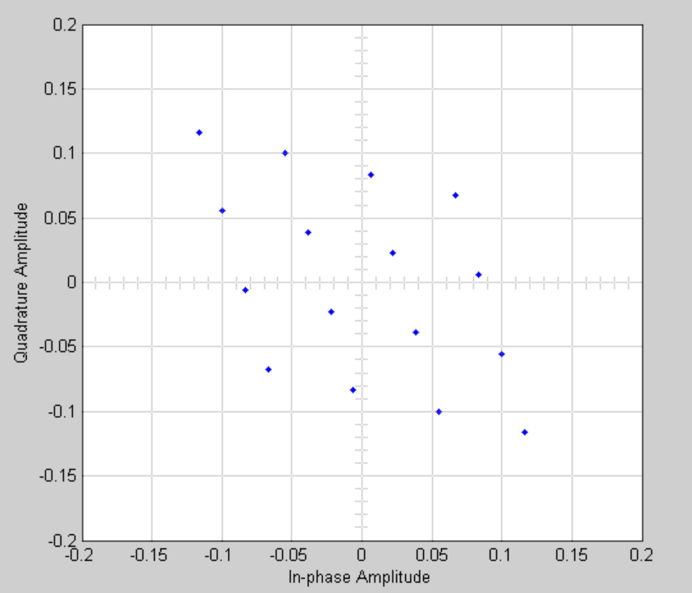


The following figure shows a scatter plot of an output signal, modulated by 16-ary QAM, from the I/Q block with **I/Q amplitude imbalance (dB)** set to `8` and all other parameters set to `0`:

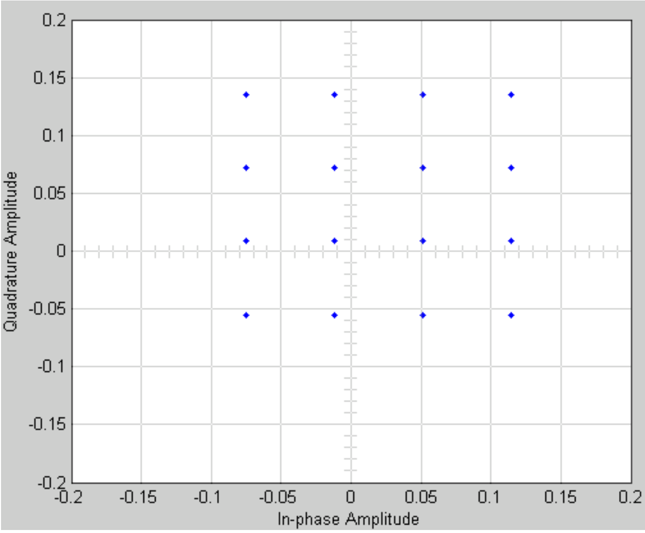


Observe that the scatter plot is stretched horizontally and compressed vertically compared to the undistorted constellation.

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.

Parameters

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.

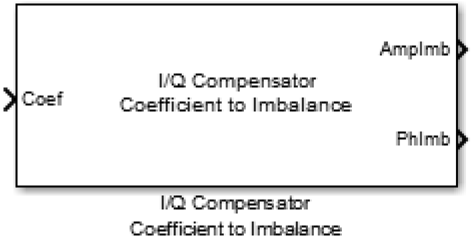
See Also

Memoryless Nonlinearity

Introduced before R2006a

I/Q Compensator Coefficient to Imbalance

Convert compensator coefficient into amplitude and phase imbalance



Library

RF Impairments Correction

Description

The I/Q Compensator Coefficient to Imbalance block converts a compensator coefficient into its equivalent amplitude and phase imbalance.

This block has a single input port, which accepts a complex coefficient or a vector of coefficients. There are amplitude and phase imbalance output ports both of which are real. The amplitude imbalance is expressed in dB while the phase imbalance is expressed in degrees.

Algorithms

See the `iqcoef2imbal` function reference page for more information on the inputs, outputs, and algorithms.

Supported Data Types

Port	Supported Data Types
Compensator Coefficient	• Double-precision, complex floating point • Single-precision, complex floating point
Amplitude Imbalance (dB)	• Double-precision floating point • Single-precision floating point
Phase Imbalance (deg)	• Double-precision floating point • Single-precision floating point

See Also

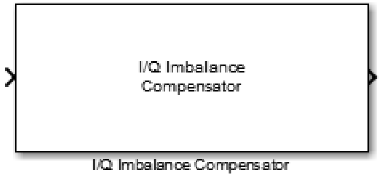
I/Q Imbalance Compensator

`iqcoef2imbal`

Introduced in R2014b

I/Q Imbalance Compensator

Compensate for imbalance between in-phase and quadrature components



Library

RF Impairments Correction

Description

The I/Q Imbalance Compensator mitigates the effects of an amplitude and phase imbalance between the in-phase and quadrature components of a modulated signal. The supported modulation schemes include OFDM, M-PSK, and M-QAM, where M > 2.

This block accepts up to three input ports, of which one is the input signal. When you set the **Source of compensator coefficient** parameter to `Estimated from input signal`, two additional input ports are enabled. The first is enabled when you set the **Source of adaptation step size** parameter to `Input port` and the second is enabled when you check the **Coefficient adaptation input port** box. The two options are independent. Additionally, you can check the **Estimated coefficient output port** box to create an optional output port from which the estimated compensator coefficients are made available.

When you set the **Source of compensator coefficient** parameter to `Input port`, only one possible configuration is possible (input signal port, coefficient input port, and output signal port).

Parameters

Source of compensator coefficient

Specify the source of the compensator coefficients as `Estimated from input signal` or `Input port`. If set to `Estimated from input signal`, the compensator calculates the coefficients from the input signal. If set to `Input port`, all other properties are disabled and you must provide the coefficients through the input port. The default value is `Estimated from input signal`.

Initial compensator coefficient

Specify the initial coefficient used by the internal algorithm to compensate for the I/Q imbalance. The default value is `0+0j`.

Source of adaptation step size

Specify the source of the adaptation step size as `Property` or `Input port`. If set to `Property`, specify the step size in the **Adaptation step size** field. If set to `Input port`, you must specify the step size through an input port. The default value is `Property`.

Adaptation step size

Specify the step size of the adaptation algorithm as a real scalar. This parameter is available only when **Source of adaptation step size** is set to `Property`. The default value is `0.00001`.

Coefficient adaptation input port

Select this check box to create an input port that permits a signal to control the adaptation process. If the check box is selected and if the input signal is `true`, the estimated compensation coefficients are updated. If the adaptation port is not enabled or if the input signal is `false`, the compensation coefficients do not change. By default, the check box is not selected.

Estimated coefficient output port

Select this check box to provide the estimated compensation coefficients to an output port. By default, the check box is not selected.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.IQImbalanceCompensator` reference page. The object properties correspond to the block parameters.

Examples

Compensate for I/Q Imbalance

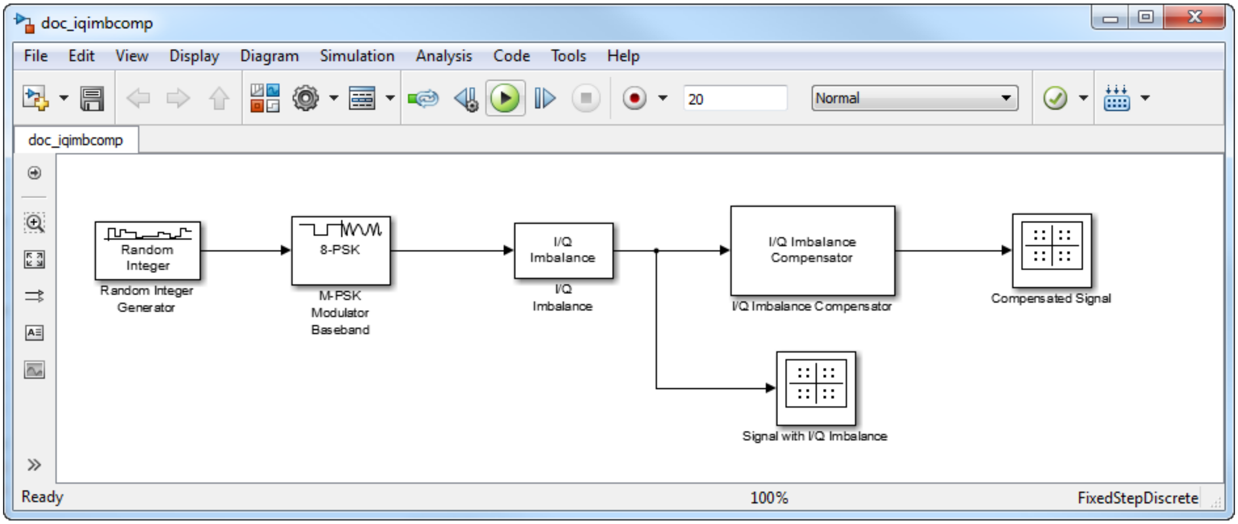
This example shows how to use the I/Q Imbalance Compensator block to remove the effects of an amplitude and phase imbalance on a modulated signal.

Open the model, doc_iqimbcomp, from the MATLAB command prompt.

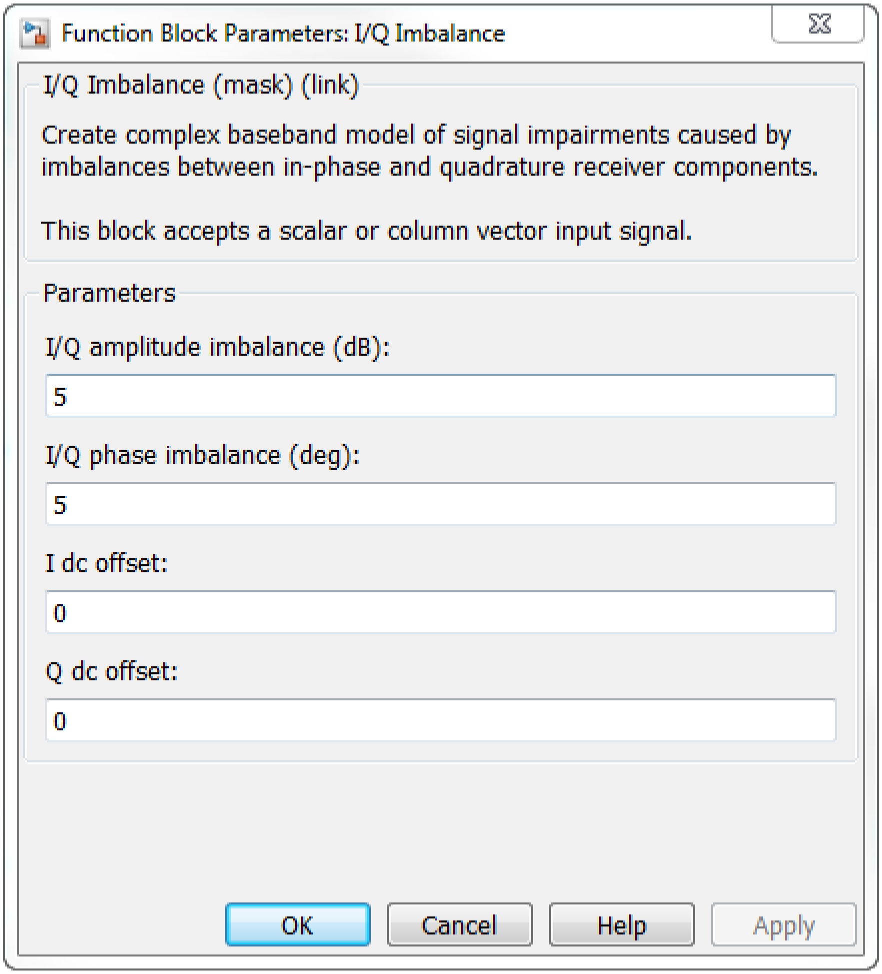
```
doc_iqimbcomp
```

The model includes these blocks:

- Random Integer Generator
- M-PSK Modulator Baseband
- I/Q Imbalance
- I/Q Imbalance Compensator
- Constellation Diagram



Double-click the I/Q Imbalance block. You can see that the **I/Q amplitude imbalance (dB)** parameter is set to 5 and the **I/Q phase imbalance (deg)** parameter is also set to 5.


Function Block Parameters: I/Q Imbalance

I/Q Imbalance (mask) (link)

Create complex baseband model of signal impairments caused by imbalances between in-phase and quadrature receiver components.

This block accepts a scalar or column vector input signal.

Parameters

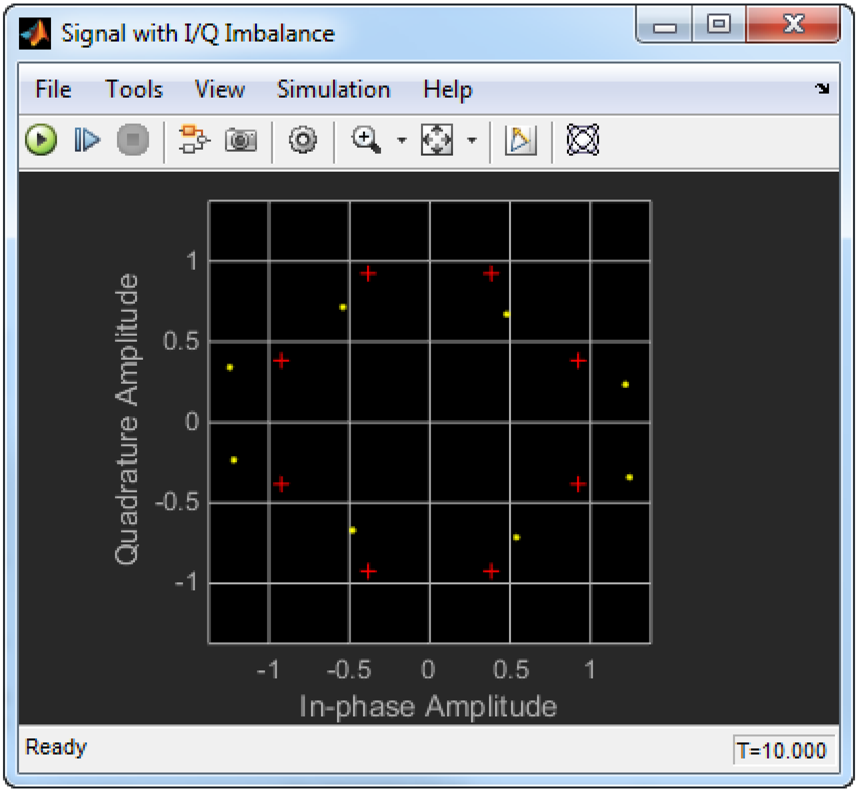
I/Q amplitude imbalance (dB): 5

I/Q phase imbalance (deg): 5

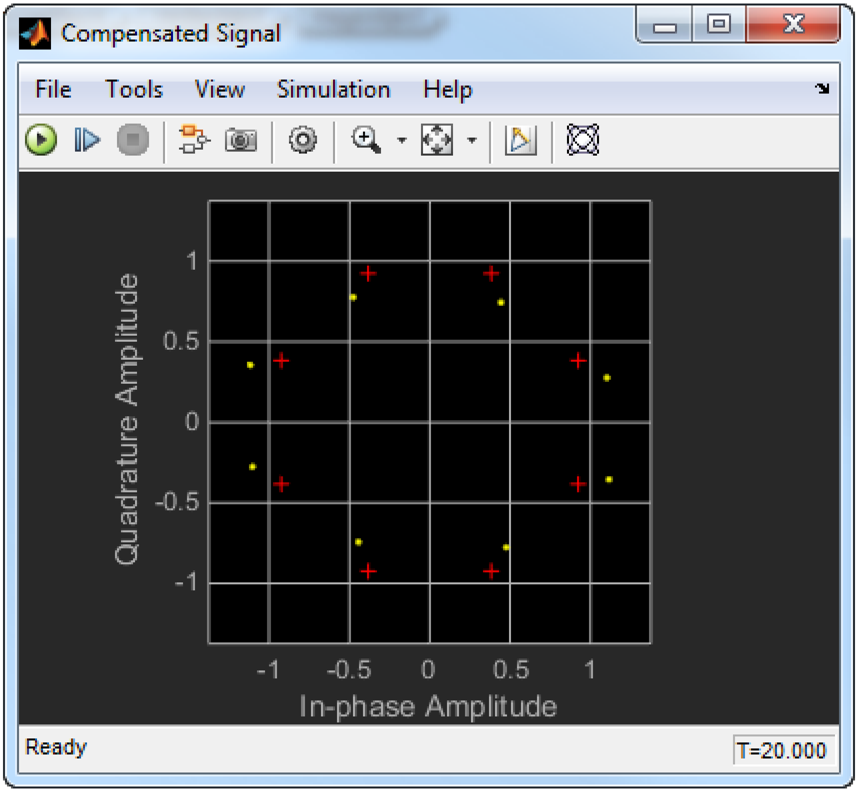
I dc offset: 0

Q dc offset: 0

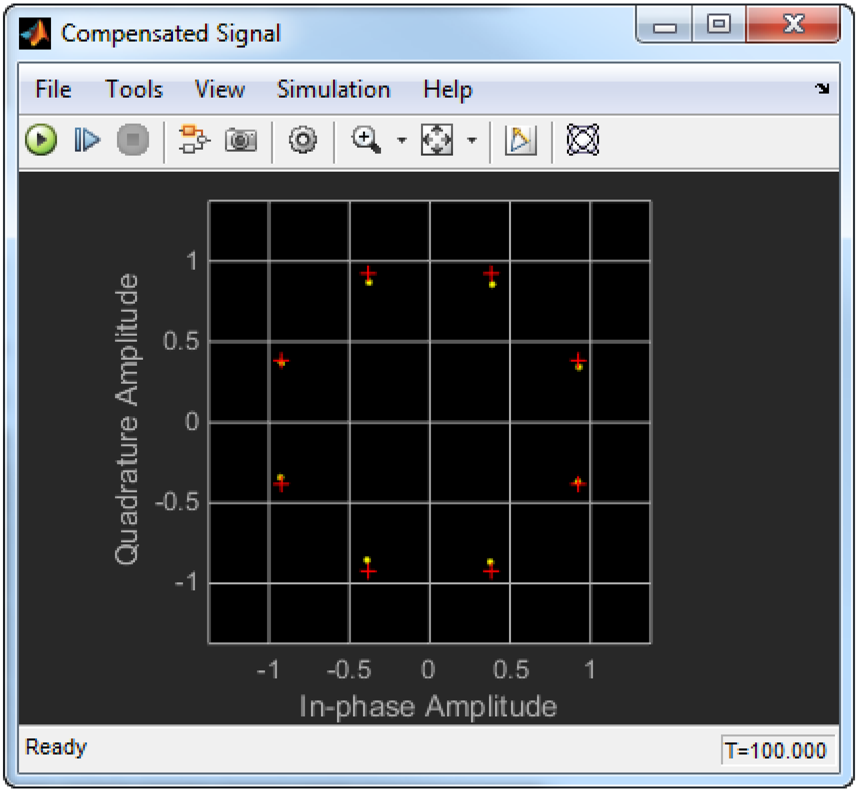
Run the model. In the *Signal with I/Q Imbalance* constellation diagram, observe the effects of the amplitude imbalance and phase imbalance on the 8-PSK signal.



Look at the *Compensated Signal* constellation diagram. Observe that the signal is not well aligned with the reference constellation (shown in red).



Increase the simulation time from 20 seconds to 100 seconds and run the model again. You can see that the constellation is now well aligned with the reference constellation. This is because the compensation algorithm is adaptive; consequently, it requires time to accurately estimate the I/Q imbalance.



Try changing other simulation parameters such as the step size in the I/Q Imbalance Compensator block, the amplitude and phase imbalance in the I/Q Imbalance block, the modulation type etc. Observe the effects on the *Compensated Signal* constellation diagram.

Supported Data Types

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

Port	Supported Data Types
Signal Output	• Double-precision floating point • Single-precision floating point
Step Size	• Double-precision floating point • Single-precision floating point
Adaptation	• Logical
Input Coefficients	• Double-precision floating point • Single-precision floating point
Output Coefficients	• Double-precision floating point • Single-precision floating point

See Also

I/Q Imbalance

`comm.IQImbalanceCompensator`

`iqcoef2imbal`

`iqimbal2coef`

Selected Bibliography

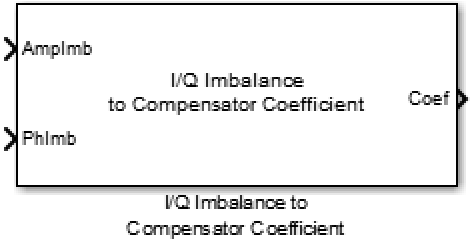
[1] Anttila, L., M. Valkama and M. Renfors. “Blind Compensation of Frequency-Selective I/Q Imbalances in Quadrature Radio Receivers: Circularity-Based Approach”. Proc. IEEE ICASSP. 2007, pp. III-245 -III-248.

[2] Kiayani, A., L. Anttila, Y. Zou, and M. Valkama, “Advanced Receiver Design for Mitigating Multiple RF Impairments in OFDM Systems: Algorithms and RF Measurements”. Journal of Electrical and Computer Engineering. Vol. 2012.

Introduced in R2014b

I/Q Imbalance to Compensator Coefficient

Converts amplitude and phase imbalance into I/Q compensator coefficient



Library

RF Impairments Correction

Description

The I/Q Imbalance to Compensator Coefficient block returns a complex coefficient to compensate for amplitude and phase imbalance.

This block has an amplitude imbalance input port and a phase imbalance input port, where the amplitude imbalance is a real number expressed in dB and the phase imbalance is a real number expressed in degrees. The imbalance inputs are vectors. The complex coefficients are returned from a single output port.

Algorithms

See `iqimbal2coef` for more information on the inputs, outputs, and algorithms.

Supported Data Types

Port	Supported Data Types
Compensator Coefficient	• Double-precision, complex floating point • Single-precision, complex floating point
Amplitude Imbalance (dB)	• Double-precision floating point • Single-precision floating point
Phase Imbalance (deg)	• Double-precision floating point • Single-precision floating point

See Also

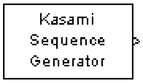
I/Q Imbalance Compensator

`iqimbal2coef`

Introduced in R2014b

Kasami Sequence Generator

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” (Simulink).

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 these formats:

- A polynomial character vector that includes the number `1`, for example, `'z^4 + z + 1'`.
- 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, `'z^8 + z^2 + 1'`, `[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 2-764 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
8	255	[8 4 3 2 0]	Small

n	N	Polynomial	Set
10	1023	[10 3 0]	Large
12	4095	[12 6 4 1 0]	Small

Parameters

Generator polynomial

Character vector or 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

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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`.

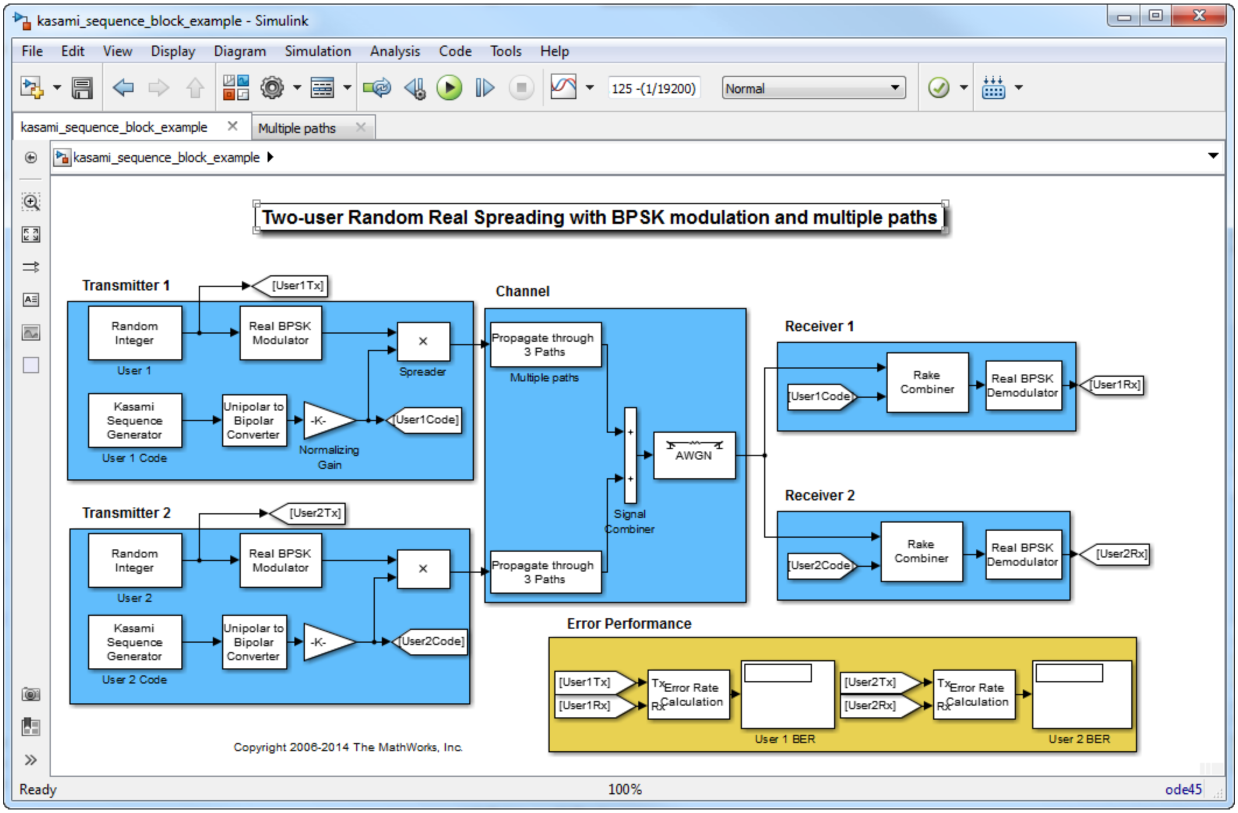
Example

Kasami Spreading with Two Users and Multipath

This model considers Kasami spreading for a combined two-user transmission in a multipath environment.

Open the model here: kasami_sequence_block_example

```
modelname = 'kasami_sequence_block_example';
open_system(modelname);
sim(modelname);
```


kasami_sequence_block_example - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
125 -(1/19200)
Normal
kasami_sequence_block_example
Multiple paths
kasami_sequence_block_example
Two-user Random Real Spreading with BPSK modulation and multiple paths
Transmitter 1
[User1Tx]
Random Integer
User 1
Real BPSK Modulator
×
Spreader
Kasami Sequence Generator
User 1 Code
Unipolar to Bipolar Converter
-K-
Normalizing Gain
[User1Code]
Transmitter 2
[User2Tx]
Random Integer
User 2
Real BPSK Modulator
×
Kasami Sequence Generator
User 2 Code
Unipolar to Bipolar Converter
-K-
[User2Code]
Channel
Propagate through 3 Paths
Multiple paths
Signal Combiner
AWGN
Propagate through 3 Paths
Receiver 1
[User1Code]
Rake Combiner
Real BPSK Demodulator
[User1Rx]
Receiver 2
[User2Code]
Rake Combiner
Real BPSK Demodulator
[User2Rx]
Error Performance
[User1Tx]
[User1Rx]
Tx Rx Error Rate Calculation
User 1 BER
[User2Tx]
[User2Rx]
Tx Rx Error Rate Calculation
User 2 BER
Copyright 2006-2014 The MathWorks, Inc.
Ready
100%
ode45

You can see very good user separation over multiple paths with the gains of combining. This can be attributed to the "good" correlation properties of Kasami sequences, which provide a balance between the ideal cross-correlation properties of orthogonal codes and the ideal auto-correlation properties of PN sequences. See the relevant examples on the Hadamard Code Generator and PN Sequence Generator reference pages.

To experiment with this model further, try selecting other path delays to see how the performance varies for the same code. Also try different codes with the same delays.

```
close_system(modelname, 0);
```

See Also

Gold Sequence Generator, PN Sequence Generator, Hadamard Code Generator

Reference

[1] Peterson and Weldon, *Error Correcting Codes*, 2nd Ed., MIT Press, Cambridge, MA, 1972.

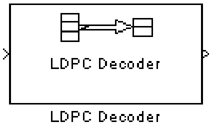
[2] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

[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.

Introduced before R2006a

LDPC Decoder

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:

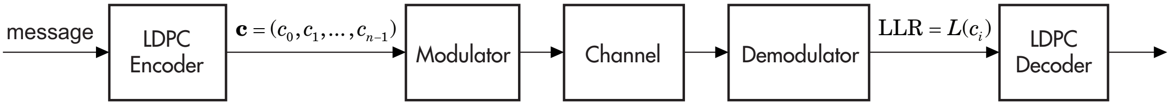
- Decode generic binary LDPC codes where no patterns in the parity-check matrix are assumed.
- Execute a number of iterations you specify or run until all parity-checks are satisfied.
- Output hard decisions or soft decisions (log-likelihood ratios) for decoded bits.

(*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

$$L(r_{ji}) = 2\,\mathrm{atanh}\left(\prod_{i' \in V_j \setminus i} \tanh\left(\frac{1}{2} L(q_{i'j})\right)\right)$$

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})$$

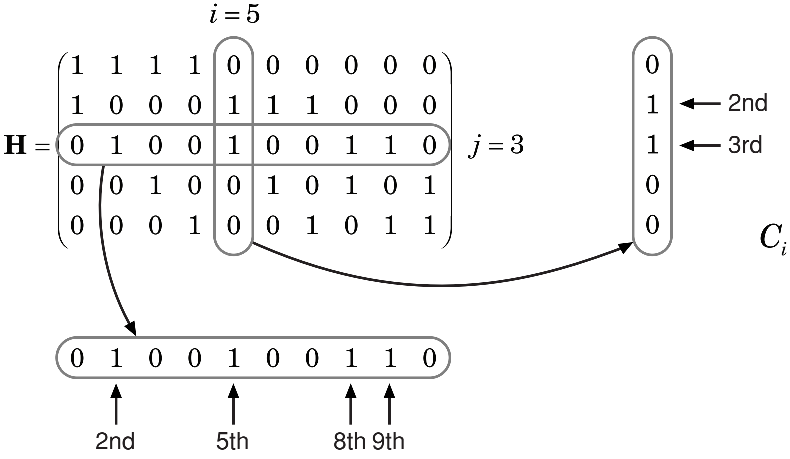
$$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i})$$

where the index sets, $C_i \setminus j$ and $V_j \setminus i$, 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



$$C_i = \{2,3\}$$
$$C_i \setminus j = \{2\}$$

$$V_j = \{2,5,8,9\}$$
$$V_j \setminus i = \{2,8,9\}$$

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.

Parameters

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

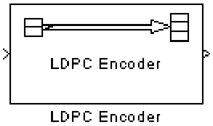
[1] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

See Also

LDPC Encoder | `comm.LDPCDecoder` | `dvbs2ldpc`

LDPC Encoder

Encode binary low-density parity-check code specified by parity-check matrix



Library

Block sublibrary of Error Detection and Correction

Description

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 2-540 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).

Parameters

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

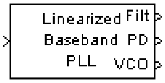
Enter `commdvbs2` at the command line to see an example that uses this block.

See Also

LDPC Decoder | `comm.LDPCEncoder` | `dvbs2ldpc`

Linearized Baseband PLL

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

Parameters

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

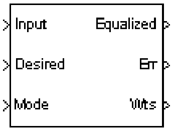
References

For more information about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization” in *Communications System Toolbox User's Guide*.

Introduced before R2006a

LMS Decision Feedback Equalizer

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.

Parameters

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.

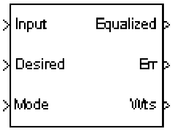
See Also

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

Introduced before R2006a

LMS Linear Equalizer

Equalize using linear equalizer that updates 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.

Parameters

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 “Adaptive Equalization with Filtering and Fading Channel” 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

Introduced before R2006a

Matrix Deinterleaver

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.

Parameters

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

Introduced before R2006a

Matrix Helical Scan Deinterleaver

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.

Parameters

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

Introduced before R2006a

Matrix Helical Scan Interleaver

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.

Parameters

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.

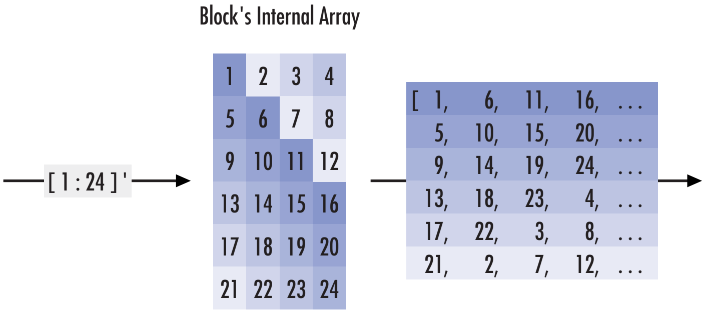
Array step size

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   2   3   4
5   6   7   8
9   10  11  12
13  14  15  16
17  18  19  20
21  22  23  24
```

```
[ 1,   6,  11,  16, ...
  5,  10,  15,  20, ...
  9,  14,  19,  24, ...
 13,  18,  23,   4, ...
 17,  22,   3,   8, ...
 21,   2,   7,  12, ...
```

```
[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

Introduced before R2006a

Matrix Interleaver

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.

Parameters

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

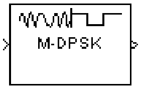
See Also

General Block Interleaver

Introduced before R2006a

M-DPSK Demodulator Baseband

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 2-562 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

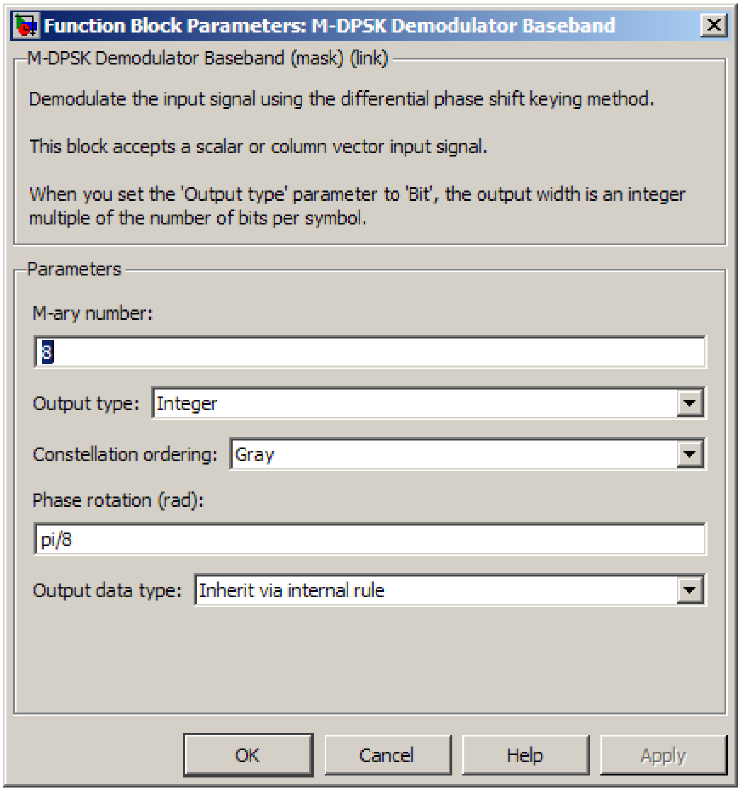
$\theta + 2\pi k/M$

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



Function Block Parameters: M-DPSK Demodulator Baseband

M-DPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential phase shift keying method.

This block accepts a scalar or column vector input signal.

When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol.

Parameters

M-ary number: 8

Output type: Integer

Constellation ordering: Gray

Phase rotation (rad): pi/8

Output data type: Inherit via internal rule

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

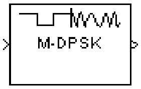
References

[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.

Introduced before R2006a

M-DPSK Modulator Baseband

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*.

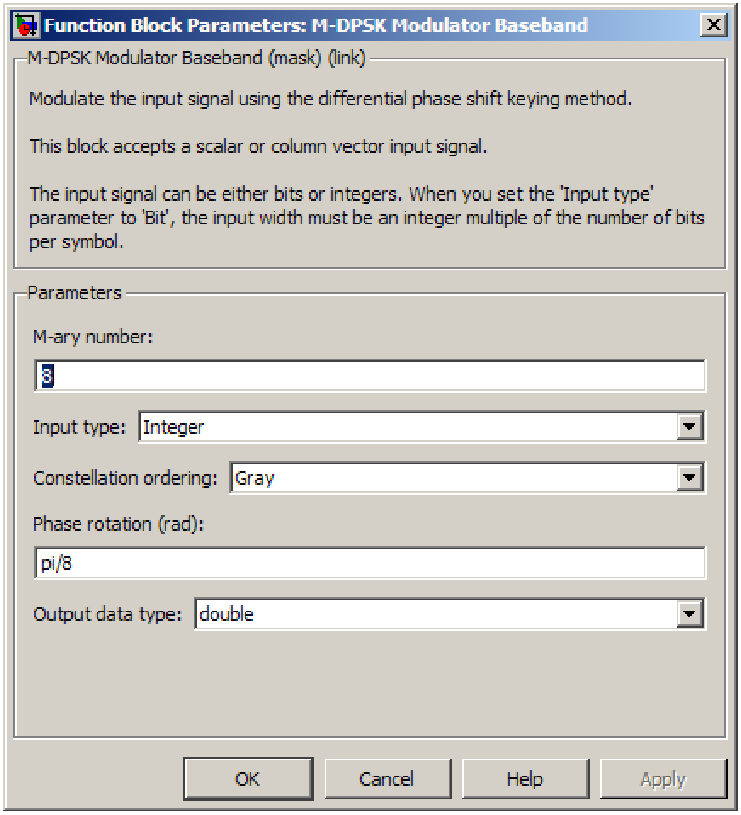
In binary input mode, the **Constellation ordering** parameter indicates how the block maps a group of *K* input bits to a corresponding phase difference. The `Binary` option uses a natural binary-to-integer mapping, while the `Gray` option uses a Gray-coded assignment of phase differences. For example, 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

Current Input	Binary-Coded Phase Difference	Gray-Coded Phase Difference
[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



Function Block Parameters: M-DPSK Modulator Baseband

M-DPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential phase shift keying method.

This block accepts a scalar or column vector input signal.

The input signal can be either bits or integers. When you set the 'Input type' parameter to 'Bit', the input width must be an integer multiple of the number of bits per symbol.

Parameters

M-ary number: 8

Input type: Integer

Constellation ordering: Gray

Phase rotation (rad): pi/8

Output data type: double

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

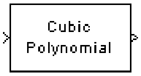
References

[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.

Introduced before R2006a

Memoryless Nonlinearity

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.

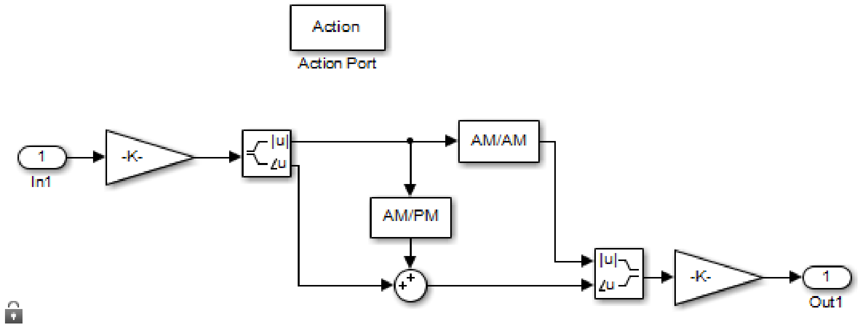
This block accepts a column vector input signal.

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 block 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.

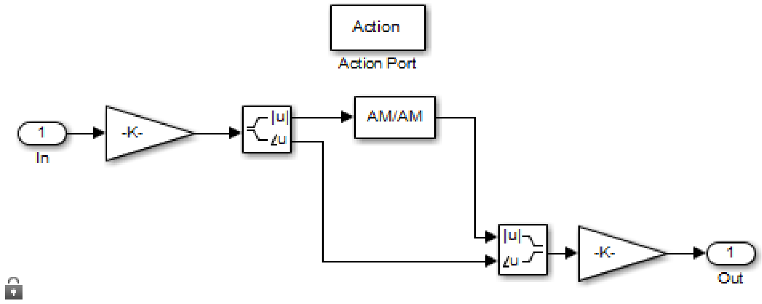


Nonlinearity Subsystem

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:

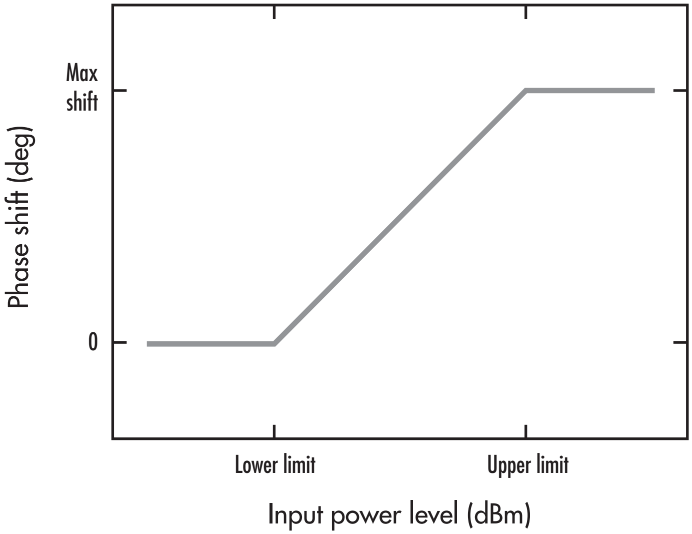
1. Multiply the signal by a gain factor.
2. Split the complex signal into its magnitude and angle components.
3. Apply an AM/AM conversion to the magnitude of the signal, according to the selected **Method**, to produce the magnitude of the output signal.
4. Combine the new magnitude and angle components 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 2-571.
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:

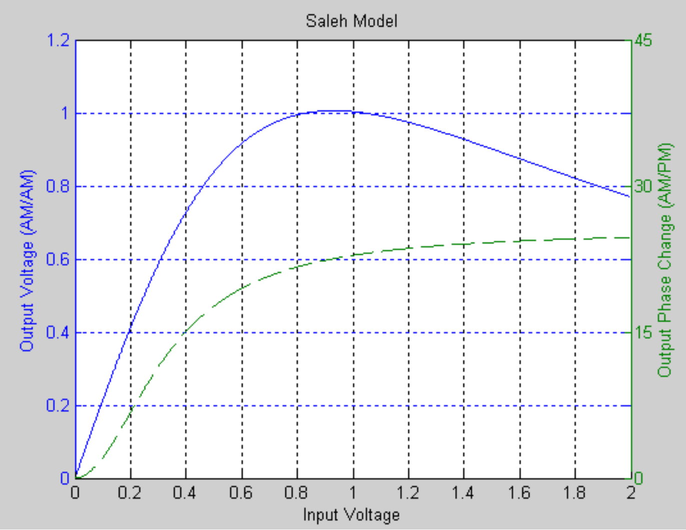


The AM/PM conversion scales linearly with input power value between the lower and upper limits of the input power level (specified by **Lower input power limit for AM/PM conversion (dBm)** and **Upper input power limit for AM/PM conversion (dBm)**). Beyond these values, AM/PM conversion is constant at 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

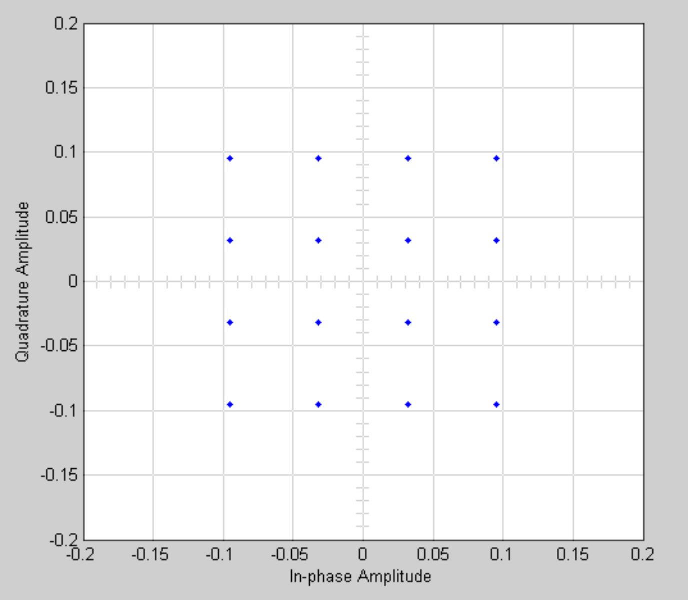
The following figure shows, for the Saleh method, plots of

- Output voltage against input voltage for the AM/AM conversion
- Output phase against input voltage for the AM/PM conversion

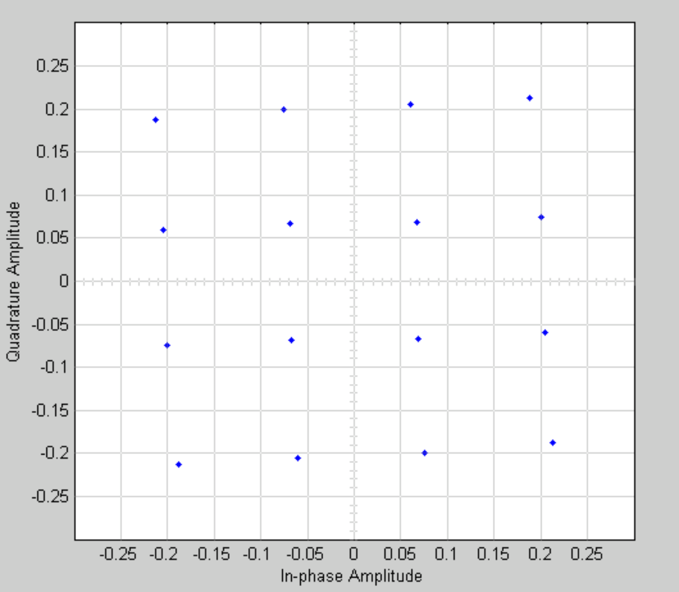


Example with 16-ary QAM

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.

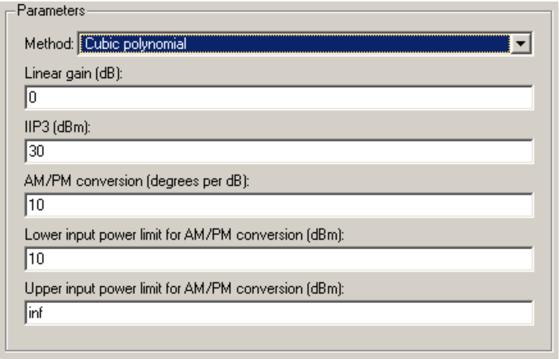
The **Output saturation level** parameter limits the output signal level.

Parameters

Method

The nonlinearity method.

The following describes specific parameters for each method.



Parameters

Method: Cubic polynomial

Linear gain (dB):
0

IIP3 (dBm):
30

AM/PM conversion (degrees per dB):
10

Lower input power limit for AM/PM conversion (dBm):
10

Upper input power limit for AM/PM conversion (dBm):
inf

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.

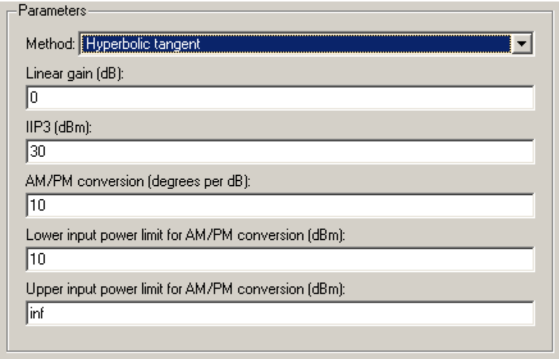
Lower input power limit (dBm)

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:

$$(\text{AM/PM conversion}) \cdot (\text{upper input power limit} - \text{lower input power limit})$$



Parameters

Method: Hyperbolic tangent

Linear gain (dB): 0

IIP3 (dBm): 30

AM/PM conversion (degrees per dB): 10

Lower input power limit for AM/PM conversion (dBm): 10

Upper input power limit for AM/PM conversion (dBm): inf

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.

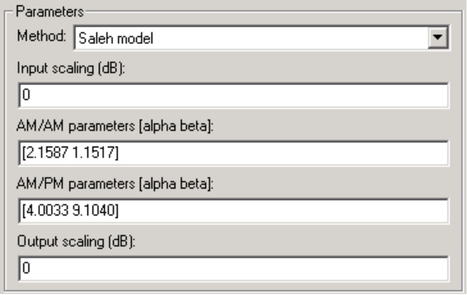
Lower input power limit (dBm)

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:

$$(\text{AM/PM conversion}) \cdot (\text{upper input power limit} - \text{lower input power limit})$$



Parameters

Method: Saleh model

Input scaling (dB):

0

AM/AM parameters [alpha beta]:

[2.1587 1.1517]

AM/PM parameters [alpha beta]:

[4.0033 9.1040]

Output scaling (dB):

0

Input scaling (dB)

Number that scales the input signal level.

AM/AM parameters [alpha beta]

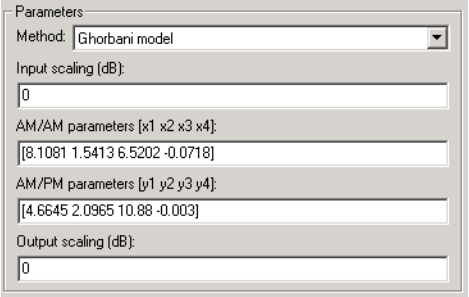
Vector specifying the AM/AM parameters.

AM/PM parameters [alpha beta]

Vector specifying the AM/PM parameters.

Output scaling (dB)

Number that scales the output signal level.



Parameters

Method: Ghorbani model

Input scaling (dB):

0

AM/AM parameters [x1 x2 x3 x4]:

[8.1081 1.5413 6.5202 -0.0718]

AM/PM parameters [y1 y2 y3 y4]:

[4.6645 2.0965 10.88 -0.003]

Output scaling (dB):

0

Input scaling (dB)

Number that scales the input signal level.

AM/AM parameters [x1 x2 x3 x4]

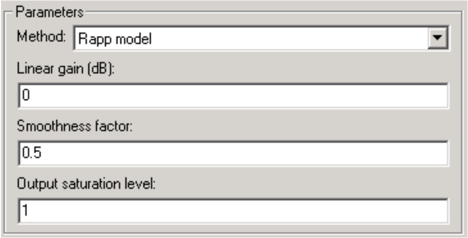
Vector specifying the AM/AM parameters.

AM/PM parameters [y1 y2 y3 y4]

Vector specifying the AM/PM parameters.

Output scaling (dB)

Number that scales the output signal level.



Parameters

Method: Rapp model

Linear gain (dB):

0

Smoothness factor:

0.5

Output saturation level:

1

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 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.

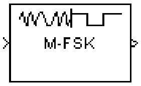
[2] A. Ghorbani, and M. Sheikhan, "The effect of Solid State Power Amplifiers (SSPAs) Nonlinearities on MPSK and M-QAM Signal Transmission", Sixth Int'l Conference on Digital Processing of Signals in Comm., 1991, pp. 193-197.

[3] C. Rapp, "Effects of HPA-Nonlinearity on a 4-DPSK/OFDM-Signal for a Digital Sound Broadcasting System", in Proceedings of the Second European Conference on Satellite Communications, Liege, Belgium, Oct. 22-24, 1991, pp. 179-184.

Introduced before R2006a

M-FSK Demodulator Baseband

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 2-588.

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, clear the **Treat each discrete rate as a separate task** checkbox (in **Simulation > Configuration Parameters > Solver**).

Parameters

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.

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

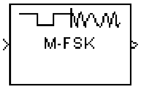
References

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

Introduced before R2006a

M-FSK Modulator Baseband

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 2-593.

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*(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, clear the **Treat each discrete rate as a separate task** checkbox (in **Simulation > Configuration Parameters > Solver**).

Parameters

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.

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

References

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

Introduced before R2006a

MIMO Fading Channel

Filter input signal through MIMO multipath fading channel
Library: Communications System Toolbox / Channels

Description

The MIMO Fading Channel block filters an input signal using a multi-input/multi-output (MIMO) multipath fading channel. This block models both Rayleigh and Rician fading and employs the Kronecker model for modeling the spatial correlation between the links. For processing details, see the Algorithms on page 2-607 section.

Signal Dimensions

The availability and dimensions of input and output port signals depends on:

- The Antenna selection parameter setting on the **Main** tab
- The Initial time source parameter setting on the **Realization** tab
- The Output channel path gains selection on the **Realization** tab

Antenna Selection Parameter	Signal Input (in)	Transmit Selection Input (Tx Sel)	Receive Selection Input (Rx Sel)	Initial Time Offset Input (Init Time)	Signal Output (Out1)	Optional Channel Gain Output (Gain)
`Off`	N_S-by-N_T	N/A	N/A	nonnegative scalar	N_S-by-N_R	N_S-by-N_P-by-N_T-by-N_R
`Tx`	N_S-by-N_{ST}	1-by-N_T	N/A		N_S-by-N_R	
`Rx`	N_S-by-N_T	N/A	1-by-N_R		N_S-by-N_{SR}	
`Tx and Rx`	N_S-by-N_{ST}	1-by-N_T	1-by-N_R		N_S-by-N_{SR}	

- N_S represents the number of samples in the input signal.
- N_T represents the number of transmit antennas, as determined by:
  - Transmit spatial correlation when Specify spatial correlation is set to `Separate Tx Rx`
  - Number of transmit antennas when Specify spatial correlation is set to `None` or `Combined`
- N_R represents the number of receive antennas, as determined by:
  - Receive spatial correlation when Specify spatial correlation is set to `Separate Tx Rx`
  - Number of receive antennas when Specify spatial correlation is set to `None`
  - Combined spatial correlation and Number of transmit antennas when Specify spatial correlation is set to `Combined`
- N_P represents the number of channel paths, as determined by the Discrete path delays (s) or Average path gains (dB).
- N_{ST} represents the number of selected transmit antennas, as determined by the number of elements set to `1` in the vector provided to the Tx Sel input port.
- N_{SR} represents the number of selected receive antennas, as determined by the number of elements set to `1` in the vector provided to the Rx Sel input port.

Ports

Input

in — Input data signal

vector

Input data signal, specified as an N_S-by-N_T or N_S-by-N_{ST} matrix.

- N_S represents the number of samples in the input signal.
- N_T represents the number of transmit antennas.
- N_{ST} represents the number of selected transmit antennas.

Data Types: `double` | `single`
Complex Number Support: Yes

`Tx Sel` — Select active transmit antennas

binary vector

Select active transmit antennas, specified as a 1-by-N_T binary vector. N_T represents the number of transmit antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Dependencies

To enable this port, on the **Main** tab, set Antenna selection to `Tx` or `Tx and Rx`.

Data Types: `double`

`Rx Sel` — Select active receive antennas

binary vector

Select active receive antennas, specified as a 1-by-N_R binary vector. N_R represents the number of receive antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Dependencies

To enable this port, on the **Main** tab, set Antenna selection to `Rx` or `Tx and Rx`.

Data Types: `double`

`Init Time` — Initial time offset

nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

Init Time must be greater than the last frame end time. When **Init Time** is not a multiple of `1/Sample Rate (Hz)`, it is rounded up to the nearest sample position.

Dependencies

To enable this port, on the **Realization** tab, set Initial time source to `Input port`.

Data Types: `double`

Output

`Out1` — Output data signal for fading channel

vector

Output data signal for the fading channel, returned as an N_S-by-N_R or N_S-by-N_{SR} matrix.

- N_S represents the number of samples in the input signal.
- N_R represents the number of receive antennas.
- N_{SR} represents the number of selected receive antennas.

`Gain` — Discrete path gains

4-D array

Discrete path gains of the underlying fading process, returned as an N_S-by-N_P-by-N_T-by-N_R array.

- N_S represents the number of samples in the input signal.
- N_P represents the number of channel paths.
- N_T represents the number of transmit antennas.
- N_R represents the number of receive antennas.

Entries for nonselected paths are filled with `NaN`.

Dependencies

To enable this port, on the **Realization** tab, select Output channel path gains.

Parameters

Main Tab

Multipath parameters (frequency selectivity)

`Inherit sample rate from input` — Option to inherit the sample rate from input

on (default) | off

Select this parameter to use the sample rate of the input signal when processing. When `Inherit sample rate from input` is selected, the sample rate is N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

`Sample rate (Hz)` — Input signal sample rate

1 (default) | positive scalar

Input signal sample rate, specified in hertz as a positive scalar. To match the model settings, set the sample rate to N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

Dependencies

This parameter appears when Inherit sample rate from input is not selected.

Data Types: `double`

`Discrete path delays (s)` — Delays for each discrete path

`0` (default) | nonnegative scalar | row vector

Delays for each discrete path in seconds, specified as a nonnegative scalar or row vector.

- When you set **Discrete path delays (s)** to a scalar, the MIMO channel is frequency flat.
- When you set **Discrete path delays (s)** to a vector, the MIMO channel is frequency selective.

Data Types: `double`

`Average path gains (dB)` — Average gain for each discrete path

`0` (default) | scalar | row vector

Average gain for each discrete path in decibels, specified as a scalar or row vector. **Average path gains (dB)** must have the same size as Discrete path delays (s).

Data Types: `double`

`Normalize average path gains to 0 dB` — Option to normalize average path gains to 0 dB

on (default) | off

Select this parameter to normalize the fading processes so that the total power of the path gains, averaged over time, is 0 dB.

`Fading distribution` — Fading distribution of channel

`Rayleigh` (default) | `Rician`

Select the fading distribution of the channel, either `Rayleigh` or `Rician`.

`K-factors` — K-factor of Rician fading channel

`3` (default) | positive scalar | row vector

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of positive-valued elements. N_P equals the value of the Discrete path delays (s) parameter.

- If you set **K-factors** to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of **K-factors**. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set **K-factors** to a row vector, the discrete path corresponding to a positive element of the **K-factors** 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 **K-factors** vector is a Rayleigh fading process.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path Doppler shifts (Hz)` — Doppler shifts for line-of-sight components

`0` (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path Doppler shifts (Hz)** to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set **LOS path Doppler shifts (Hz)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of **LOS path Doppler shifts (Hz)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path initial phases (rad)` — Initial phases for line-of-sight components

`0` (default) | scalar | row vector

Initial phases for the line-of-sight component of the Rician fading channel in radians, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path initial phases (rad)** to a scalar, it is the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set **LOS path initial phases (rad)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the elements of **LOS path initial phases (rad)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

Doppler parameters (time dispersion)

`Maximum Doppler shift (Hz)` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

Maximum Doppler shift (Hz) must be smaller than (Sample Rate (Hz)/10)/f_c for each path, where f_c is the cutoff frequency factor of the path. For more information, see “Cutoff Frequency Factor” on page 2-608.

Data Types: `double`

`Doppler spectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this parameter is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.
- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter applies when Maximum Doppler shift (Hz) is greater than zero.

If the Technique for generating fading samples parameter is set to `Sum of sinusoids`, Doppler spectrum must be `doppler('Jakes')`.

Antenna parameters (spatial dispersion)

`Specify spatial correlation` — Spatial correlation mode

`None` (default) | `Separate Tx Rx` | `Combined`

Select the spatial correlation mode: `None`, `Separate Tx Rx`, or `Combined`.

- Choose `'None'` to specify the number of transmit and receive antennas.
- Choose `'Spatial Tx Rx'` to specify the transmit and receive spatial correlation matrices separately. The number of transmit (N_T) and receive (N_R) antennas are derived from the dimensions of the Transmit spatial correlation and Receive spatial correlation parameters, respectively.
- Choose `'Combined'` to specify a single correlation matrix for the whole channel. The product of N_T and N_R is derived from the dimension of Combined spatial correlation.

`Number of transmit antennas` — Number of transmit antennas

2 (default) | positive integer

Number of transmit antennas, specified as a positive integer.

Dependencies

This parameter appears when Specify spatial correlation is `None` or `Combined`.

Data Types: `double`

`Number of receive antennas` — Number of receive antennas

2 (default) | positive integer

Number of receive antennas, specified as a positive integer.

Dependencies

This parameter appears when Specify spatial correlation is `None`.

Data Types: `double`

`Transmit spatial correlation` — Spatial correlation of transmitter

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the transmitter as an N_T-by-N_T matrix or N_T-by-N_T-by-N_P array. N_T is the number of transmit antennas, and N_P equals the value of the Discrete path delays (s) parameter.

- If **Discrete path delays (s)** is a scalar, the channel is frequency flat, and **Transmit spatial correlation** is an N_T-by-N_T Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If **Discrete path delays (s)** is a vector, the channel is frequency selective, and you can specify **Transmit spatial correlation** as a matrix. Each path has the same transmit spatial correlation matrix.
- Alternatively, you can specify **Transmit spatial correlation** as an N_T-by-N_T-by-N_P array, where each path can have its own different transmit spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Separate Tx Rx`.

Data Types: `double`
Complex Number Support: Yes

`Receive spatial correlation` — Spatial correlation of receiver

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the receiver as an N_R-by-N_R matrix or N_R-by-N_R-by-N_P array. N_R is the number of receive antennas, and N_P equals the value of the Discrete path delays (s) parameter.

- If **Discrete path delays (s)** is a scalar, the channel is frequency flat, and **Receive spatial correlation** is an N_R-by-N_R Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If **Discrete path delays (s)** is a vector, the channel is frequency selective, and you can specify **Receive spatial correlation** as a matrix. Each path has the same receive spatial correlation matrix.
- Alternatively, you can specify **Receive spatial correlation** as an N_R-by-N_R-by-N_P array, where each path can have its own different receive spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Separate Tx Rx`.

Data Types: `double`
Complex Number Support: Yes

`Combined spatial correlation` — Combined spatial correlation matrix

`[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]` (default) | matrix | 3-D array

Specify the combined spatial correlation matrix as an N_{TR}-by-N_{TR} matrix or N_{TR}-by-N_{TR}-by-N_P array, where $N_{TR} = (N_T \times N_R)$, and N_P equals the number of delay paths specified by the Discrete path delays (s) parameter.

- If Discrete path delays (s) is a scalar, the channel is frequency flat, and **Combined spatial correlation** is an N_{TR}-by-N_{TR} Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If Discrete path delays (s) is a vector, the channel is frequency selective, and you can specify **Combined spatial correlation** as a matrix. Each path has the same spatial correlation matrix.
- Alternatively, you can specify **Combined spatial correlation** as an N_{TR}-by-N_{TR}-by-N_P array, where each path can have its own different combined spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Combined`.

Data Types: `double`

`Normalize outputs by number of receive antennas` — Normalize channel output

on (default) | off

Select this parameter to normalize the channel outputs by the number of receive antennas.

`Simulate using` — Compilation type

`Interpreted execution` (default) | `Code generation`

Compilation type, specified as `Interpreted execution` or `Code generation`.

Antenna selection — Antenna mode

Off (default) | Tx | Rx | Tx and Rx

The antenna mode you select corresponds to additional input ports on the block.

Antenna selection Setting	Input Ports Added
Off	**None**
Tx	**Tx Sel**
Rx	**Rx Sel**
Tx and Rx	**Tx Sel**, **Rx Sel**

Realization Tab

Technique for generating fading samples — Channel modeling technique

Filtered Gaussian noise (default) | Sum of sinusoids

Select the channel modeling technique, either Filtered Gaussian noise or Sum of sinusoids.

Number of sinusoids — Number of sinusoids used

48 (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

This parameter appears when Technique for generating fading samples is Sum of sinusoids.

Initial time source — Source of initial time offset

Property (default) | Input port

Indicate the source of the initial time offset for the fading model, either Property or Input port.

- When you set **Initial time source** to Property, use Initial time (s) to set the initial time offset.
- When you set **Initial time source** to Input port, use the input port Init Time to set the initial time offset.

Dependencies

This parameter appears when Technique for generating fading samples is `Sum of sinusoids`.

`Initial time (s)` — Initial time offset

`0` (default) | nonnegative scalar

Initial time offset for the fading model, specified as a nonnegative scalar.

When `Initial time (s)` is not a multiple of `1/`Sample Rate (Hz), it is rounded up to the nearest sample position.

Dependencies

This parameter appears when Technique for generating fading samples is `Sum of sinusoids` and Initial time source is set to `Property`.

`Initial seed` — Random number generator initial seed

`73` (default) | nonnegative integer

Random number generator initial seed for this block, specified as a nonnegative integer.

`Output channel path gains` — Option to output channel path gains

off (default) | on

Select this parameter to add the Gain output port to the block and output the channel path gains of the underlying fading process.

Visualization Tab

`Channel visualization` — Select the channel visualization

`Off` (default) | `Impulse response` | `Frequency response` | `Doppler spectrum` | `Impulse and frequency responses`

Select the channel visualization: `Off`, `Impulse response`, `Frequency response`, `Doppler spectrum`, or `Impulse and frequency responses`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

`Antenna pair to display` — Transmit-receive antenna pair to display

`[1,1]` (default) | vector

Transmit-receive antenna pair to display, specified as a 1-by-2 vector, where the first element corresponds to the desired transmit antenna and the second corresponds to the desired receive antenna. At this time, only a single pair can be displayed.

Dependencies

This parameter appears when Channel visualization is not `Off`.

`Percentage of samples to display` — Percentage of samples to display
`25%` (default) | `10%` | `50%` | `100%`

Select the percentage of samples to display: `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This parameter appears when Channel visualization is `Impulse response`, `Frequency response`, or `Impulse and frequency responses`.

`Path for Doppler spectrum display` — Path for which Doppler spectrum is displayed
`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_P, where N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter appears when Channel visualization is `Doppler spectrum`.

Algorithms

The fading processing per link is described in Methodology for Simulating Multipath Fading Channels and assumes the same parameters for all ($N_T \times N_R$) 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 is:

$$R_H = E[R_t \otimes R_r]$$

- The ⊗ symbol represents the Kronecker product.
- R_t represents the correlation matrix at the transmit side: $R_t = E\left[H^H H\right]$, of size N_T-by-N_T.
- R_r represents the correlation matrix at the receive side: $R_r = E\left[HH^H\right]$, of size N_R-by-N_R.

You can obtain a realization of the MIMO channel matrix as:

$$H = R_r^{\frac{1}{2}} A R_t^{\frac{1}{2}}$$

A is an N_R-by-N_T matrix of independent identically distributed complex Gaussian variables with zero mean and unit variance.

Cutoff Frequency Factor

The following information explains how the cutoff frequency factor, f_c, is determined for different Doppler spectrum types:

- For any Doppler spectrum type other than Gaussian and BiGaussian, f_c equals `1`.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation•sqrt(2•log(2))`.
- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)•sqrt(2•log(2))`.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)•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:

- All random path gains are always generated and keep evolving for each link, whether or not a given link is selected. The path gain values output for the non-selected links are populated with `NaN`.
- The spatial correlation only applies to the selected transmit and/or receive antennas, and the correlation coefficients are the corresponding entries in the transmit, receive, or combined correlation matrices. In other words, the spatial correlation matrix for the selected transmit or receive antennas is a submatrix of the transmit, receive, or combined spatial correlation matrix property value.
- The input filtering through the path gains is applied to the selected links only.
- Whenever a link associated with a specific transmitter transitions from a selected state to a non-selected state, its channel filter is reset. For example, if the antenna selection property is set to `Tx` and the selected transmit antenna is changed from `2` to `1`, the channel filter corresponding to antenna `2` will be reset.
- Channel output normalization happens over the number of selected receive antennas.

References

[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.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

See Also

Blocks
AWGN Channel | SISO Fading Channel

Functions
`doppler`

System Objects
`comm.MIMOChannel`

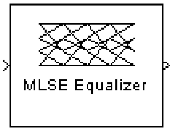
Topics

Channel Visualization

Introduced in R2013b

MLSE Equalizer

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

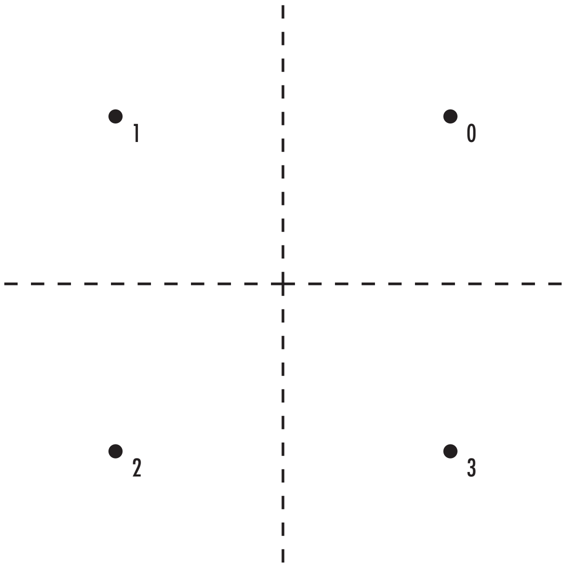
The **Signal constellation** parameter specifies the constellation for the modulated signal, as determined by the modulator in your model. **Signal constellation** is a vector of complex numbers, where the kth complex number in the vector is the constellation point to which the modulator maps the integer 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

$$
\begin{aligned}
0 &\rightarrow +1+i \\
1 &\rightarrow -1+i \\
2 &\rightarrow -1-i \\
3 &\rightarrow +1-i
\end{aligned}
$$

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 2-614 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.

Parameters

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.

- 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.

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

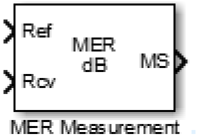
[1] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

[2] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, Wiley, 1996.

Introduced before R2006a

MER Measurement

Measure signal-to-noise ratio (SNR) in digital modulation applications



Library

Utility Blocks

Description

The MER Measurement block outputs the modulation error ratio (MER). MER is a measure of the signal-to-noise ratio (SNR) in digital modulation applications. The block measures all outputs in dB.

The MER Measurement block accepts a received signal at input port `Rcv`. It may use an ideal input signal at reference port `Ref` or, optionally, a reference constellation. 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 always outputs MER in dB, with an option to output minimum MER and *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 table shows the output type, the parameter that selects the output type, the computation units, and the corresponding measurement interval.

Output	Activation Parameter	Units	Measurement Interval
MER	None (output by default)	dB	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Minimum MER	**Output minimum MER**	dB	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Percentile MER	**Output X-percentile EVM**	dB	`Entire history`
Number of symbols	**Output X-percentile EVM** and **Output the number of symbols processed**	None	`Entire history`

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always `double`.

Algorithms

Parameters

Reference signal

Specifies the reference signal source as either `Input port` or `Estimated from reference constellation`.

Reference constellation

Specifies the reference constellation points as a vector. This parameter is available only when **Reference signal** is `Estimated from reference constellation`. The default is `constellation(comm.QPSKModulator)`.

Measurement interval

Specify the measurement interval as: `Input length`, `Entire history`, `Custom`, or `Custom with periodic reset`. This parameter affects the RMS and minimum MER outputs only.

- To calculate MER using only the current samples, set this parameter to `'Input length'`.
- To calculate MER for all samples, set this parameter to `'Entire history'`.
- To calculate MER over an interval you specify and to use a sliding window, set this parameter to `'Custom'`.
- To calculate MER over an interval you specify and to reset the object each time the measurement interval is filled, set this parameter to `'Custom with periodic reset'`.

Custom measurement interval

Specify the custom measurement interval in samples as a real positive integer. This is the interval over which the MER is calculated. This parameter is available when **Measurement interval** is `Custom` or `Custom with periodic reset`. The default is `100`.

Averaging dimensions

Specify the dimensions over which to average the MER measurements as a scalar or row vector whose elements are positive integers. For example, to average across the rows, set this parameter to `2`. The default is `1`.

This block supports var-size inputs of the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must be constant. For example, if the input size is `[1000 3 2]` and **Averaging dimensions** is `[1 3]`, then the output size is `[1 3 1]`. The number of elements in the second dimension is fixed at 3.

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 is available only 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. That is, 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 is available only when you select **Output X-percentile MER**.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

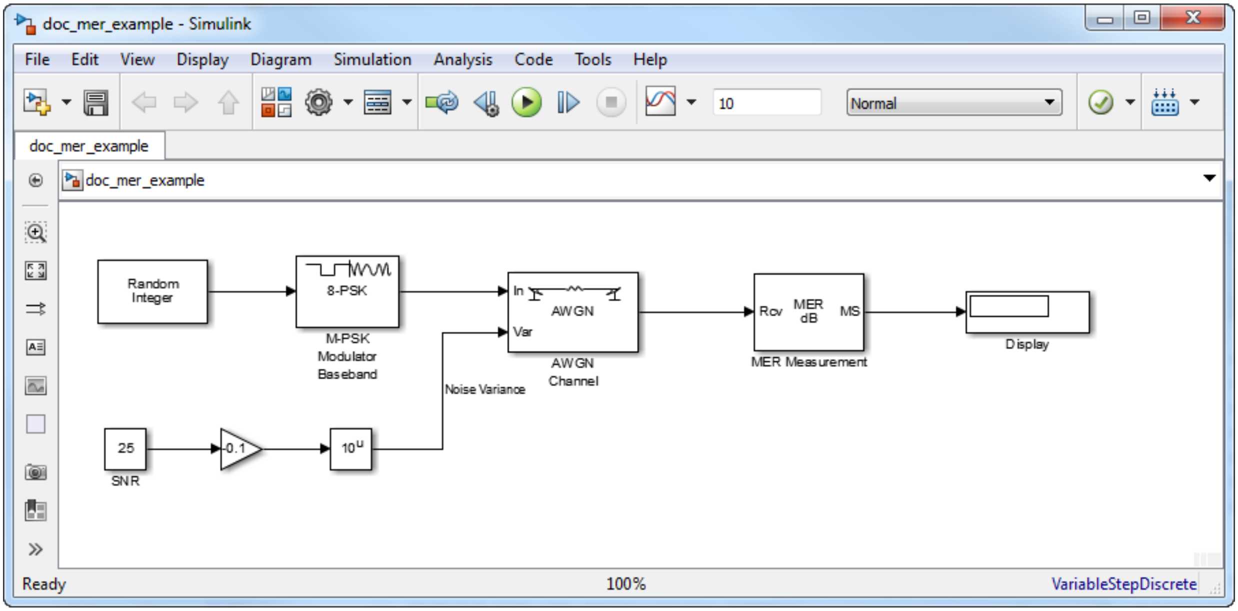
Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Examples

Measure MER of Noisy PSK Signal

Measure the MER of a noisy 8-PSK signal.

Load the model by typing doc_mer_example at the command line.



Run the model. The MER is shown in the Display block and is approximately equal to the SNR, which is set by using the Constant block. Experiment with different SNR values, and observe the effect on the estimated MER.

- “EVM and MER Measurements with Simulink”

References

[1] DVB (ETSI) Standard ETR290. *Digital Video Broadcasting (DVB): Measurement guidelines for DVB systems*. May 1997.

Algorithms

MER is a measure of the SNR in a modulated signal calculated in dB. The MER over *N* symbols is

$$MER = 10 \,{*}\log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2+Q_k^2\right)}{\sum_{n=1}^{N}\left(e_k\right)}\right)\text{dB.}$$

The MER for the *k*th symbol is

$$MER_k = 10 \,{*}\log_{10}\left(\frac{\frac{1}{N}\sum_{n=1}^{N}\left(I_k^2+Q_k^2\right)}{e_k}\right)\text{dB.}$$

The minimum MER represents the minimum MER value in a burst, or

$$MER_{\min} = \min_{k\in[1,\ldots,N]}\left\{MER_k\right\},$$

where:

- $$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$
- I_k = In-phase measurement of the *k*th symbol in the burst
- Q_k = Quadrature phase measurement of the *k*th 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 the *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 fall.

See Also

EVM Measurement | `comm.MER`

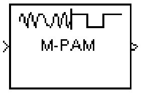
Topics

"EVM and MER Measurements with Simulink"
"Modulation Error Ratio (MER)"

Introduced in R2009b

M-PAM Demodulator Baseband

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 2-633.

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*.

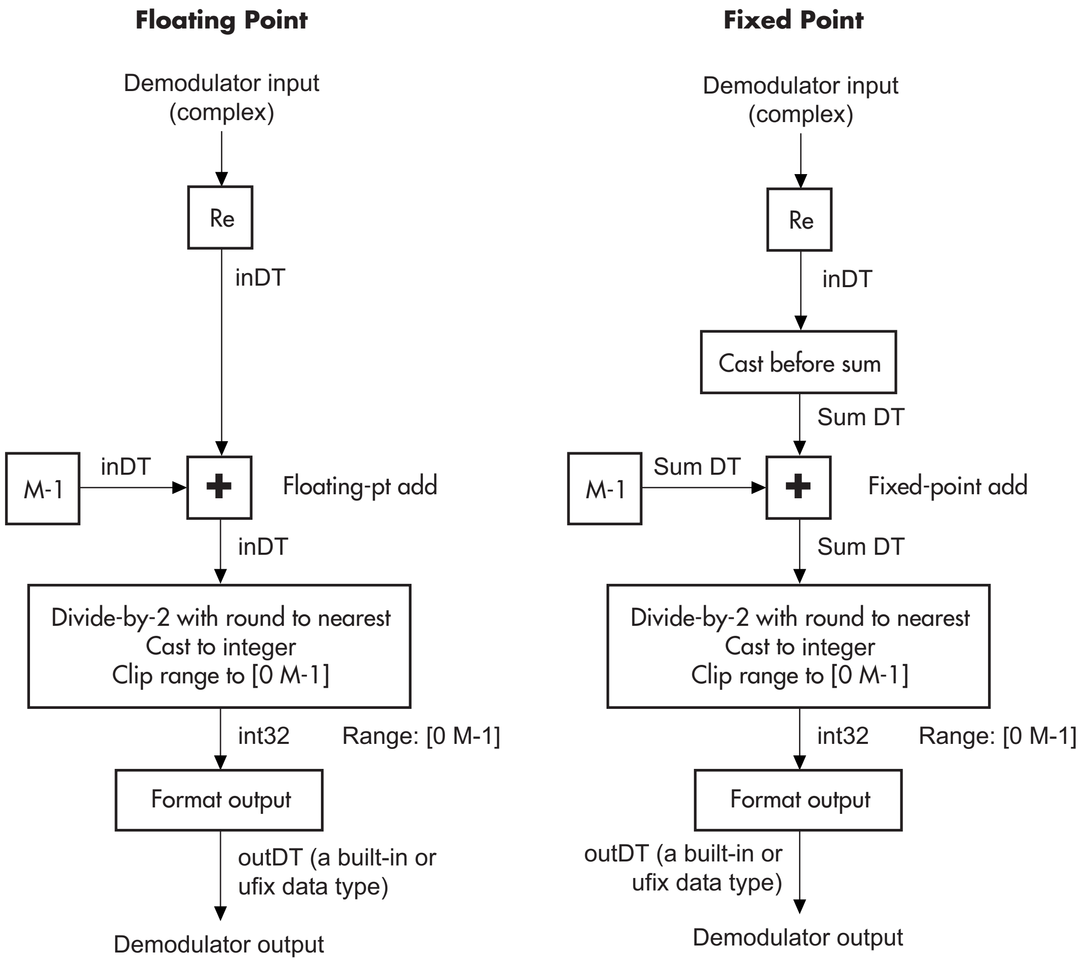
The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation. More details are on the reference page for the M-PAM Modulator Baseband block.

Algorithm

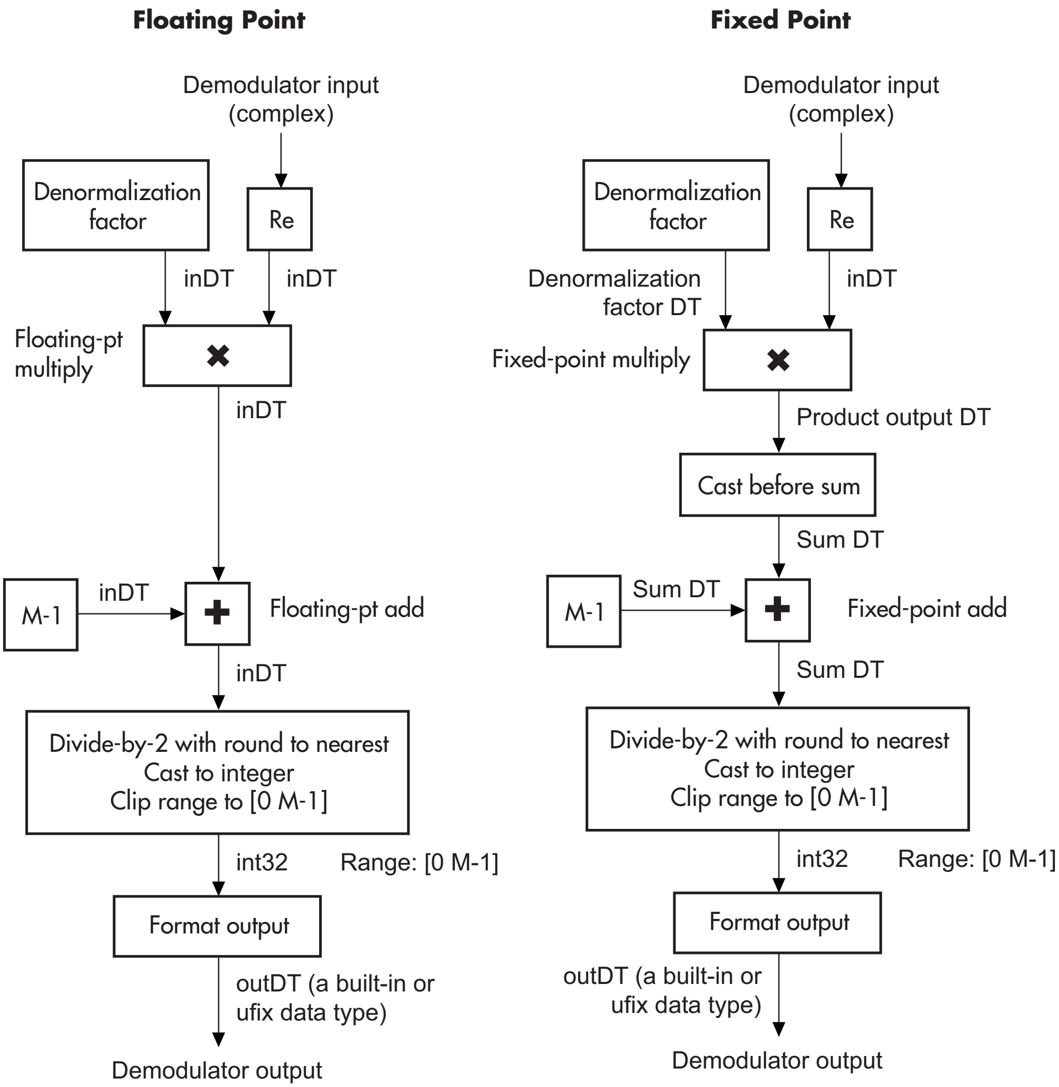
The demodulator algorithm maps received input signal constellation values to M-ary integer symbol indices between 0 and M-1 and then maps these demodulated symbol indices to formatted output values.

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

Parameters

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

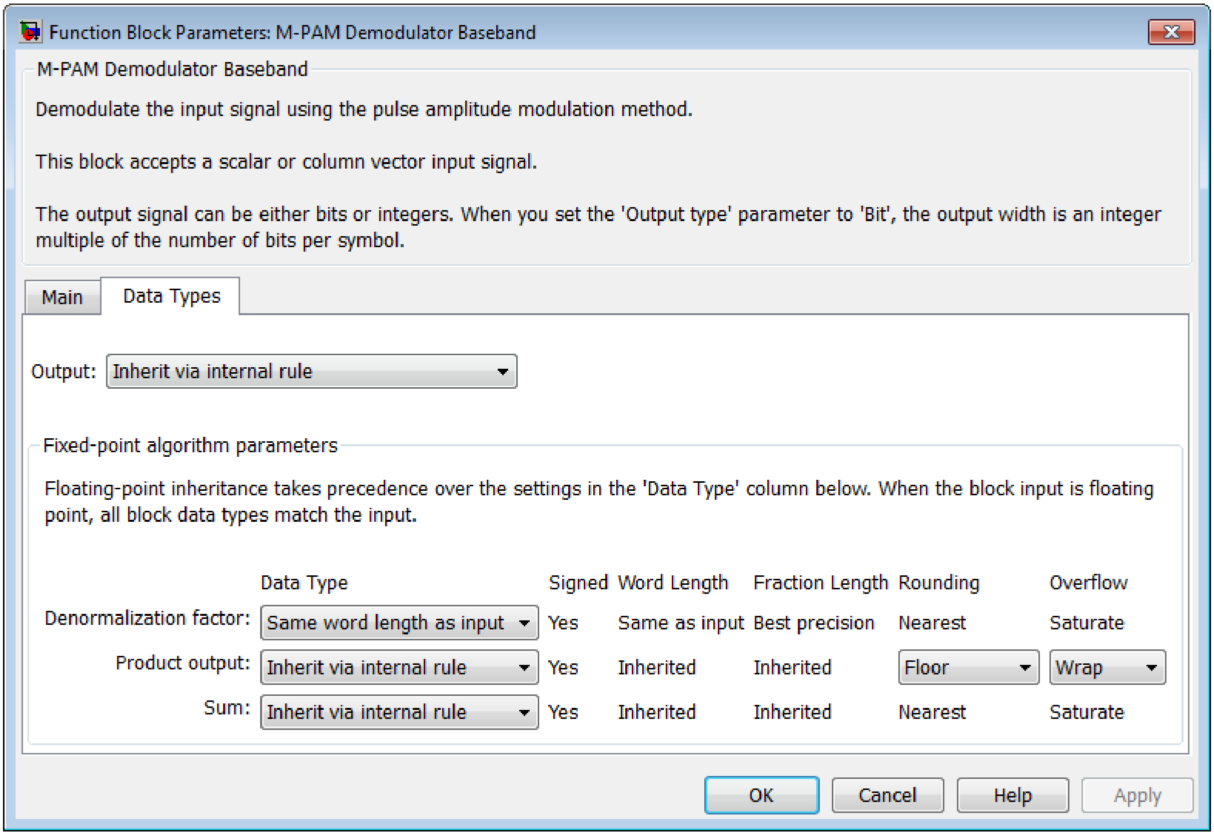
The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

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`.



Function Block Parameters: M-PAM Demodulator Baseband

M-PAM Demodulator Baseband

Demodulate the input signal using the pulse amplitude modulation method.

This block accepts a scalar or column vector input signal.

The output signal can be either bits or integers. When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol.

Output: Inherit via internal rule

Fixed-point algorithm parameters

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

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Denormalization factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Product output:	Inherit via internal rule	Yes	Inherited	Inherited	Floor	Wrap
Sum:	Inherit via internal rule	Yes	Inherited	Inherited	Nearest	Saturate

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 (DSP System Toolbox) 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” (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

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 on page 2-627 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

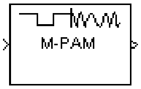
See Also

General QAM Demodulator Baseband

Introduced before R2006a

M-PAM Modulator Baseband

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*.

Parameters

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 • $ufix\left(\lceil \log_2 M \rceil\right)$ 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

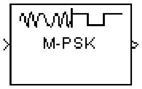
General QAM Modulator Baseband

Introduced before R2006a

M-PSK Demodulator Baseband

Demodulate PSK-modulated data

Library: Modulation / Digital Baseband Modulation / PM



Description

The M-PSK Demodulator Baseband block demodulates a baseband representation of a PSK-modulated signal. The modulation order, *M*, which is equivalent to the number of points in the signal constellation, is determined by the **M-ary number** parameter. The block accepts scalar or column vector input signals.

Input/Output Ports

Input

`Port_1` — Input signal

scalar | vector

Input port accepting a baseband representation of a PSK-modulated signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Output signal

scalar | vector

Output signal, returned as a scalar or vector. The output is a demodulated version of the PSK-modulated signal.

Data Types: `single` | `double` | `fixed point`

Parameters

`M-ary number` — Modulation order of the PSK constellation

8 (default) | scalar

Specify the modulation order as a positive integer power of two.

Example: 2 | 16

`Output type` — Output signal data type

`Integer` (default) | `Bit`

Specify the elements of the input signal as integers or bits. If **Output type** is `Bit`, the number of samples per frame is an integer multiple of the number of bits per symbol, $\log_2(M)$.

`Decision type` — Demodulator output

`Hard decision` (default) | `Log-likelihood ratio` | `Approximate log-likelihood ratio`

Specify the demodulator output to be hard decision, log-likelihood ratio (LLR), or approximate LLR. The LLR and approximate LLR outputs are used with error decoders that support soft-decision inputs such as a Viterbi decoder, to achieve superior performance. This parameter is available when **Output type** is `Bit`.

See “Phase Modulation” for algorithm details. The output values for `Log-likelihood ratio` and `Approximate log-likelihood ratio` decision types are of the same data type as the input values

`Noise variance source` — Source of noise variance

`Dialog` (default) | `Port`

Specify the source of the noise variance estimate. This parameter is available when **Decision type** is `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

- To specify the noise variance from the dialog box, select `Dialog`.
- To input the noise variance from an input port, select `Port`.

`Noise variance` — Estimate of noise variance

1 (default) | positive scalar

Specify the estimate of the noise variance as a positive scalar. This parameter is available when **Noise variance source** is `Dialog`.

This parameter is tunable in all simulation modes. 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.

Note The `Log-likelihood ratio` decision type computes 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 log-likelihood ratio`, as its algorithm does not compute exponentials.

`Constellation ordering` — Symbol mapping

`Gray` (default) | `Binary` | `User-defined`

Specify how the integer or group of $\log_2(M)$ bits is mapped to the corresponding symbol.

- When **Constellation ordering** is set to `Gray`, the output symbol is mapped to the input signal using a Gray-encoded signal constellation.
- When **Constellation ordering** is set to `Binary`, the modulated symbol is $\exp(j\phi + j2\pi m/M)$, where ϕ is the phase offset in radians, m is the integer output such that $0 \le m \le \mathrm{M} - 1$, and M is the modulation order.
- When **Constellation ordering** is `User-defined`, specify a vector of size M, which has unique integer values in the range $[0, M–1]$. The first element of this vector corresponds to the constellation point having an value of $e^{j\phi}$ with subsequent elements running counterclockwise.

Example: `[0 3 2 1]`

`Constellation mapping` — User-defined symbol mapping

`[0:7]` (default) | vector

Specify the order in which input integers are mapped to output integers. The parameter is available when **Constellation ordering** is `User-defined`, and must be a row or column vector of size *M* having unique integer values in the range [0, *M* – 1].

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.

`Phase offset (rad)` — Phase offset in radians
`pi/8` (default) | scalar

Specify, in radians, the phase offset of the initial constellation as a real scalar.

Example: `pi/4`

`Output data type` — Output data type
`Inherit via internal rule` (default) | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

Specify the data type of the demodulated output signal.

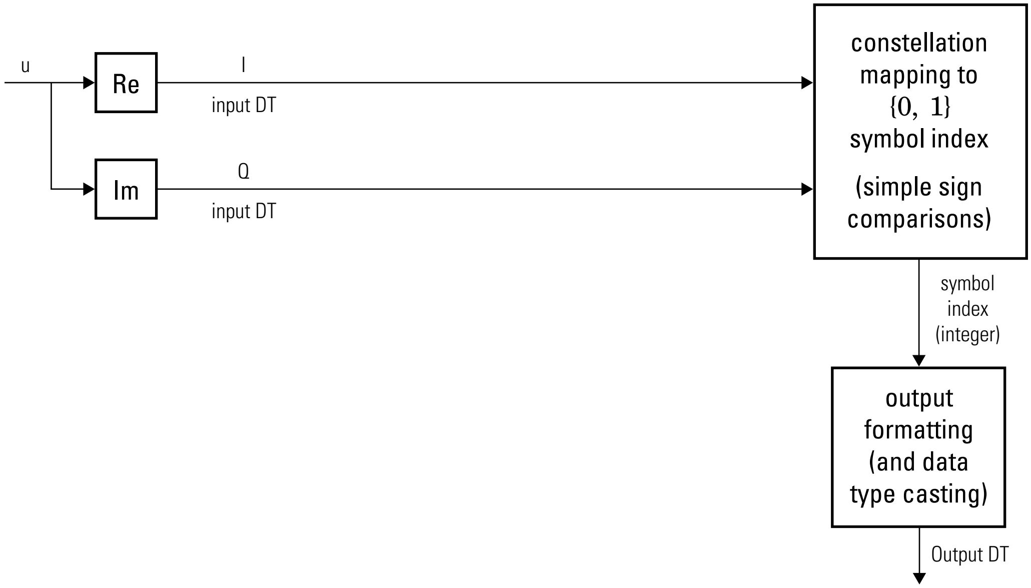
Tips

- This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see M-PSK Demodulator Baseband in the HDL Coder documentation.

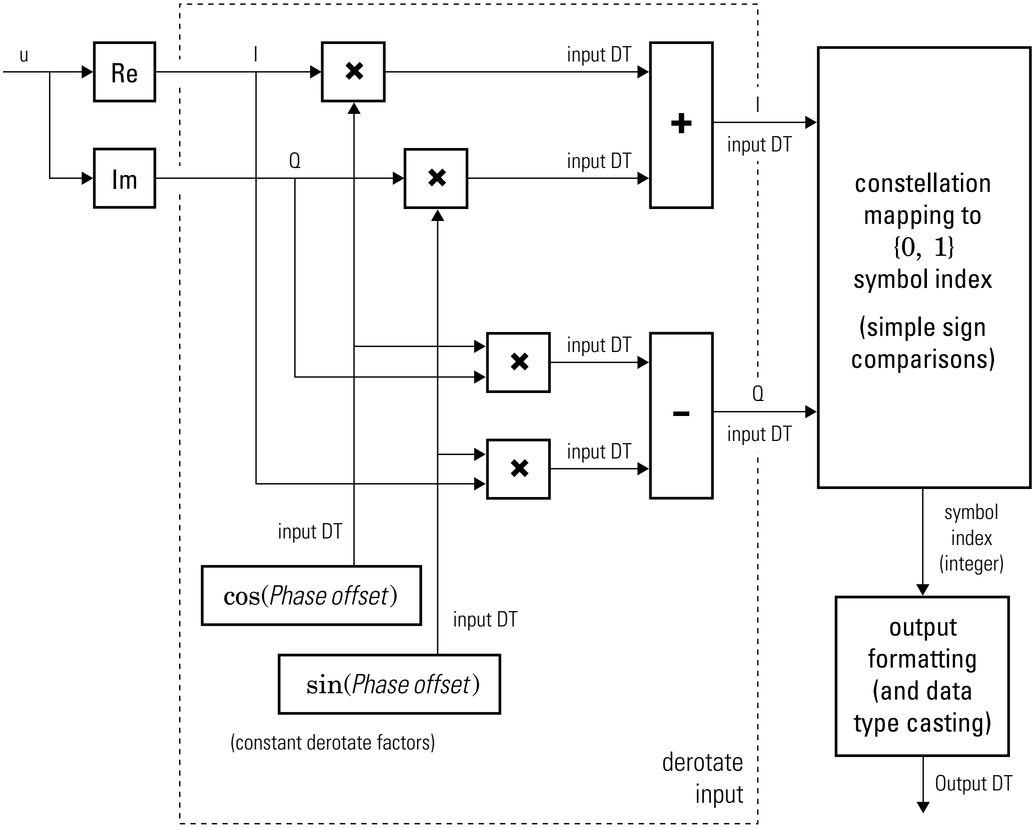
Algorithms

BPSK

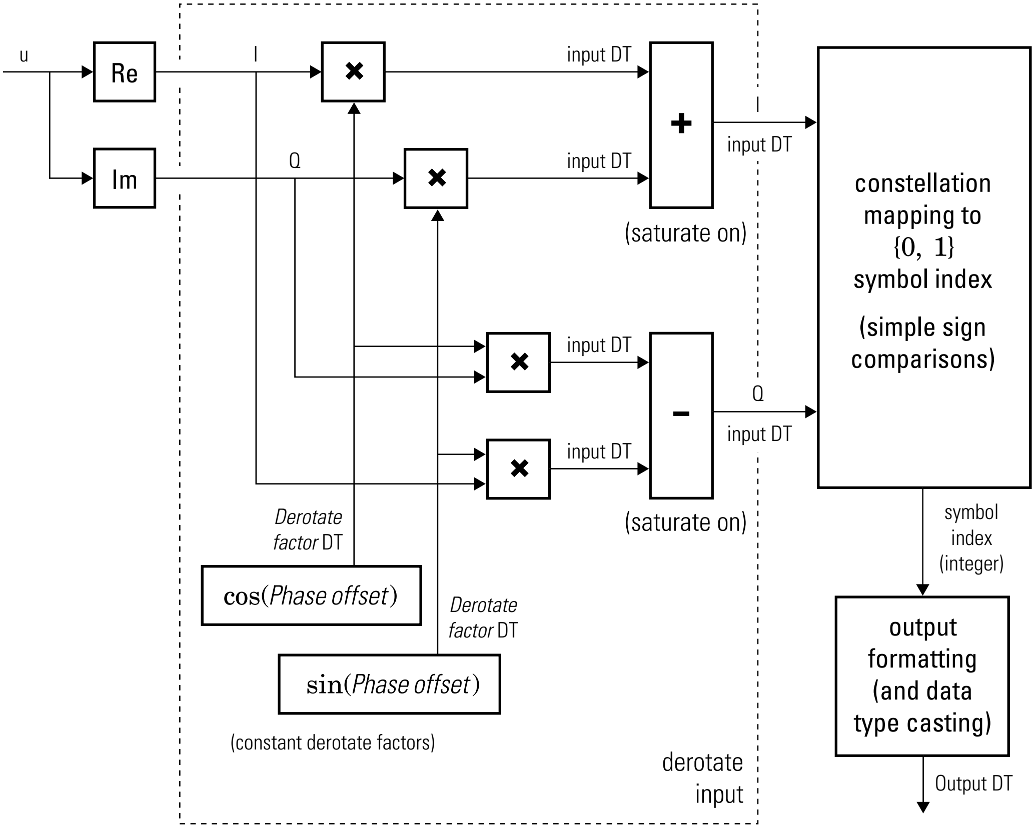
Diagrams for hard-decision demodulation of BPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)


u
Re
Im
I
Q
×
+
−
input DT
I
Q
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
cos(*Phase offset*)
sin(*Phase offset*)
(constant derotate factors)
derotate input

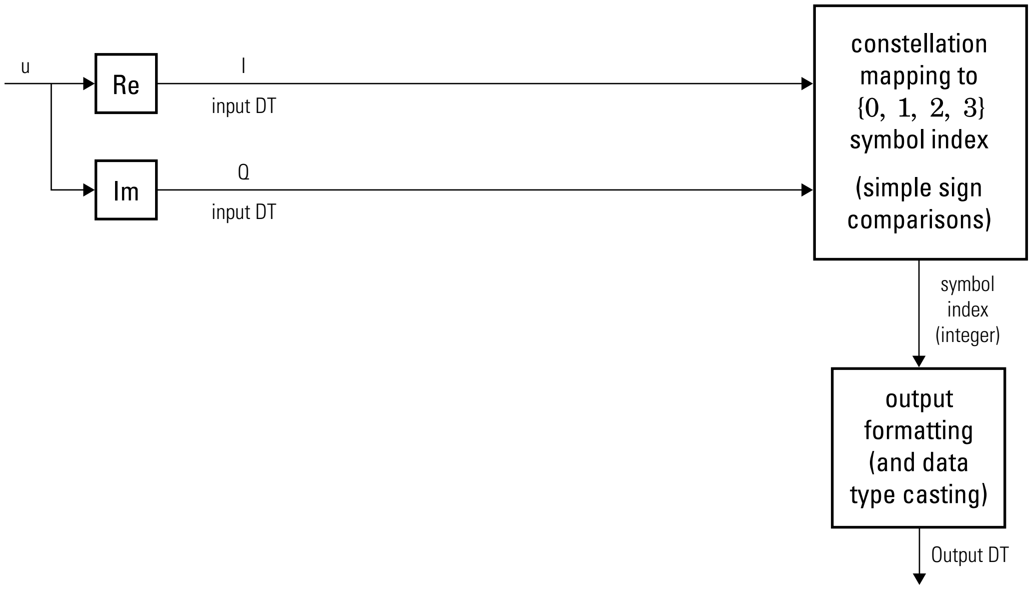
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



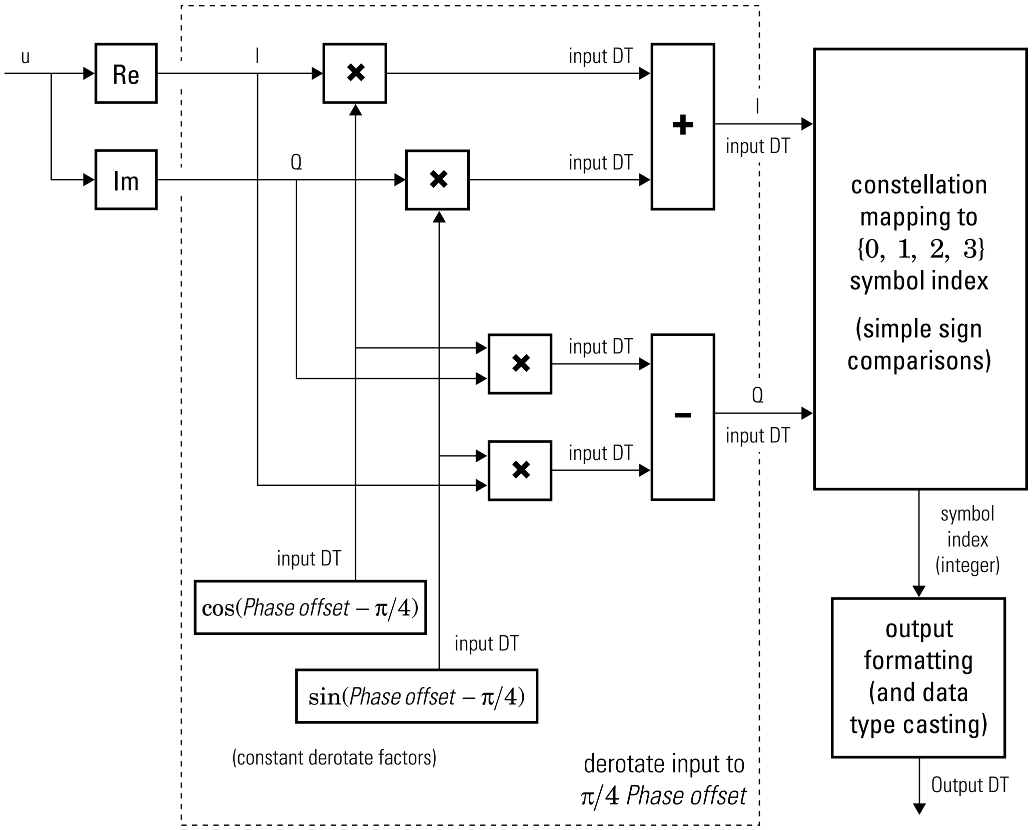
Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

QPSK

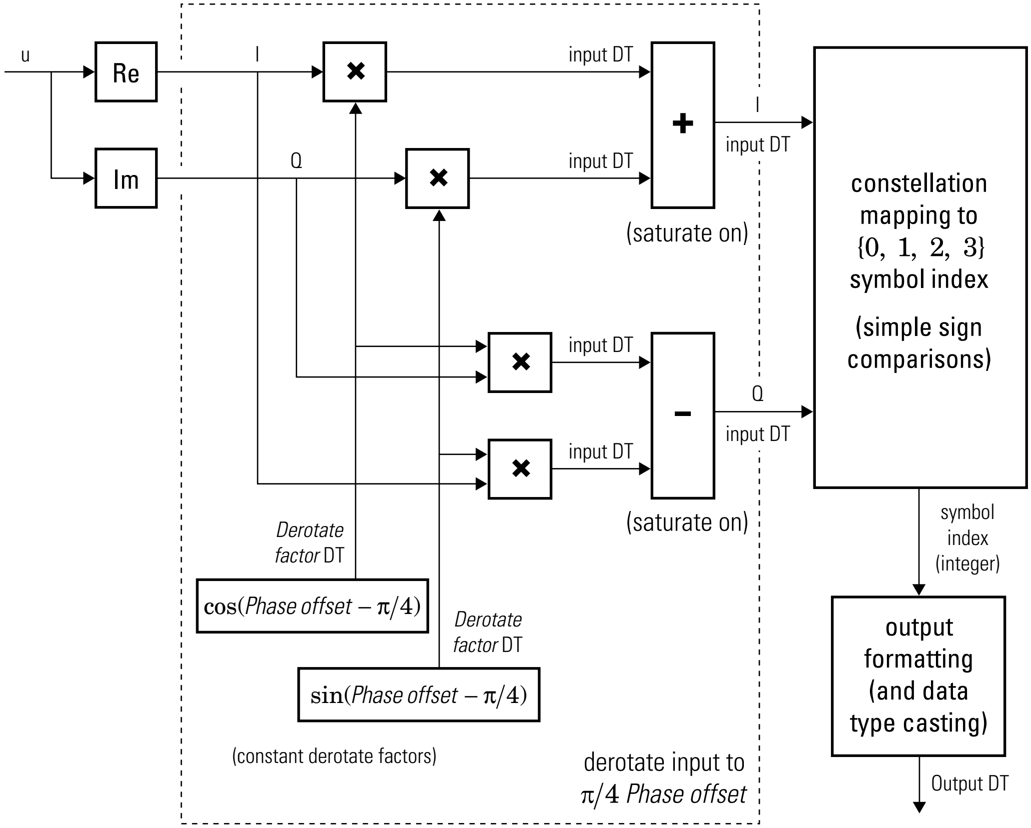
Diagrams for hard-decision demodulation of QPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to $\{0,\ 1,\ 2,\ 3\}$ symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of $\frac{\pi}{4}$)


u
Re
Im
I
Q
×
×
×
×
input DT
input DT
input DT
input DT
+
−
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
input DT
cos(Phase offset − π/4)
input DT
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset
output formatting (and data type casting)
Output DT

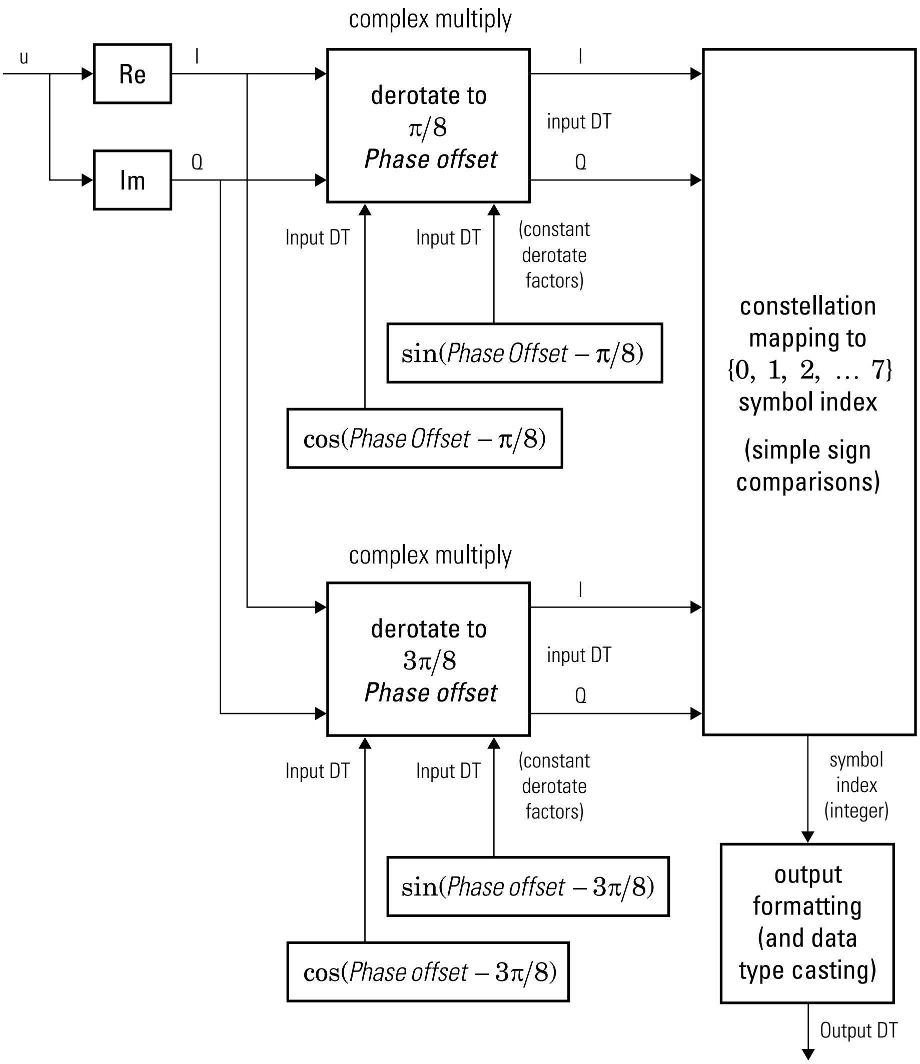
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



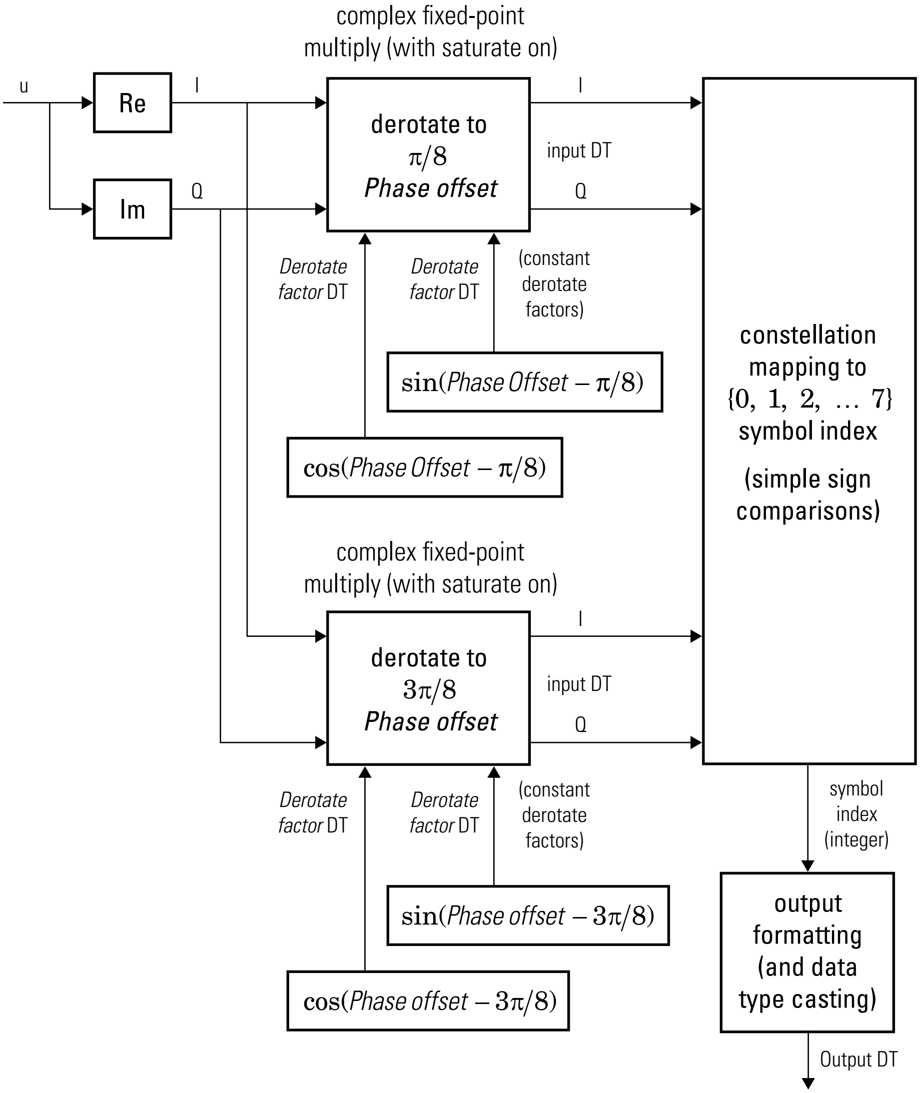
Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

Higher-Order PSK

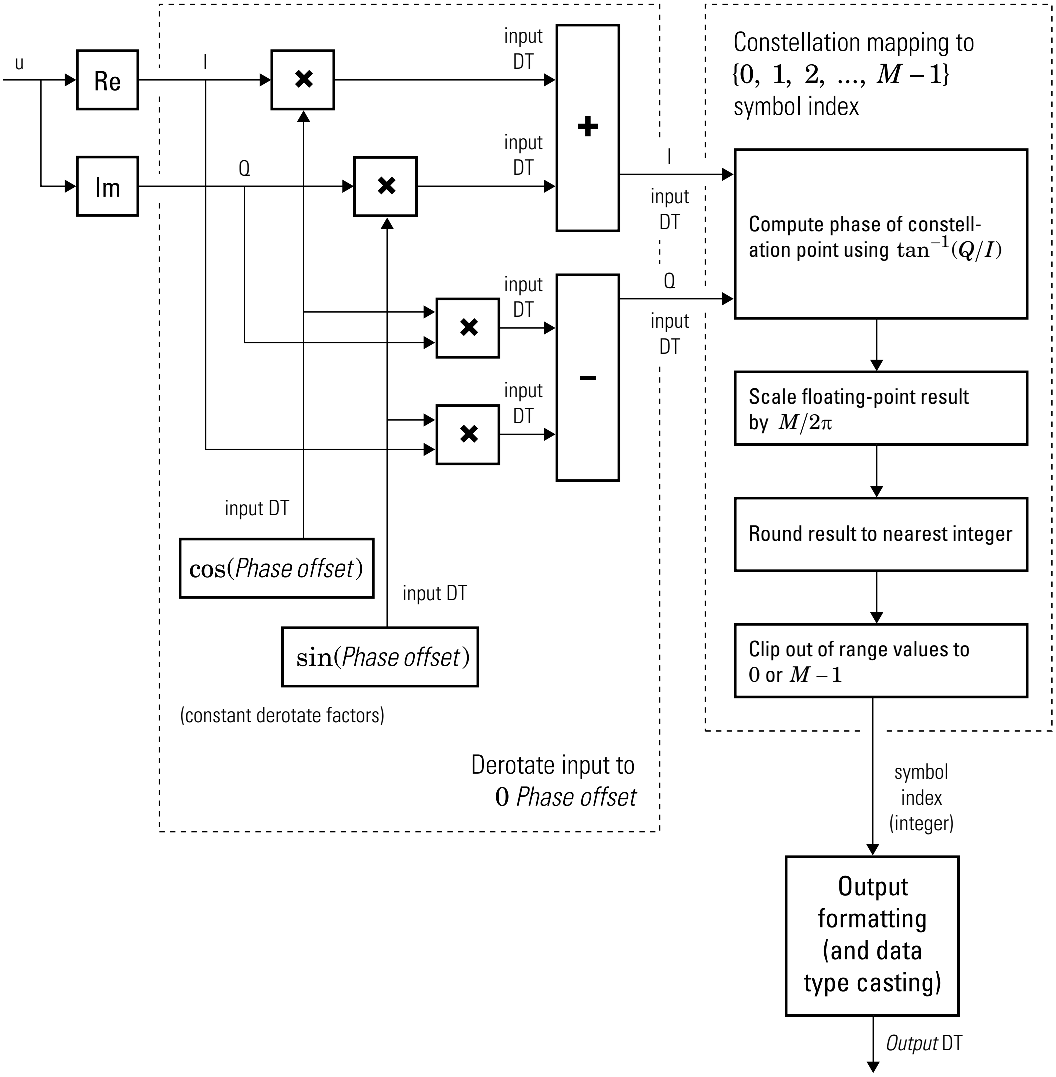
Diagrams for hard-decision demodulation of higher-order ($M \geq 8$) signals follow.



Hard-Decision 8-PSK Demodulator Floating-Point Signal Diagram


complex fixed-point
multiply (with saturate on)
u
Re
Im
I
Q
derotate to
$\pi/8$
Phase offset
I
input DT
Q
Derotate factor DT
Derotate factor DT
(constant derotate factors)
$\sin(\text{Phase Offset} - \pi/8)$
$\cos(\text{Phase Offset} - \pi/8)$
constellation mapping to
$\{0, 1, 2, \dots 7\}$
symbol index
(simple sign comparisons)
complex fixed-point
multiply (with saturate on)
derotate to
$3\pi/8$
Phase offset
I
input DT
Q
Derotate factor DT
Derotate factor DT
(constant derotate factors)
$\sin(\text{Phase offset} - 3\pi/8)$
$\cos(\text{Phase offset} - 3\pi/8)$
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision 8-PSK Demodulator Fixed-Point Signal Diagram


u
Re
Im
I
Q
input DT
input DT
input DT
input DT
input DT
input DT
input DT
input DT
+
−
I
Q
cos(*Phase offset*)
sin(*Phase offset*)
(constant derotate factors)
Derotate input to 0 *Phase offset*
Constellation mapping to {0, 1, 2, ..., $M-1$} symbol index
Compute phase of constellation point using $\tan^{-1}(Q/I)$
Scale floating-point result by $M/2\pi$
Round result to nearest integer
Clip out of range values to 0 or $M-1$
symbol index (integer)
Output formatting (and data type casting)
Output DT

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`.

Log-likelihood Ratio and Approximate Log-likelihood Ratio

The exact LLR and approximate LLR algorithms (soft-decision) are described in “Phase Modulation”.

See Also

M-DPSK Demodulator Baseband | M-PSK Modulator Baseband

Topics

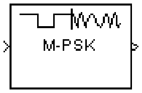
“Phase Modulation”

Introduced before R2006a

M-PSK Modulator Baseband

Modulate using M-ary phase shift keying

Library: Modulation / Digital Baseband Modulation / PM



Description

The M-PSK Modulator Baseband block modulates an input signal using M-ary phase shift keying (PSK) and returns a complex baseband output. The modulation order, *M*, which is equivalent to the number of points in the signal constellation, is determined by the **M-ary number** parameter. The block accepts scalar or column vector input signals.

Input/Output Ports

Input

`Port_1` — Input signal
scalar | vector

Specify the input signal as an integer scalar, integer vector, or binary vector.

- When **Input type** is `Integer`, specify the input signal elements as integers from 0 to *M* – 1.
- When **Input type** is `Bit`, specify the input signal as a binary vector in which the number of elements is an integer multiple of the bits per symbol. The bits per symbol is equal to $\log_2(M)$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Output signal
scalar | vector

Output signal, returned as a complex scalar or vector. The output is the complex baseband representation of the PSK-modulated signal.

Data Types: `single` | `double` | `fixed point`

Parameters

`M-ary number` — Modulation order of the PSK constellation
`8` (default) | scalar

Specify the modulation order as a positive integer power of two.

Example: `2` | `16`

`Input type` — Type of input signal
`Integer` (default) | `Bit`

Specify the elements of the input signal as integers or bits. If **Input type** is `Bit`, the number of samples per frame must be an integer multiple of the number of bits per symbol. The number of bits per symbol is $\log_2(M)$.

`Constellation ordering` — Symbol mapping
`Gray` (default) | `Binary` | `User-defined`

Specify how the integer or group of $\log_2(M)$ bits is mapped to the corresponding symbol.

- When **Constellation ordering** is set to `Gray`, the input signal is mapped to the output symbols using a Gray-encoded signal constellation.
- When **Constellation ordering** is set to `Binary`, the modulated symbol is $\exp(j\phi + j2\pi m/M)$, where ϕ is the phase offset in radians, m is the integer input such that $0 \le m \le \mathrm{M} - 1$, and M is the modulation order.
- When **Constellation ordering** is `User-defined`, specify a vector of size M, which has unique integer values in the range [0, M–1]. The first element of this vector corresponds to the constellation point having an value of $e^{j\phi}$ with subsequent elements running counterclockwise.

Example: `[0 3 2 1]`

`Constellation mapping` — User-defined symbol mapping

`[0:7]` (default) | vector

Specify the order in which input integers are mapped to output integers. The parameter is available when **Constellation ordering** is `User-defined`, and must be a row or column vector of size *M* having unique integer values in the range [0, *M* – 1].

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.

`Phase offset (rad)` — Phase offset in radians

`pi/8` (default) | scalar

Specify, in radians, the phase offset of the initial constellation as a real scalar.

Example: `pi/4`

`Output data type` — Output data type

`double` (default) | `single` | `Inherit via back propagation` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

Specify the data type of the modulated output signal. Set this parameter to one of the fixed point options or `<data type expression>` to enable parameters in which you specify additional details. Set this parameter to `Inherit via back propagation`, to match the output data type and scaling to the following block in the model.

Tips

- The M-PSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. Clicking the **View Constellation** button allows you to visualize a signal constellation for the specified block parameters.
- This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see M-PSK Modulator Baseband in the HDL Coder documentation.

Algorithms

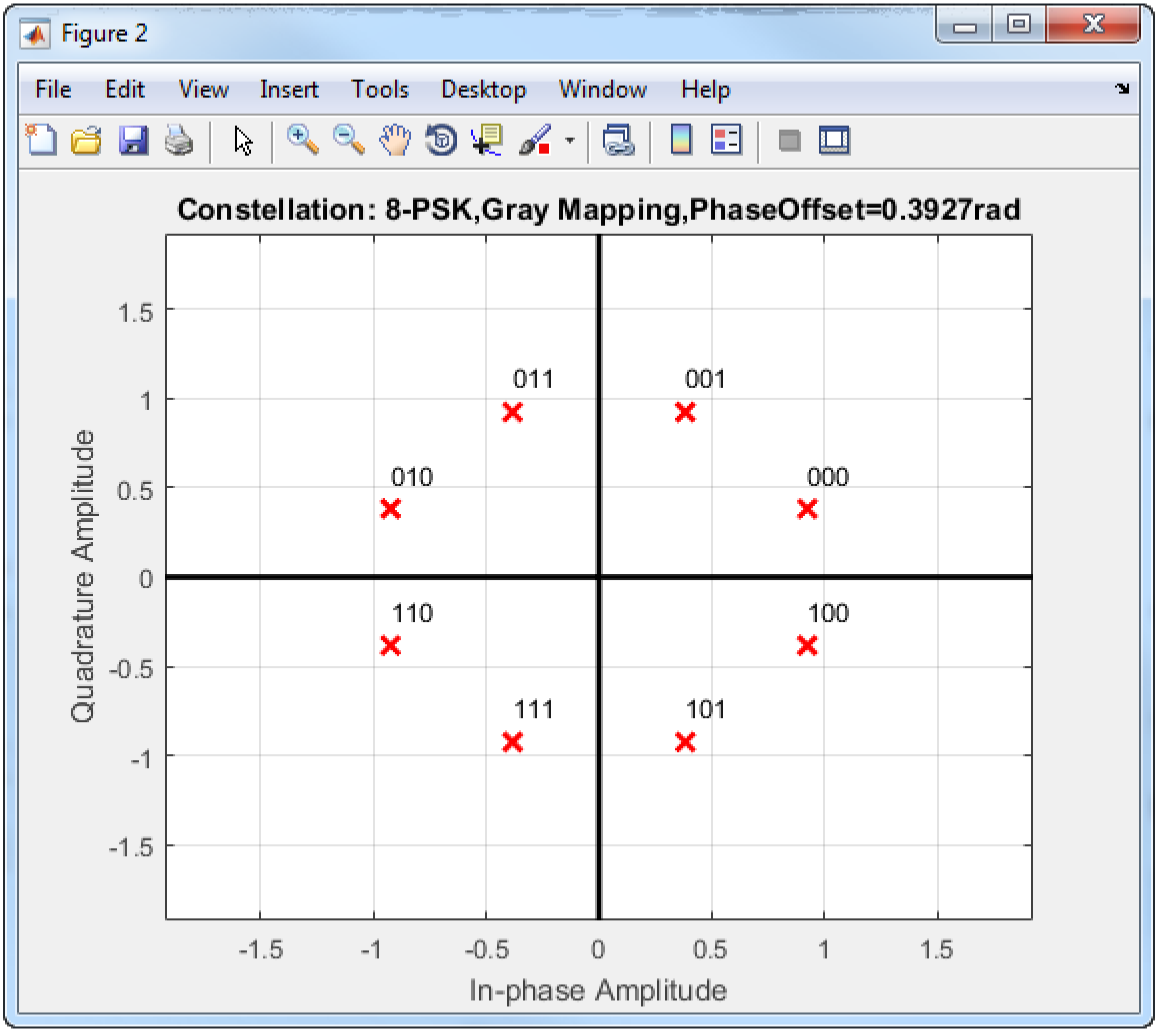
The block outputs a baseband signal by mapping input bits or integers to complex symbols according to the following:

$$s_n(t) = \exp\left(j\pi \left(\frac{2n+1}{M} \right) \right); \quad n \in \{0,1,\ldots,M-1\}.$$

This applies when a natural binary ordering is used. Another common mapping is Gray coding, which has the advantage that only one bit changes between adjacent constellation points. This results in better bit error rate performance. For 8-PSK modulation with Gray coding, the mapping between the input and output symbols is shown.

Input	Output
0	0 *(000)*
1	1 *(001)*
2	3 *(011)*
3	2 *(010)*
4	6 *(110)*
5	7 *(111)*
6	5 *(101)*
7	4 *(100)*

The corresponding constellation diagram follows.



When the input signal is composed of bits, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $\log_2(M)$ bits.

See Also

M-DPSK Modulator Baseband | M-PSK Demodulator Baseband

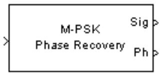
Topics

"Phase Modulation"

Introduced before R2006a

M-PSK Phase Recovery

Recover carrier phase using M-Power method



Note M-PSK Phase Recovery will be removed in a future release. Use the Carrier Synchronizer block instead.

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.

Parameters

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 clear the **Treat each discrete rate as a separate task** checkbox.

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.

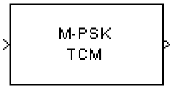
See Also

CPM Phase Recovery, M-PSK Modulator Baseband

Introduced before R2006a

M-PSK TCM Decoder

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 2-665.

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**.

Parameters

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

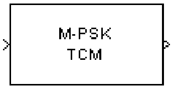
[1] Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

Introduced before R2006a

M-PSK TCM Encoder

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”. 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

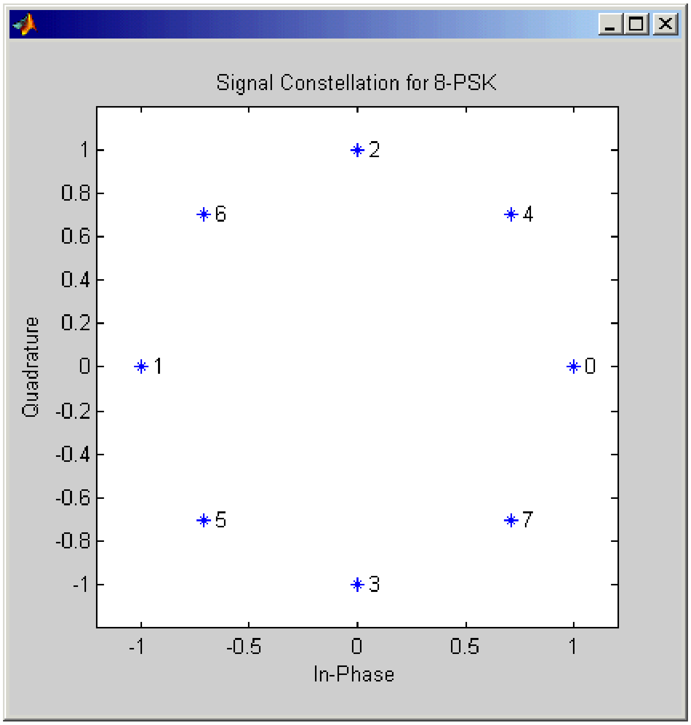
  ```
  poly2trellis(7,[171 133],171)
  ```
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink software to spend less time updating the diagram at the beginning of 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].

Parameters

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.

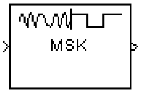
[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Introduced before R2006a

MSK Demodulator Baseband

Demodulate differentially encoded 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 differentially encoded minimum shift keying method. The block expects the input signal to be a baseband representation of a coherent modulated signal with no precoding. The **Phase offset** parameter represents the initial phase of the modulated waveform.

Pulse Shape Filtering

Differentially encoded minimum shift keying 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 this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to this rectangular pulse shape expression, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t)=\begin{cases}\frac{1}{2LT}, & 0\le t\le LT\\ 0 & \text{otherwise}\end{cases}$

- L_{main} is the main lobe pulse duration in symbol intervals.
- The duration of the pulse, *LT*, is the pulse length in symbol intervals.

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×log2(*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

Parameters

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".

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.

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

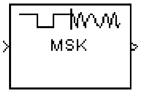
References

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

Introduced before R2006a

MSK Modulator Baseband

Modulate using differentially encoded minimum shift keying method



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The MSK Modulator Baseband block modulates using the differentially encoded 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.

Pulse Shape Filtering

Differentially encoded minimum shift keying 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 this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to this rectangular pulse shape expression, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t) = \begin{cases} \frac{1}{2LT}, & 0 \leq t \leq LT \\ 0 & \text{otherwise} \end{cases}$

- L_{main} is the main lobe pulse duration in symbol intervals.
- The duration of the pulse, *LT*, is the pulse length in symbol intervals.

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 2-680 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.

Parameters

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.

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.

Introduced before R2006a

MSK-Type Signal Timing Recovery

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.

Parameters

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 (Simulink).

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

$$\begin{aligned} e(k) = (-1)^{D+1} \operatorname{Re}\{r^2(kT - T_s + d_{k-1}) r^{*2}((k-1)T - T_s + d_{k-2})\} \\ -(-1)^{D+1} \operatorname{Re}\{r^2(kT + T_s + d_{k-1}) r^{*2}((k-1)T + T_s + d_{k-1})\} \end{aligned}$$

$$\begin{aligned} e(k) = (-1)^{D+1} \operatorname{Re}\{r^2(kT - T_s + d_{k-1}) r^{*2}((k-1)T - T_s + d_{k-2})\} \\ -(-1)^{D+1} \operatorname{Re}\{r^2(kT + T_s + d_{k-1}) r^{*2}((k-1)T + T_s + d_{k-1})\} \end{aligned}$$

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

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

Introduced before R2006a

Mueller-Muller Timing Recovery

Recover symbol timing phase using Mueller-Muller method

Note Mueller-Muller Timing Recovery has been removed. Use the Symbol Synchronizer block instead.

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.

Parameters

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 (Simulink) 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

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

Symbol Synchronizer

Introduced before R2006a

Mu-Law Compressor

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.

Parameters

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

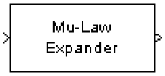
References

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

Introduced before R2006a

Mu-Law Expander

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.

$$x = \frac{V}{\mu}\left(e^{|y|\log(1+\mu)/V} - 1\right)\operatorname{sgn}(y)$$

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

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

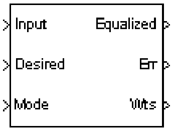
References

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

Introduced before R2006a

Normalized LMS Decision Feedback Equalizer

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.

Parameters

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.

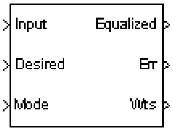
See Also

Normalized LMS Linear Equalizer, LMS Decision Feedback Equalizer

Introduced before R2006a

Normalized LMS Linear Equalizer

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.

Parameters

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.

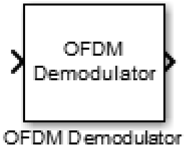
See Also

Normalized LMS Decision Feedback Equalizer, LMS Linear Equalizer

Introduced before R2006a

OFDM Demodulator Baseband

Demodulate orthogonal frequency division modulated data



Library

OFDM, in Digital Baseband sublibrary of Modulation

Description

The Orthogonal Frequency Division Modulation (OFDM) Demodulator Baseband block demodulates an OFDM input signal. The block accepts a single input and has one or two output ports, depending on the status of **Pilot output port**.

Signal Dimensions

Pilot Output Port	Pilot Carrier Indices	Signal Input	Signal Output	Pilot Output
false	N/A	$N_{CPTotal}+N_{FFT}\times N_{sym}$-by-$N_r$	N_{data}-by-N_{sym}-by-N_r	N/A
true	2-D			N_{pilot}-by-N_{sym}-by-N_r
	3-D			N_{pilot}-by-N_{sym}-by-N_t-by-N_r

where

- N_{CP} represents the cyclic prefix length as determined by **Cyclic prefix length**.

- $N_{CPTotal}$ represents the cyclic prefix length over all the symbols. When N_{CP} is a scalar, $N_{CPTotal} = N_{CP} \times N_{sym}$. When N_{CP} is a row vector, $N_{CPTotal} = \sum N_{CP}$.
- N_{FFT} represents the number of subcarriers as determined by **FFT length**.
- N_{sym} represents the number of symbols as determined by **Number of OFDM symbols**.
- N_r represents the number of receive antennas as determined by **Number of receive antennas**.
- N_{data} represents the number of data subcarriers. For further information on how N_{data} is determined, see the `comm.OFDMDemodulator.info` reference page.
- N_{pilot} represents the number of pilot symbols determined by the second dimension in the **Pilot subcarrier indices** array.
- N_t represents the number of transmit antennas. This parameter is derived from the third dimension of the **Pilot subcarrier indices** array.

Parameters

FFT Length

Specify the FFT length, which is equivalent to the number of subcarriers. The length of the FFT, N_{FFT}, must be greater than or equal to 8.

Number of guard bands

Assign the number of subcarriers to the left, N_{leftG}, and right, N_{rightG}, guard bands. The input is a 2-by-1 vector. The number of subcarriers must fall within $[0,N_{FFT}/2 - 1]$.

Remove DC carrier

Select to remove the DC subcarrier.

Pilot output port

Select to separate the data from the pilot signal and output the demodulated pilot signal.

Pilot subcarrier indices

Specify the pilot subcarrier indices. This field is available only when the **Pilot output port** check box is selected. You can assign the indices can be assigned to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the indices' array vary from 1 to 3. Valid pilot indices fall in the range

$$[N_{leftG}+1, N_{FFT}/2] \cup \left[N_{FFT}/2+2, N_{FFT}-N_{rightG}\right],$$

where the index value cannot exceed the number of subcarriers. If the number of transmit antennas is greater than one, ensure that the indices per symbol are mutually distinct across antennas to minimize interference.

Cyclic prefix length

Specify the length of the cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same length through all antennas.

Number of OFDM symbols

Specify the number of OFDM symbols, N_{sym}, in the time-frequency grid.

Number of receive antennas

Specify the number of receive antennas, N_r, as a positive integer such that $N_r \leq 64$.

Simulate using

Select the simulation type from these choices:

- `Code generation`
- `Interpreted execution`

Algorithms

This block implements the algorithm, inputs, and outputs described in the `OFDM Demodulator` System object reference page. The object properties correspond to the block parameters.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
Pilot (optional)	• Double-precision floating point

Port	Supported Data Types
Output	- Double-precision floating point

Selected Bibliography

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX*.Upper Saddle River, NJ: Prentice Hall, 2007.

Pair Block

OFDM Modulator Baseband

See Also

QPSK Demodulator Baseband | Rectangular QAM Demodulator Baseband | `comm.OFDMDemodulator`

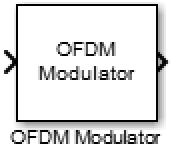
Topics

"IEEE 802.16-2009 WirelessMAN-OFDMA PHY Downlink PUSC"
"Digital Video Broadcasting - Terrestrial"

Introduced in R2014a

OFDM Modulator Baseband

Modulate using orthogonal frequency division modulation



Library

OFDM, in Digital Baseband sublibrary of Modulation

Description

The OFDM Modulator Baseband block applies OFDM modulation to an incoming data signal. The block accepts one or two inputs depending on the state of the **Pilot input port**.

Signal Dimensions

Pilot Input Port	Signal Input	Pilot Input	Signal Output
false	N_{data}-by-N_{sym}-by-N_t	N/A	$N_{CPTotal}+N_{FFT}\times N_{sym}$-by-$N_t$
true		N_{pilot}-by-N_{sym}-by-N_t	

where

- N_{data} represents the number of data subcarriers. For further information on how N_{data} is determined, see the `comm.OFDMModulator.info` reference page.
- N_{sym} represents the number of symbols determined by **Number of OFDM symbols**.
- N_t represents the number of transmit antennas determined by **Number of transmit antennas**.

- N_{pilot} represents the number of pilot symbols determined by the first dimension size in the **Pilot subcarrier indices** array.
- N_{CP} represents the cyclic prefix length as determined by **Cyclic prefix length**.
- $N_{CPTotal}$ represents the cyclic prefix length over all the symbols. When N_{CP} is a scalar, $N_{CPTotal} = N_{CP} \times N_{sym}$. When N_{CP} is a row vector, $N_{CPTotal} = \sum N_{CP}$.
- N_{FFT} represents the number of subcarriers as determined by **FFT length**.

Parameters

FFT Length

Specify the FFT length, which is equivalent to the number of subcarriers. The length of the FFT, N_{FFT}, must be greater than or equal to 8.

Number of guard bands

Assign the number of subcarriers to the left and right guard bands. The input is a 2-by-1 vector. The number of subcarriers must fall within $[0,N_{FFT}/2 - 1]$.

Insert DC null

Select to insert a null on the DC subcarrier.

Pilot input port

Select to allow the specifying of pilot subcarrier indices.

Pilot subcarrier indices

Specify the pilot subcarrier indices. This field is available only when the **Pilot input port** check box is selected. You can assign the indices to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the indices array vary. Valid pilot indices fall in the range

$$\left[N_{leftG}+1, N_{FFT}/2\right] \cup \left[N_{FFT}/2+2, N_{FFT}-N_{rightG}\right],$$

where the index value cannot exceed the number of subcarriers. When the pilot indices are the same for every symbol and transmit antenna, the property has dimensions N_{pilot}-by-1. When the pilot indices vary across symbols, the property has dimensions of N_{pilot}-by-N_{sym}. If there is only one symbol but multiple transmit antennas, the property has dimensions of N_{pilot}-by-1-by-N_t. If the indices vary across the number of symbols and transmit antennas, the property will have dimensions of N_{pilot}-by-N_{sym}-by-N_t. If the number of transmit antennas is greater than one, ensure that the indices per symbol are mutually distinct across antennas to minimize interference. The default value is `[12; 26; 40; 54]`.

Cyclic prefix length

Specify the length of the cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same through all antennas.

Apply raised cosine windowing between OFDM symbols

Select to apply raised cosine windowing. Windowing is the process in which the OFDM symbol is multiplied by a raised cosine window before transmission to reduce the power of out-of-band subcarriers, which serves to reduce spectral regrowth.

Window length

Set the length of the raised cosine window. The field is available only when **Apply raised cosine windowing between OFDM symbols** is selected. Use positive integers having a maximum value no greater than the minimum cyclic prefix length. For example, in a configuration in which there are four symbols with cyclic prefix lengths of `[12 16 14 18]`, the window length cannot exceed 12.

Number of OFDM symbols

Specify the number of OFDM symbols in the time-frequency grid.

Number of transmit antennas

Specify the number of transmit antennas, N_t, as a positive integer such that $N_t \leq 64$.

Simulate using

Select the simulation type from these choices:

- `Code generation`
- `Interpreted execution`

Algorithms

This block implements the algorithm, inputs, and outputs described in the `OFDM Modulator` System object reference page. The object properties correspond to the block parameters.

Supported Data Types

Port	Supported Data Types
Input	- Double-precision floating point
Pilot (optional)	- Double-precision floating point
Output	- Double-precision floating point

Selected Bibliography

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX*.Upper Saddle River, NJ: Prentice Hall, 2007.

Pair Block

OFDM Demodulator Baseband

See Also

QPSK Modulator Baseband | Rectangular QAM Modulator Baseband | `comm.OFDMModulator`

Topics

"IEEE 802.16-2009 WirelessMAN-OFDMA PHY Downlink PUSC"
"Digital Video Broadcasting - Terrestrial"

Introduced in R2014a

OQPSK Demodulator Baseband

Demodulation using OQPSK method

Library: Communications System Toolbox / Modulation / Digital Baseband / PM

Description

The OQPSK Demodulator Baseband block applies pulse shape filtering to the input waveform and demodulates it using the offset quadrature phase shift keying (OQPSK) 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 delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 2-713.

Ports

Input

`Port_1 ( In1 )` — Input baseband waveform
column vector

Input baseband waveform, specified as a column vector.

The block processes the input signal based on the Output type setting.

Data Types: `double`
Complex Number Support: Yes

Output

`Port_1 ( Out1 )` — Output data

integer column vector | bit column vector

Output data, returned as an integer or bit column vector.

Parameters

Modulation

`Output type` — Output type

`Integer` (default) | `Bit`

Output type, specified as `Integer` or `Bit`.

- When you set **Output type** to `Integer`, the block outputs a vector of integer symbols with values from 0 to 3, the length of which is the number of output symbols.
- When you set **Output type** to `Bit`, the block outputs a 2-bit binary representation of integers, in a binary-valued, even-length vector.

The input period for each integer or bit pair is the Samples per symbol times the output sample period.

`Phase offset (rad)` — Phase of zeroth point of constellation shifted from π/4

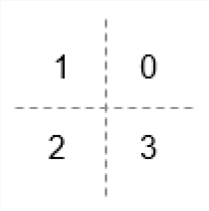
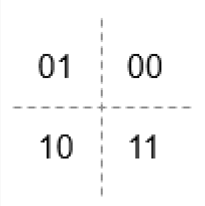
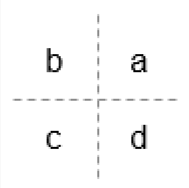
`0` (default) | scalar

Phase of zeroth point of constellation shifted from π/4 radians, specified as a scalar.

`Symbol Mapping` — Signal constellation bit mapping

`Gray` (default) | `Binary` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `Gray`, `Binary`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping	Comment
`Gray`		The signal constellation mapping is a Gray-encoded integer.
`Binary`		The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4)\ +\ j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3		Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Filtering

`Pulse shape` — Filtering pulse shape

`Half sine` (default) | `Normal raised cosine` | `Root raised cosine` | `Custom`

Select the filtering pulse shape: `Half sine`, `Normal raised cosine`, `Root raised cosine`, or `Custom`.

`Rolloff factor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property appears when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter span (in symbols)` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to **Filter span (in symbols)** symbols.

Dependencies

This property appears when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter numerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This parameter appears when Pulse shape is `Custom`.

Data Types: `double`

`Samples per symbol` — Number of samples per symbol

`4` (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

Other Parameters

`Rate options` — Processing rate option

`Enforce single-rate processing` (default) | `Allow multirate processing`

- `Enforce single-rate processing` — Executes the model, ensuring that 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. For integer outputs, the output width equals 1/Samples per symbol times the input width.

For more information, see Single-Rate Processing with OQPSK Demodulator Block.

- `Allow multirate processing` — Executes the model, allowing the input and output signals to have different port sample times. The output symbol time is Samples per symbol times the input sample time.

  For more information, see Multirate Processing with OQPSK Demodulator Block.

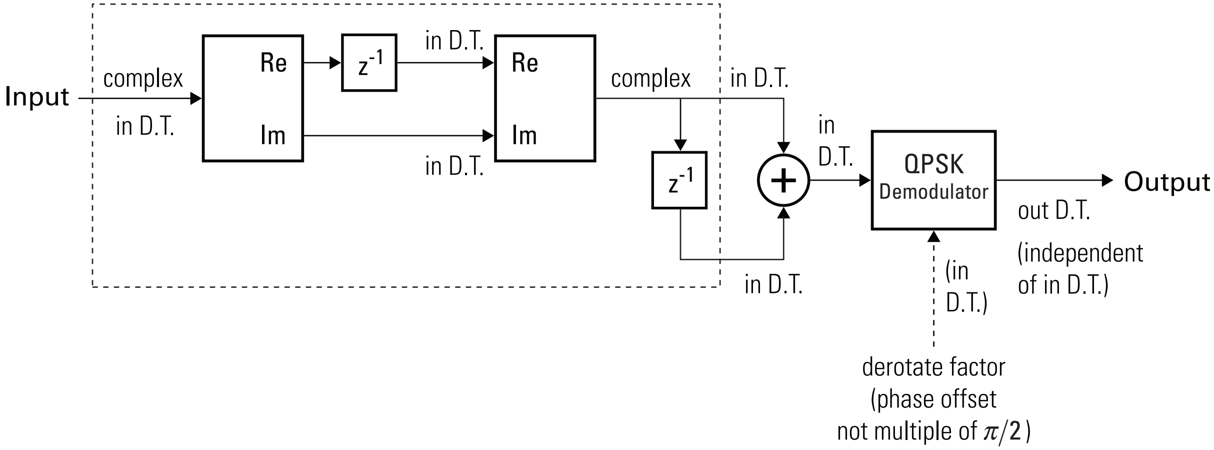
`Output data type` — Output data type
`double` (default) | `single` | `uint8`

Select the output data type: `double`, `single`, or `uint8`.

Definitions

OQPSK Signal Flow Diagram

Every Samples per symbol input samples produce one output symbol. In this figure, the dotted line represents the region comprising the input sample processing.



Modulation Delays

Digital modulation and demodulation blocks incur delays between their inputs and outputs that result in an offset in the arrival time of the received data. Data that enters a modulation or demodulation block at time *T* appears in the output at time *T*+delay. Take system delays into account when comparing transmitted data with received data, such as in overlaid plots or when computing error statistics. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter, input/output data setting, and simulation configuration.

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
`Half sine` or `Custom`	`Enforce single-rate operation`	N/A	`Integer`	1
			`Bit`	2
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + 1 + 1
			`Bit`	length(data) + 2 + 2
		`true` (multitasking)	`Integer`	length(data) + 1 + 2
			`Bit`	length(data) + 2 + 4
`Normal raised cosine` or `Root raised cosine`	`Enforce single-rate operation`	N/A	`Integer`	**Filter span (in symbols)**
			`Bit`	2***Filter span (in symbols)**
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + **Filter span (in symbols)** + 1
			`Bit`	length(data) + 2***Filter span (in symbols)** + 2

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
		`true` (multitasking)	`Integer`	2*length(data) + **Filter span (in symbols)** + 2
			`Bit`	2*length(data) + 2***Filter span (in symbols)** + 4
(*) The data type parameter is **Input type** for modulation and **Output type** for demodulation.				

See Also

Blocks

OQPSK Modulator Baseband | QPSK Demodulator Baseband

System Objects

`comm.OQPSKDemodulator`

Topics

Phase Modulation

Introduced before R2006a

OQPSK Modulator Baseband

Modulation using OQPSK method

Library: Communications System Toolbox / Modulation / Digital Baseband / PM

Description

The OQPSK Modulator Baseband block modulates the input signal using the offset quadrature phase shift keying (OQPSK) method and applies pulse shape filtering to the waveform. The output is a baseband representation of the modulated signal.

For information about delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 2-720.

Ports

Input

`Port_1 ( In1 )` — Input data

integer column vector | bit column vector

Input data, specified as an integer or bit column vector.

The input signal is processed based on the setting selected for Input type.

Data Types: `double`

Output

`Port_1 ( Out1 )` — Output baseband waveform

column vector

Output baseband waveform, returned as a column vector of complex data.

Parameters

Modulation

Input type — Input type

Integer (default) | Bit

Input type, specified as Integer or Bit.

- 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 the period of each integer or bit pair in the input divided by Samples per symbol.

Phase offset (rad) — Phase of zeroth point of constellation shifted from π/4

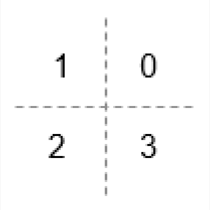
0 (default) | scalar

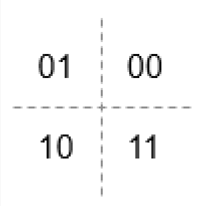
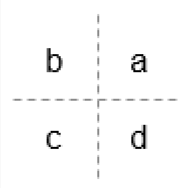
Phase of zeroth point of constellation shifted from π/4 radians, specified as a scalar.

Symbol Mapping — Signal constellation bit mapping

Gray (default) | Binary | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as Gray, Binary, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping	Comment
Gray		The signal constellation mapping is a Gray-encoded integer.

Setting	Constellation Mapping	Comment
`Binary`		The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3		Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Filtering

`Pulse shape` — Filtering pulse shape
`Half sine` (default) | `Normal raised cosine` | `Root raised cosine` | `Custom`

Select the filtering pulse shape: `Half sine`, `Normal raised cosine`, `Root raised cosine`, or `Custom`.

`Rolloff factor` — Raised cosine filter rolloff factor
`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property appears when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter span (in symbols)` — Filter length
`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to **Filter span (in symbols)** symbols.

Dependencies

This property appears when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter numerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This parameter appears when Pulse shape is `Custom`.

Data Types: `double`

`Samples per symbol` — Number of samples per symbol

4 (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

Other Parameters

`Rate options` — Processing rate option

`Enforce single-rate processing` (default) | `Allow multirate processing`

- `Enforce single-rate processing` — Executes the model, ensuring that 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. For integer inputs, the output width equals Samples per symbol times the number of symbols.

  For more information, see Single-Rate Processing with OQPSK Modulator Block.
- `Allow multirate processing` — Executes the model, allowing the input and output signals to have different port sample times. The output sample time equals the symbol period divided by Samples per symbol.

For more information, see Single-Rate Processing with OQPSK Modulator Block.

Output data type — Output data type
double (default) | single

Select the output data type: double or single.

Definitions

Modulation Delays

Digital modulation and demodulation blocks incur delays between their inputs and outputs that result in an offset in the arrival time of the received data. Data that enters a modulation or demodulation block at time *T* appears in the output at time *T*+delay. Take system delays into account when comparing transmitted data with received data, such as in overlaid plots or when computing error statistics. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter, input/output data setting, and simulation configuration.

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
Half sine or Custom	Enforce single-rate operation	N/A	Integer	1
			Bit	2
	Allow multirate processing	false (single tasking)	Integer	length(data) + 1 + 1
			Bit	length(data) + 2 + 2
		true (multitasking)	Integer	length(data) + 1 + 2
			Bit	length(data) + 2 + 4

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
`Normal raised cosine` or `Root raised cosine`	`Enforce single-rate operation`	N/A	`Integer`	**Filter span (in symbols)**
			`Bit`	2***Filter span (in symbols)**
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + **Filter span (in symbols)** + 1
			`Bit`	length(data) + 2***Filter span (in symbols)** + 2
		`true` (multitasking)	`Integer`	2*length(data) + **Filter span (in symbols)** + 2
			`Bit`	2*length(data) + 2***Filter span (in symbols)** + 4
(*) The data type parameter is **Input type** for modulation and **Output type** for demodulation.				

See Also

Blocks
OQPSK Demodulator Baseband | QPSK Modulator Baseband

System Objects
`comm.OQPSKModulator`

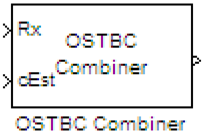
Topics

Phase Modulation

Introduced before R2006a

OSTBC Combiner

Combine inputs for received signals and channel estimate according to orthogonal space-time block code (OSTBC)



Library

MIMO

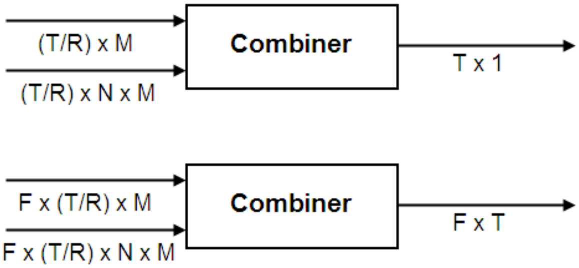
Description

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 on page 2-725 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 2-731 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	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\Vert H\Vert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} \end{pmatrix}.$
3	1/2	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\Vert H\Vert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j} r_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
3	3/4	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \\ \hat{s}_3 \end{pmatrix} = \frac{1}{\Vert H\Vert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$
4	1/2	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\Vert H\Vert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j3,j} r_{3,j} + h_{4,j} r^*_{4,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
4	3/4	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \\ \hat{s}_3 \end{pmatrix} = \frac{1}{\Vert H\Vert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} - h^*_{4,j} r_{4j} \\ h^*_{2,j} r_{1j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1j} + h^*_{4,j} r_{2,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$

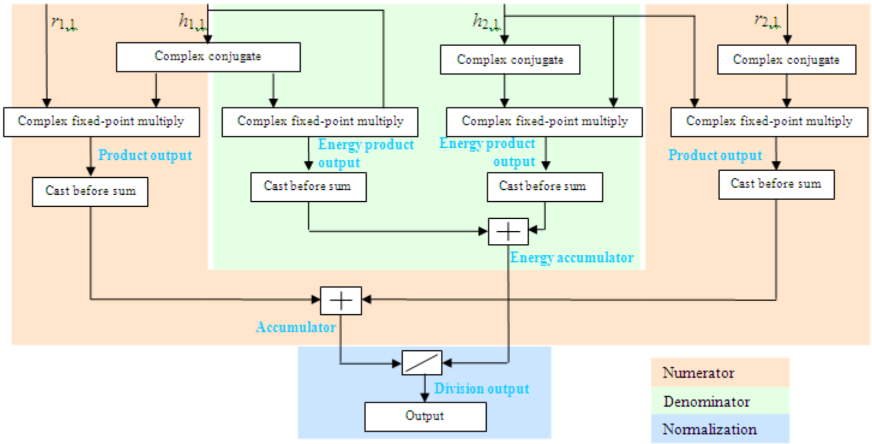
$\hat{s}_k$ represents the estimated *k*th symbol in the OSTBC codeword matrix. h_{ij} represents the estimate for the channel from the *i*th transmit antenna and the *j*th receive antenna. The values of *i* and *j* can range from 1 to *N* (the number of transmit antennas) and to *M* (the number of receive antennas) respectively. r_{lj} represents the *l*th symbol at the *j*th receive antenna per codeword block. The value of *l* can range from 1 to the codeword block length. $\Vert H\Vert^2$ represents the summation of channel power per link, i.e., $\Vert H\Vert^2 = \sum_{i=1}^{N} \sum_{j=1}^{M} \Vert h_{ij}\Vert^2$

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.

$$\hat{s}_1 = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,} r^*_{2,1}}{\|H\|^2} = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,} r^*_{2,1}}{h_{1,1} h^*_{1,1,} + h_{2,1,} h^*_{2,1}}$$

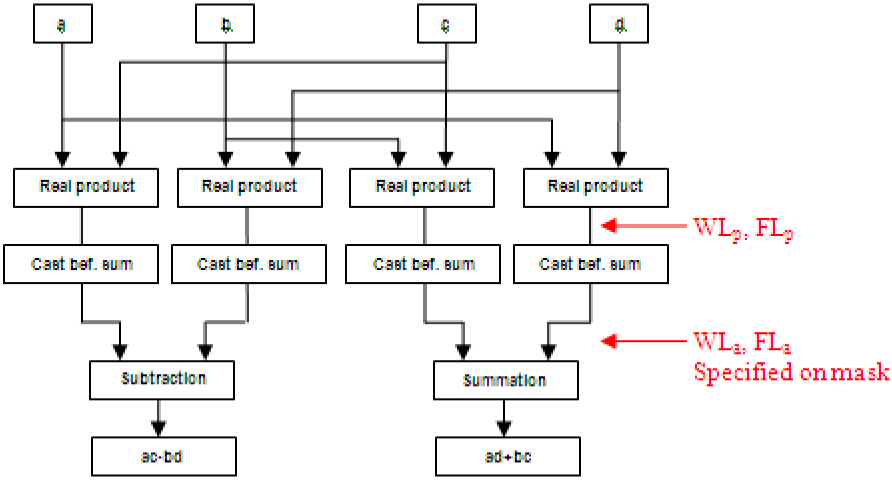
In this equation, the data types for **Product output** and **Accumulator** correspond to the product and summation in the numerator. Similarly, the types for **Energy product output** and **Energy accumulator** correspond to the product and summation in the denominator.



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 cannot 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 (DSP System Toolbox) 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 (DSP System Toolbox).

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 (Fixed-Point Designer).

Saturate on integer overflow

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 (DSP System Toolbox).

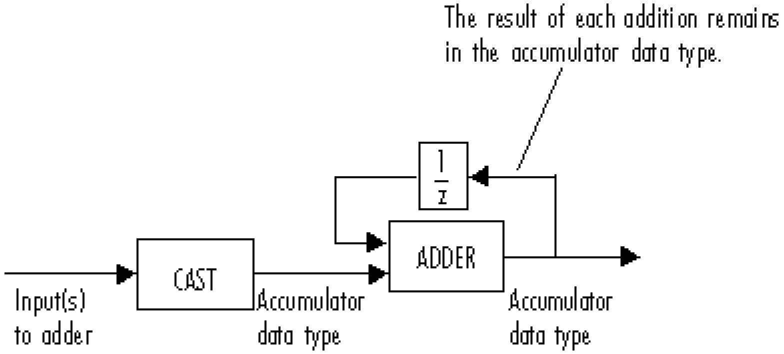
Product Output

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 (DSP System Toolbox) 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. 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` at the MATLAB command line.

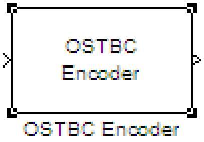
See Also

OSTBC Encoder

Introduced in R2009a

OSTBC Encoder

Encode input message using orthogonal space-time block code (OSTBC)



Library

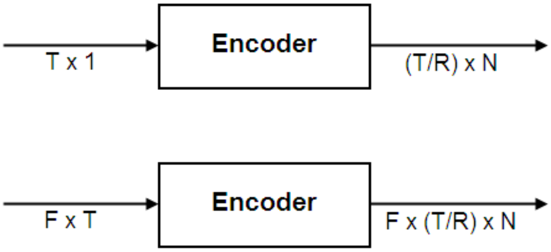
MIMO

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 on page 2-734 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.
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 2-736 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 **Saturate on integer overflow** 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	$\begin{pmatrix} s_1 & s_2 & s_3 \\ -s_2^* & s_1^* & 0 \\ s_3^* & 0 & -s_1^* \\ 0 & s_3^* & -s_2^* \end{pmatrix}$
4	1/2	$\begin{pmatrix} s_1 & s_2 & 0 & 0 \\ -s_2^* & s_1^* & 0 & 0 \\ 0 & 0 & s_1 & s_2 \\ 0 & 0 & -s_2^* & s_1^* \end{pmatrix}$

Transmit Antenna	Rate	OSTBC Codeword Matrix
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 *i*th antenna in the *l*th 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.

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.

Saturate on integer overflow

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” (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” . 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` at the MATLAB command line.

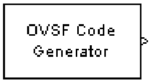
See Also

OSTBC Combiner

Introduced in R2009a

OVSF Code Generator

Generate orthogonal variable spreading factor (OVSF) code from set of orthogonal codes



Library

Spreading Codes

Description

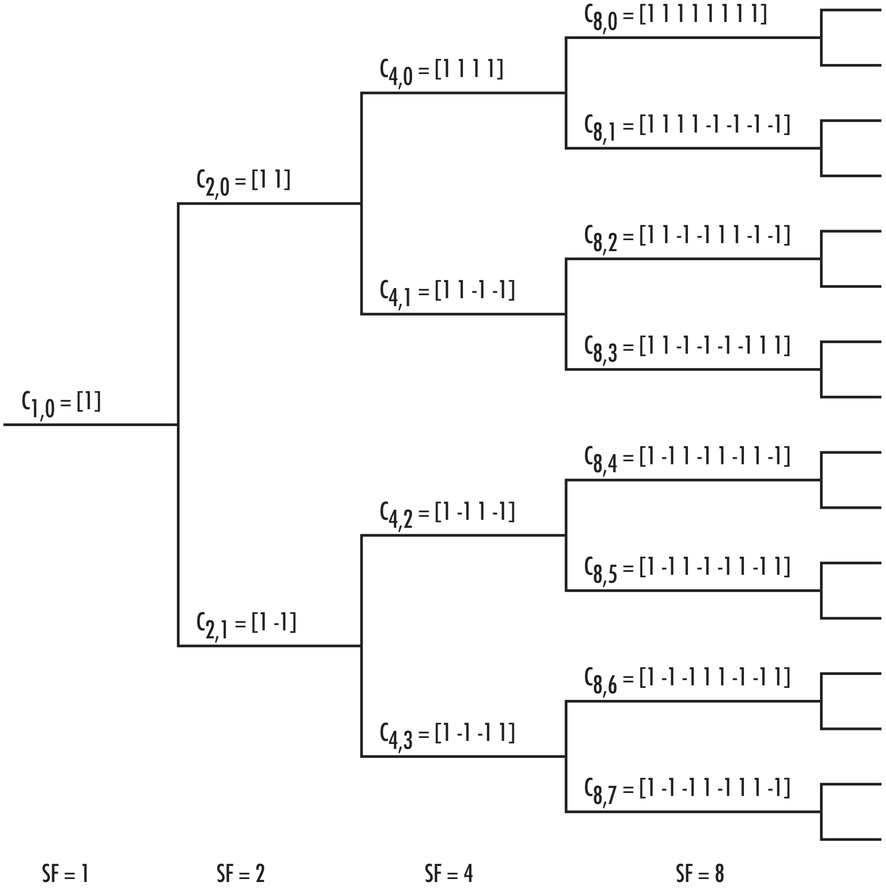
The OVSF Code Generator block generates an OVSF code from a set of orthogonal codes. OVSF codes were first introduced for 3G communication systems. OVSF codes are primarily used to preserve orthogonality between different channels in a communication system.

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

$$C_{2N} = \begin{bmatrix} C_N(0) & C_N(0) \\ C_N(0) & -C_N(0) \\ C_N(1) & C_N(1) \\ C_N(1) & -C_N(1) \\ \dots & \dots \\ C_N(N-1) & C_N(N-1) \\ C_N(N-1) & -C_N(N-1) \end{bmatrix}$$

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.


$C_{8,0}$ = [1 1 1 1 1 1 1 1]
$C_{4,0}$ = [1 1 1 1]
$C_{8,1}$ = [1 1 1 1 -1 -1 -1 -1]
$C_{2,0}$ = [1 1]
$C_{8,2}$ = [1 1 -1 -1 1 1 -1 -1]
$C_{4,1}$ = [1 1 -1 -1]
$C_{8,3}$ = [1 1 -1 -1 -1 -1 1 1]
$C_{1,0}$ = [1]
$C_{8,4}$ = [1 -1 1 -1 1 -1 1 -1]
$C_{4,2}$ = [1 -1 1 -1]
$C_{8,5}$ = [1 -1 1 -1 -1 1 -1 1]
$C_{2,1}$ = [1 -1]
$C_{8,6}$ = [1 -1 -1 1 1 -1 -1 1]
$C_{4,3}$ = [1 -1 -1 1]
$C_{8,7}$ = [1 -1 -1 1 -1 1 1 -1]
SF = 1
SF = 2
SF = 4
SF = 8

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 $x_1\ x_2\ ...\ x_r$ 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]

i	x_i	C_i
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.

Parameters

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

The time between each sample of the output signal. Specify as a nonnegative real scalar.

Samples per frame

The number of samples per frame in one column of the output signal. Specify as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

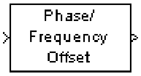
See Also

Hadamard Code Generator, Walsh Code Generator

Introduced before R2006a

Phase/Frequency Offset

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

φ(t) = Phase offset

The discrete-time output is:

$$y(0) = u(0)\big(\cos(\varphi(0)) + j\sin(\varphi(0))\big)$$

$$y(i) = u(i)\left(\cos\left(2\pi \sum_{n=0}^{i-1} f(n)\Delta t + \varphi(i) \right) + j \sin\left(2\pi \sum_{n=0}^{i-1} f(n)\Delta t + \varphi(i) \right) \right) \qquad i > 0$$

where

Δt = Sample time

This block accepts real and complex inputs of data type `double` or `single`.

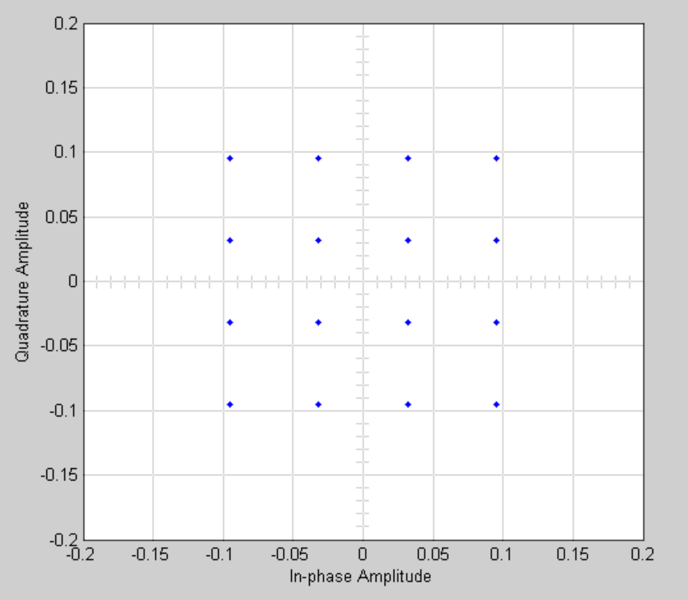
Phase Offset

The block applies a phase offset to the input signal, specified by the **Phase offset** parameter.

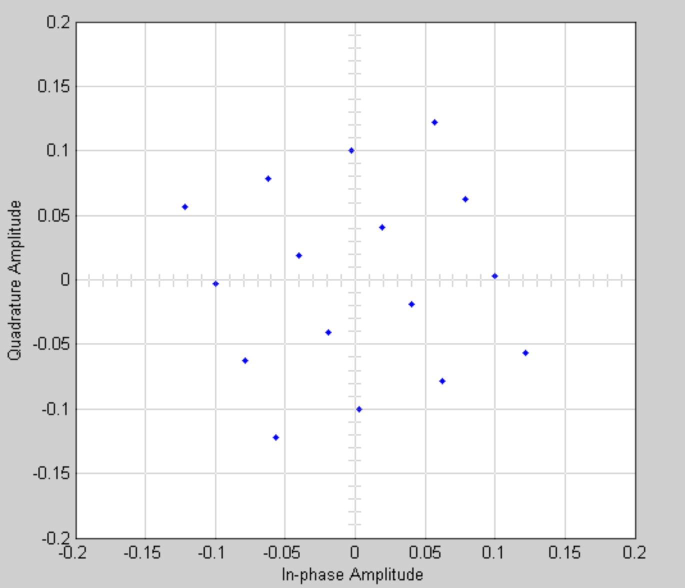
Frequency Offset

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 2-746.

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:

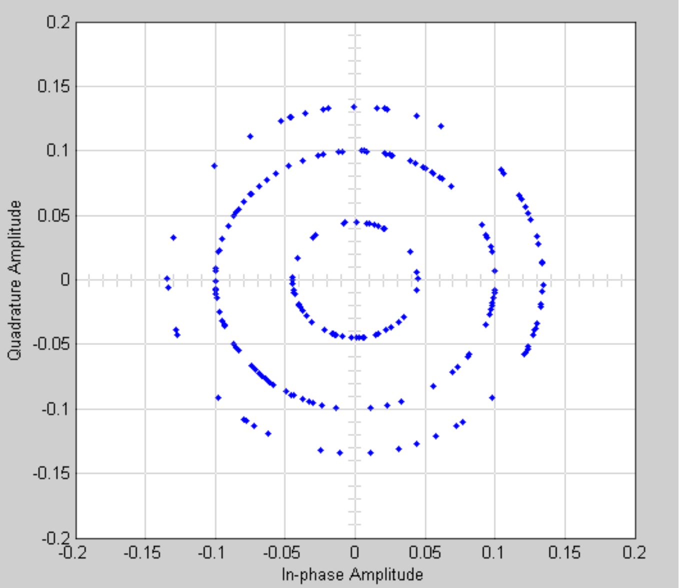


The following figure shows a scatter plot of an output signal, modulated by 16-ary QAM, from the Phase/Frequency Offset block with **Phase offset** set to `20` and **Frequency offset** set to `0`:



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
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.

Parameters

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 (Simulink) 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 (Simulink) 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

Introduced before R2006a

Phase-Locked Loop

Implement phase-locked loop to recover phase of input signal

PLL
Filt
PD
VCO

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

Parameters

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*.

Introduced before R2006a

Phase Noise

Apply receiver phase noise to complex baseband signal
Library: Communications System Toolbox / RF Impairments

Description

The Phase Noise block adds phase noise to a complex signal. This block emulates impairments introduced by the local oscillator of a wireless communication transitter or receiver. The block generates filtered phase noise according to the specified spectral mask and adds it to the input signal. For a description of the phase noise modeling, see "Algorithms" on page 2-753.

Ports

Input

`In` — Input signal
complex column vector

Input signal, specified as an N_S-by-1 vector of complex values. N_S represents the number of samples in the input signal.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Output signal
column vector

Output signal, returned as an N_S-by-1 vector of complex values. N_S equals the number of samples in the input signal.

Data Types: `double`
Complex Number Support: Yes

Parameters

`Phase noise level (dBc/Hz)` — Phase noise level

`[-60 -80]` (default) | vector of negative scalars

Phase noise level in decibels relative to carrier per hertz (dBc/Hz), specified as a vector of negative scalars. The **Phase noise level (dBc/Hz)** and **Frequency offset (Hz)** parameters must have the same length.

Tunable: Yes

`Frequency offset (Hz)` — Frequency offset

`[20 200]` (default) | vector of positive increasing values

Frequency offset in Hz, specified as a vector of positive increasing values. The **Phase noise level (dBc/Hz)** and **Frequency offset (Hz)** parameters must have the same length.

Tunable: Yes

Data Types: `double`

`Sample rate (Hz)` — Sample rate

`1024` (default) | positive scalar

Sample rate in samples per second, specified as a positive scalar. To avoid aliasing, the sample rate must be greater than twice the largest value specified by **Frequency offset (Hz)**.

Tunable: Yes

Data Types: `double`

`Initial seed` — Initial seed of noise generator

`2137` (default) | positive scalar

Initial seed of noise generator, specified as a positive scalar.

This block uses the Random Source block to generate noise. The block generates random numbers using the Ziggurat method (V5 RANDN algorithm). Every time you rerun the simulation, the block reuses the same initial seed. That way, the block outputs the same signal each time you run a simulation.

Tunable: Yes

Data Types: `double`

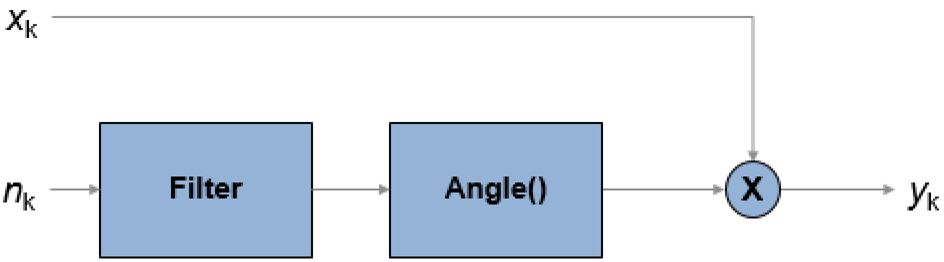
`View Filter Response` — Display magnitude response of filter

button

Display magnitude response of filter defined by the Phase Noise block. The block uses the `fvtool` function to display the magnitude response.

Algorithms

The output signal, y_k, is related to input sequence x_k by $y_k = x_k e^{j\varphi_k}$, where φ_k is the phase noise. The phase noise is filtered Gaussian noise such that $\varphi_k = f(n_k)$, where n_k is the noise sequence and *f* represents a filtering operation.



To model the phase noise, define the power spectrum density (PSD) mask characteristic by specifying scalar or vector values for the frequency offset and phase noise level.

- For a scalar frequency offset and phase noise level specification, an IIR digital filter computes the spectrum mask. The spectrum mask has a 1/*f* characteristic that passes through the specified point.
- For a vector frequency offset and phase noise level specification, an FIR filter computes the spectrum mask. The spectrum mask is interpolated across log10(*f*). It is

flat from DC to the lowest frequency offset, and from the highest frequency offset to half the sample rate.

IIR Digital Filter

For the IIR digital filter, the numerator coefficient is

$$\lambda = \sqrt{2\pi f_{offset} 10^{L/10}} \, ,$$

where f_{offset} is the frequency offset in Hz and L is the phase noise level in dBc/Hz. The denominator coefficients, γ_{i}, are recursively determined as

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1} \, ,$$

where $\gamma_1 = 1$, $i = \{1, 2,..., N_{\text{t}}\}$, and N_{t} is the number of filter coefficients. N_{t} is a power of 2, from 2^7 to 2^{19}. The value of N_{t} grows as the phase noise offset decreases towards 0 Hz.

FIR Filter

For the FIR filter, the phase noise level is determined through log10(*f*) interpolation for frequency offsets over the range [*df*, f_{s} / 2], where *df* is the frequency resolution and f_{s} is the sample rate. The phase noise is flat from 0 Hz to the smallest frequency offset, and from the largest frequency offset to f_{s} / 2. The frequency resolution is equal to $\frac{f_s}{2}\left(\frac{1}{N_t}\right)$, where N_{t} is the number of coefficients, and is a power of 2 less than or equal to 2^{16}. If $N_{\text{t}} < 2^7$, a time domain FIR filter is used. Otherwise, a frequency domain FIR filter is used.

The algorithm increases N_{t} until these conditions are met:

- The frequency resolution is less than the minimum value of the frequency offset vector.
- The frequency resolution is less than the minimum difference between two consecutive frequencies in the frequency offset vector.
- The maximum number of FIR filter taps is 2^{16}.

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*. Vol. 83, No. 5, May, 1995, pp 802–827.

See Also

Blocks
Phase/Frequency Offset

Functions
`plotPhaseNoiseFilter`

System Objects
`comm.PhaseNoise`

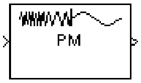
Topics

"View Phase Noise Effects on Signal Spectrum"

Introduced before R2006a

PM Demodulator Passband

Demodulate PM-modulated data



Library

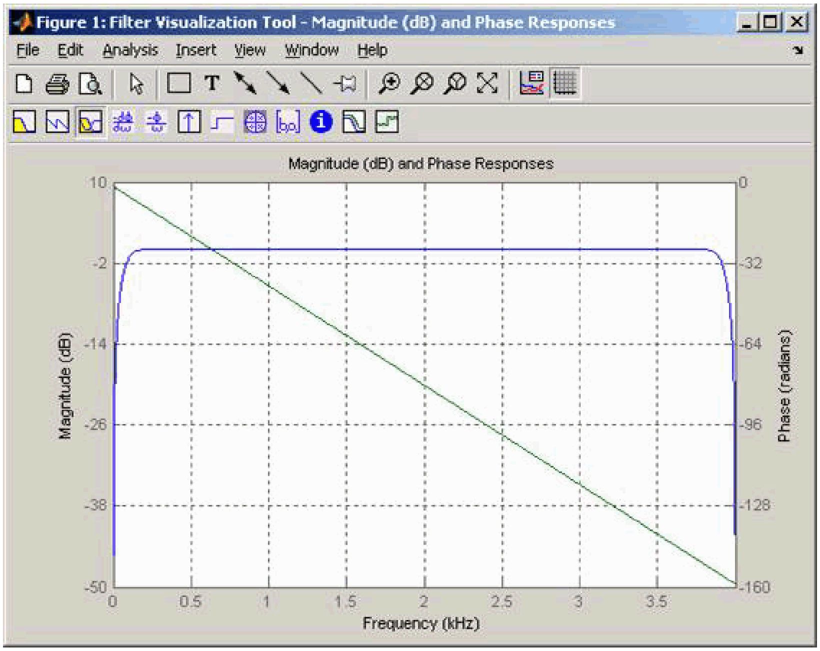
Analog Passband Modulation, in Modulation

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.

Parameters

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.

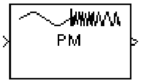
Pair Block

PM Modulator Passband

Introduced before R2006a

PM Modulator Passband

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.

Parameters

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.

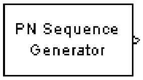
Pair Block

PM Demodulator Passband

Introduced before R2006a

PN Sequence Generator

Generate pseudonoise sequence



Library

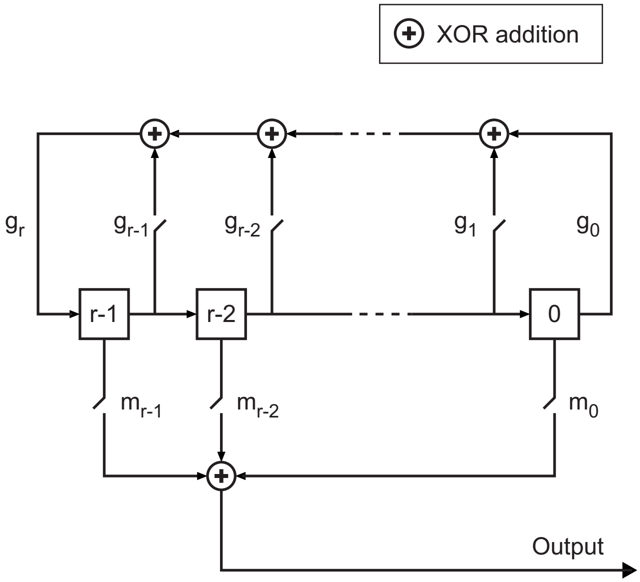
Sequence Generators sublibrary of Comm Sources

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” (Simulink).

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 these formats:

- A polynomial character vector that includes the number `1`, for example, `'z^4 + z + 1'`.
- 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, `'z^8 + z^2 + 1'`, `[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.

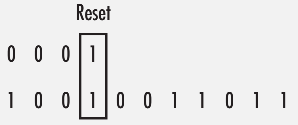
	T = 0	T = 1	T = 2	...	T = d	T = d+1
Shift = 0	x_0	x_1	x_2	...	x_d	x_{d+1}
Shift = d	x_d	x_{d+1}	x_{d+2}	...	x_{2d}	x_{2d+1}

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.

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 the effect of the reset signal on the PN Sequence Generator block.

Reset Signal Properties	PN Sequence Generator block	Reset Signal, Output Signal
Sample time = 1	**Sample time** = 1	

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.

Sequences of Maximum Length

To generate a maximum length sequence for a generator polynomial having degree, *r*, set **Generator polynomial** to a value from the following table. The maximum sequence length is $2^r - 1$. See [1] for more information about the shift-register configurations that these polynomials represent.

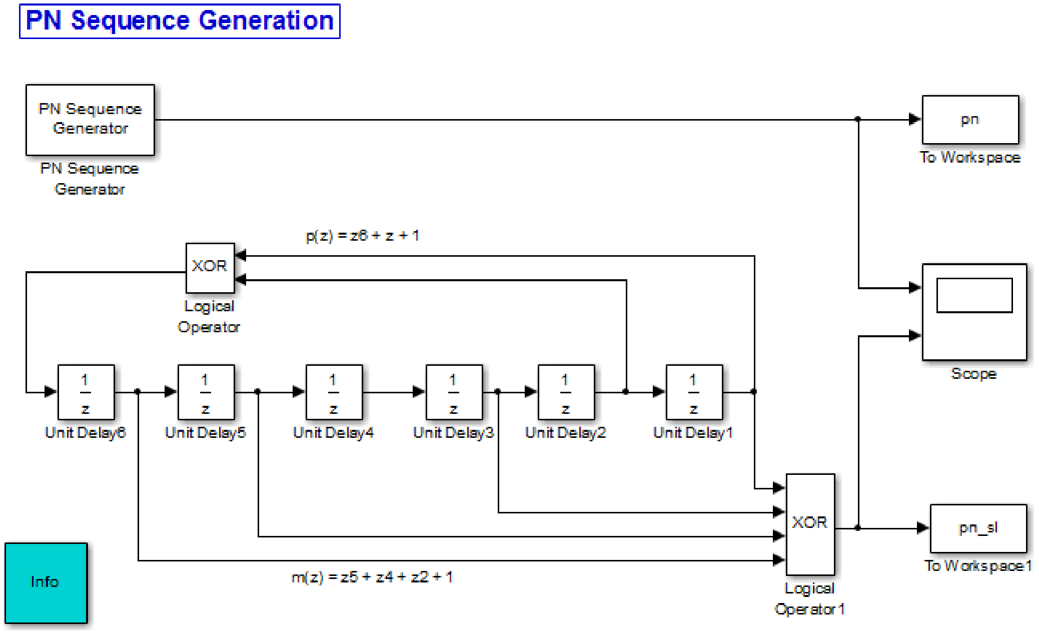
r	Generator Polynomial	r	Generator Polynomial
2	[2 1 0]	21	[21 19 0]
3	[3 2 0]	22	[22 21 0]
4	[4 3 0]	23	[23 18 0]
5	[5 3 0]	24	[24 23 22 17 0]
6	[6 5 0]	25	[25 22 0]
7	[7 6 0]	26	[26 25 24 20 0]
8	[8 6 5 4 0]	27	[27 26 25 22 0]
9	[9 5 0]	28	[28 25 0]
10	[10 7 0]	29	[29 27 0]
11	[11 9 0]	30	[30 29 28 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]
42	[42 23 22 1 0]	49	[49 9 0]
43	[43 6 4 3 0]	50	[50 4 3 2 0]

r	Generator Polynomial	r	Generator Polynomial
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.

Parameters

Generator polynomial

Polynomial, specified as a character vector or vector, 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

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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.

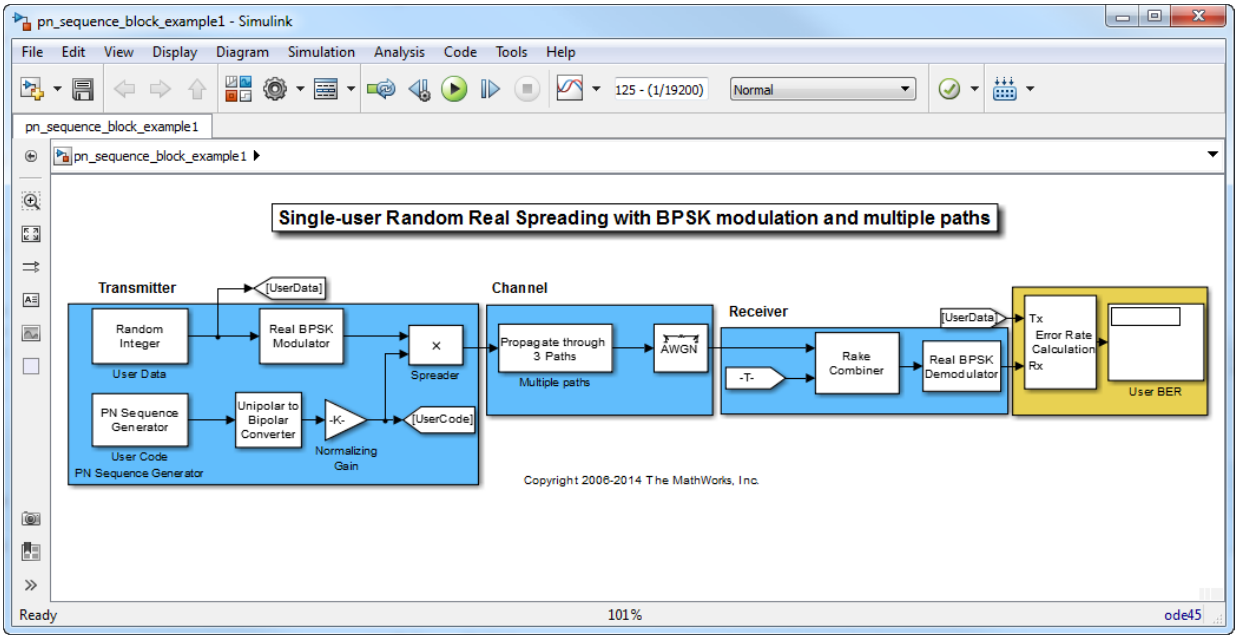
Examples

PN Spreading with Multipath

This example model considers pseudo-random spreading for a single-user system in a multipath transmission environment.

Open the model here: pn_sequence_block_example1

```
modelname = 'pn_sequence_block_example1';
open_system(modelname);
sim(modelname);
```


pn_sequence_block_example1 - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
125 - (1/19200)
Normal
pn_sequence_block_example1
pn_sequence_block_example1
Single-user Random Real Spreading with BPSK modulation and multiple paths
Transmitter
[UserData]
Random Integer
User Data
Real BPSK Modulator
×
Spreader
PN Sequence Generator
User Code
Unipolar to Bipolar Converter
-K-
Normalizing Gain
[UserCode]
PN Sequence Generator
Channel
Propagate through 3 Paths
Multiple paths
AWGN
Receiver
-T-
Rake Combiner
Real BPSK Demodulator
[UserData]
Tx
Error Rate Calculation
Rx
User BER
Copyright 2006-2014 The MathWorks, Inc.
Ready
101%
ode45

In this case for a three path channel, there are gains due to diversity combining. This is made possible by the ideal auto-correlation properties of the PN sequences used.

To experiment with this model further, change the PN Sequence Generator block parameters. Additionally for the same sequences,select other path delays to see performance variations.

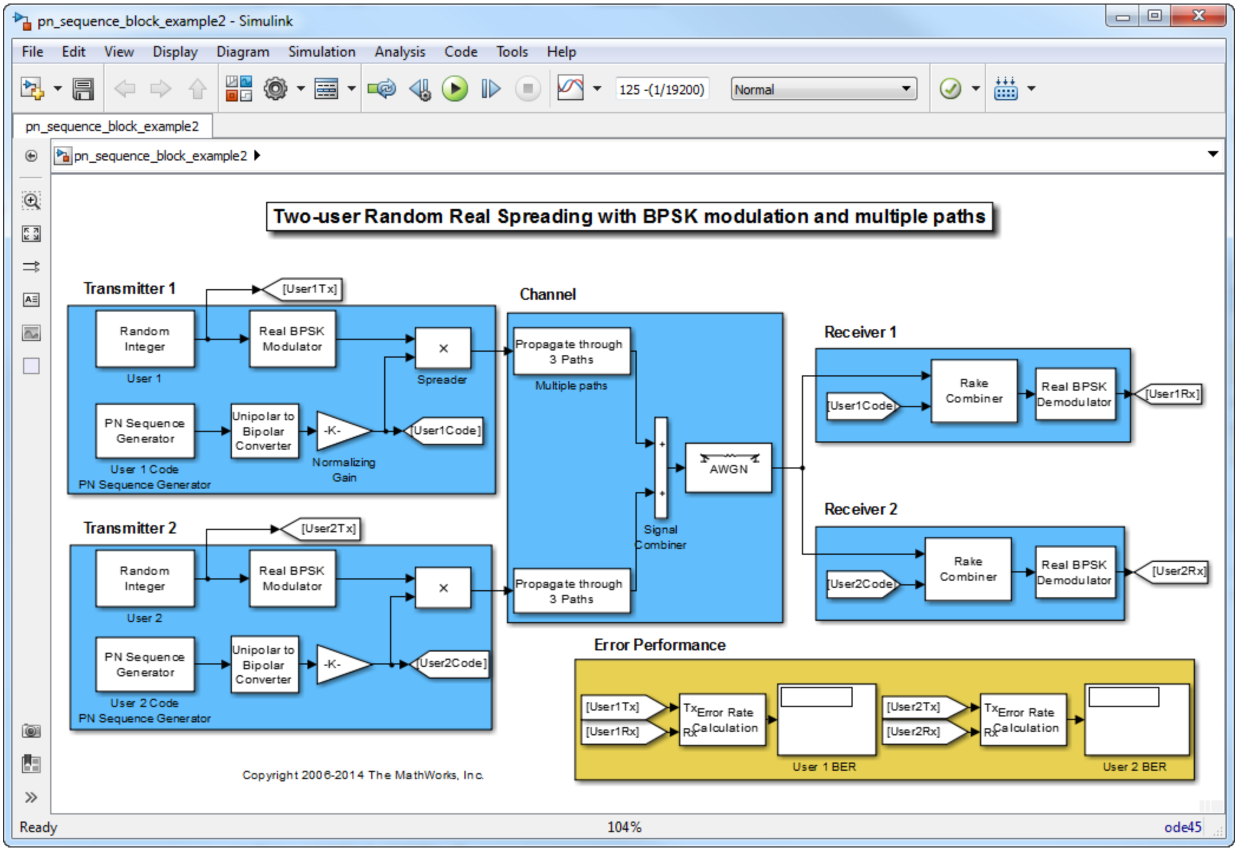
```
close_system(modelname, 0);
```

PN Spreading with Two Users and Multipath

This model considers pseudo-random spreading for a combined two-user transmission in a multipath environment.

Open the model here: pn_sequence_block_example2

```
modelname = 'pn_sequence_block_example2';
open_system(modelname);
sim(modelname);
```



For the two distinct PN sequences used for spreading, note that the individual user performance has now worsened for the same channel conditions (compare 139 errors to 41 from above). This is primarily due to the higher cross-correlation values between the two sequences which prevent ideal separation. Note, there are still advantages to combining as the error rate for a multipath plus AWGN channel with RAKE combining is nearly as good as for an AWGN-only case.

```
close_system(modelname, 0);
```

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see PN Sequence Generator in the HDL Coder documentation.

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.

Introduced before R2006a

Poisson Integer Generator

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 column or row vector, a two-dimensional matrix, or a scalar. The number of rows in the output signal corresponds to the number of samples in one frame and is determined by the **Samples per frame** parameter. The number of columns in the output signal corresponds to the number of channels and is determined by the number of elements in the **Lambda** parameter. See “Sources and Sinks” in *Communications System Toolbox User*'s *Guide* for more details.

Parameters

Lambda

The Poisson parameter λ. Specify λ as a scalar or row vector whose elements are real numbers. If **Lambda** is a scalar, then every element in the output vector shares the same Poisson parameter. If **Lambda** is a row vector, then the number of elements correspond to the number of independent channels output from the block.

Source of initial seed

The source of the initial seed for the random number generator. Specify the source as either `Auto` or `Parameter`. When set to `Auto`, the block uses the global random number stream.

Note When **Source of initial seed** is `Auto` in `Code generation` mode, the random number generator uses an initial seed of zero. Therefore, the block generates the same random numbers each time it is started. Use `Interpreted execution` to ensure that the model uses different initial seeds. If `Interpreted execution` is run in `Rapid accelerator` mode, then it behaves the same as `Code generation` mode.

Initial seed

The initial seed value for the random number generator. Specify the seed as a nonnegative integer scalar. **Initial seed** is available when the **Source of initial seed** parameter is set to `Parameter`.

Sample time

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal. Specify **Samples per frame** as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs an integer every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Output data type

The output type of the block can be specified as a `boolean`, `uint8`, `uint16`, `uint32`, `single`, or `double`. The default is `double`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

See Also

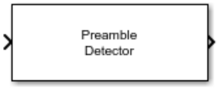
Random Integer Generator; `poissrnd` (Statistics and Machine Learning Toolboxo)

Introduced before R2006a

Preamble Detector

Detect preamble in data

Library: Communications System Toolbox / Synchronization



Description

The Preamble Detector block detects a preamble in a data sequence. A preamble is a set of symbols or bits used in packet-based communication systems to indicate the start of a packet. The block finds the location corresponding to the end of the preamble.

Input/Output Ports

Input

`Port_1` — Input data

scalar | column vector

This port accepts real or complex input data sequences. The input data can be either symbols or bits. This port accepts variable-size arrays.

Data Types: `single` | `double` | `Boolean` | `int8` | `uint8`

Output

`Idx` — Index of last preamble symbol

scalar | column vector

This port outputs the index corresponding to the last element of each detected preamble. This port outputs variable-size arrays.

Data Types: `double`

`DtMt` — Detection metric

scalar | column vector

This port outputs the detection metric. The detection metric has the same size and data type as the input data. This port is available when the **Input** parameter is `Symbol` and the **Output detection metric** is selected.

- If either the preamble or input data is complex, the detection metric is the absolute value of the cross-correlation of the preamble and the input signal.
- If both the preamble and input data are real, the detection metric is the cross-correlation of the preamble and the input signal.

Data Types: `single` | `double`

Parameters

`Input` — Input signal type

`Symbol` (default) | `Bit`

Specify the input signal type as `Symbol` or `Bit`. For binary inputs, set this parameter to `Bit`. For all other inputs, set this parameter to `Symbol`.

`Preamble` — Preamble sequence

`[1 + 1i; 1 - 1i]` (default) | column vector

Specify the preamble sequence as a column vector. If **Input** is `Bit`, the preamble must be binary. If **Input** is `Symbol`, the preamble can be any real or complex sequence.

`Detection threshold` — Detection threshold

`3` (default) | nonnegative real scalar

Specify the detection threshold as a nonnegative real scalar. When the detection metric is greater than or equal to the threshold, the block detects the preamble and updates the index. Tunable.

`Output detection metric` — Enable output detection metric port

`false` (default) | `true`

Select this check box to create the **DtMt** port, from which the detection metric data is output.

Algorithms

Bit Inputs

When the input data is composed of bits, the preamble detector uses an exact pattern match.

Symbol Inputs

When the input data is composed of symbols, the preamble detector uses a cross-correlation algorithm. An FIR filter, in which the coefficients are specified from the preamble, computes the cross-correlation between the input data and the preamble. When a sequence of input samples matches the preamble, the filter output reaches its peak. The index of the peak corresponds to the end of the preamble sequence in the input data. See Discrete FIR Filter for further information on the FIR filter algorithm.

The cross-correlation values that are greater than or equal to the specified threshold are reported as peaks. As a result, there may be no detected peaks or there may be as many detected peaks as there are input samples. Consequently, the selection of the detection threshold is very important.

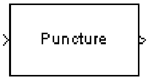
See Also

Carrier Synchronizer | Coarse Frequency Compensator | Symbol Synchronizer | `comm.PreambleDetector`

Introduced in R2016b

Puncture

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.

Parameters

Puncture vector

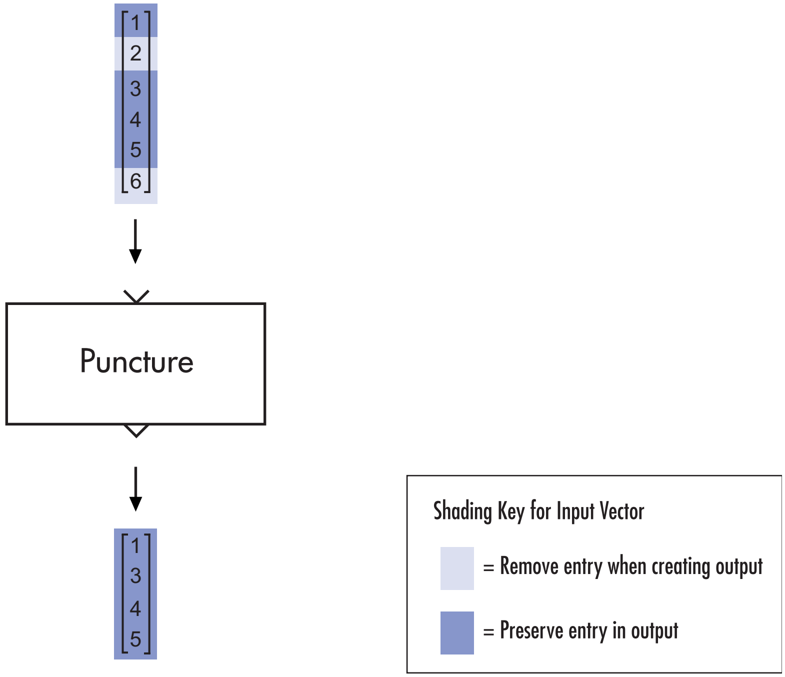
A binary vector whose pattern of 0s (1s) indicates which elements of the input the block should remove (preserve).

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.


1
2
3
4
5
6
Puncture
1
3
4
5
Shading Key for Input Vector
= Remove entry when creating output
= Preserve entry in output

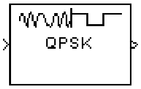
See Also

Insert Zero

Introduced before R2006a

QPSK Demodulator Baseband

Demodulate QPSK-modulated data



Library

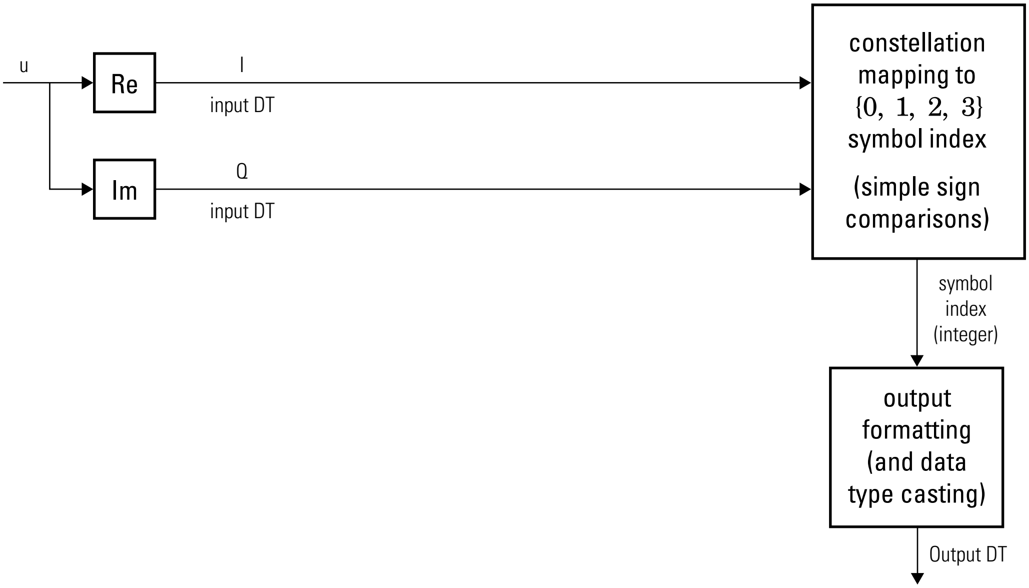
PM, in Digital Baseband sublibrary of Modulation

Description

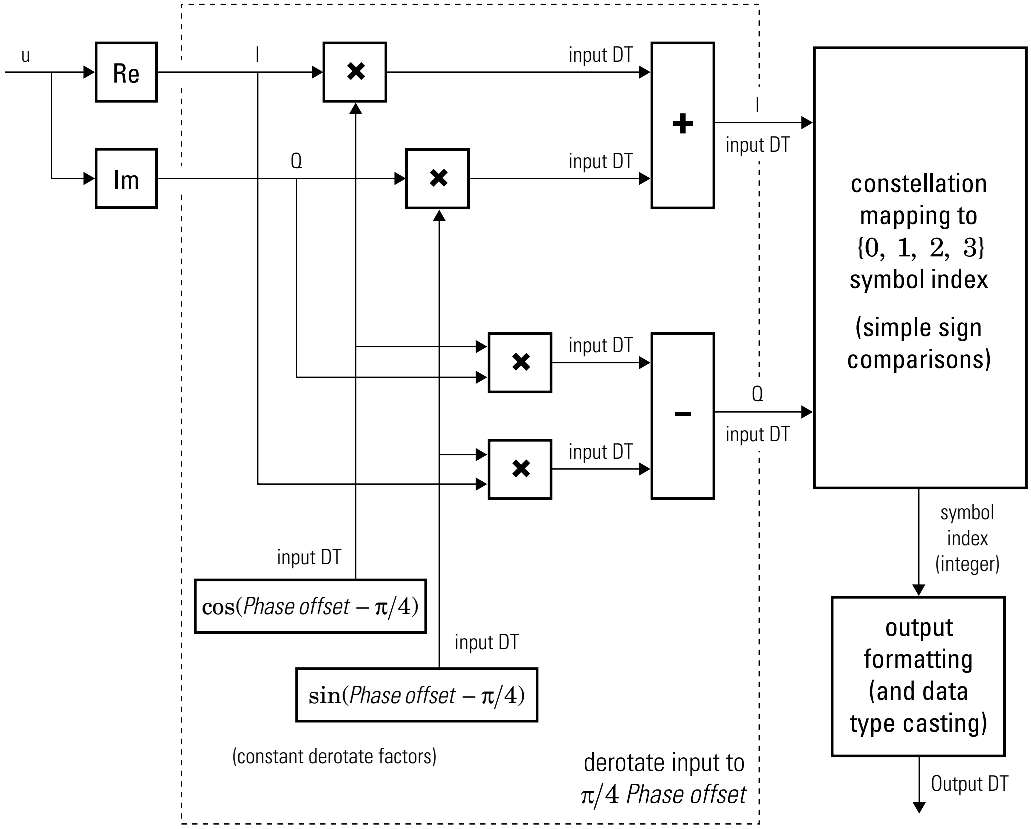
The QPSK Demodulator Baseband block demodulates a signal that was modulated using the quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

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 2-791.

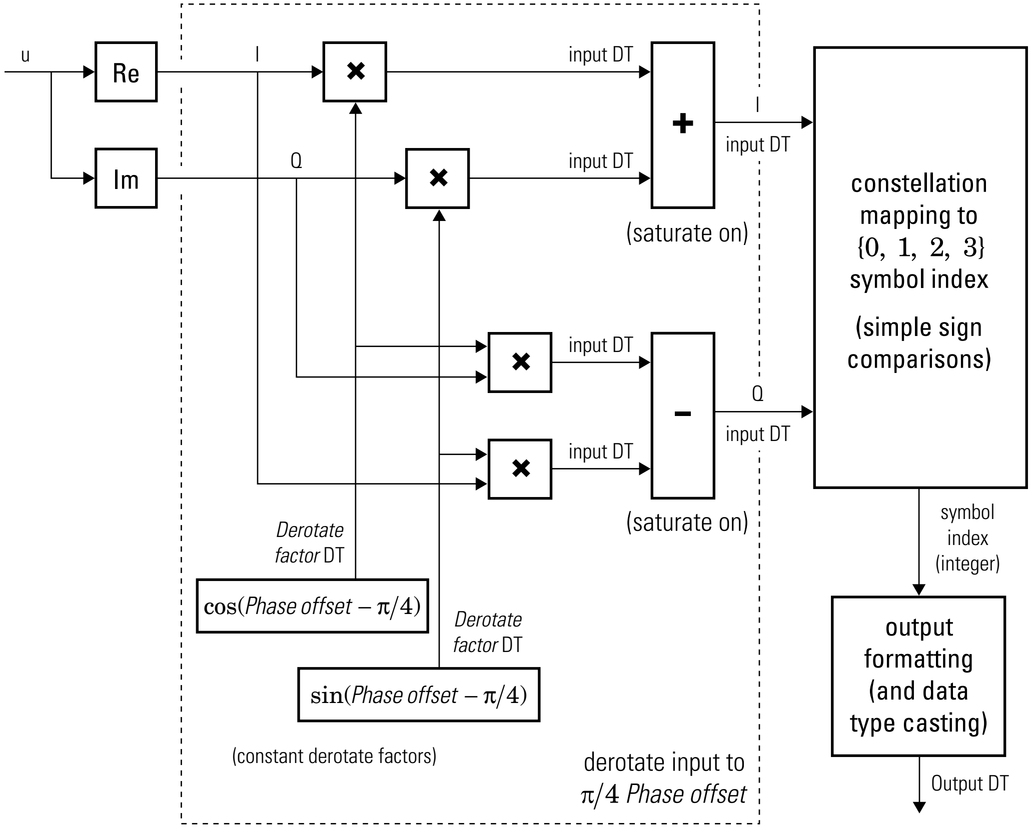
Algorithm


u
Re
Im
I
input DT
Q
input DT
constellation mapping to $\{0,\ 1,\ 2,\ 3\}$ symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of $\frac{\pi}{4}$)


u
Re
Im
I
Q
input DT
input DT
+
I
input DT
input DT
input DT
–
Q
input DT
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
input DT
cos(Phase offset – π/4)
input DT
sin(Phase offset – π/4)
(constant derotate factors)
derotate input to π/4 Phase offset
output formatting (and data type casting)
Output DT

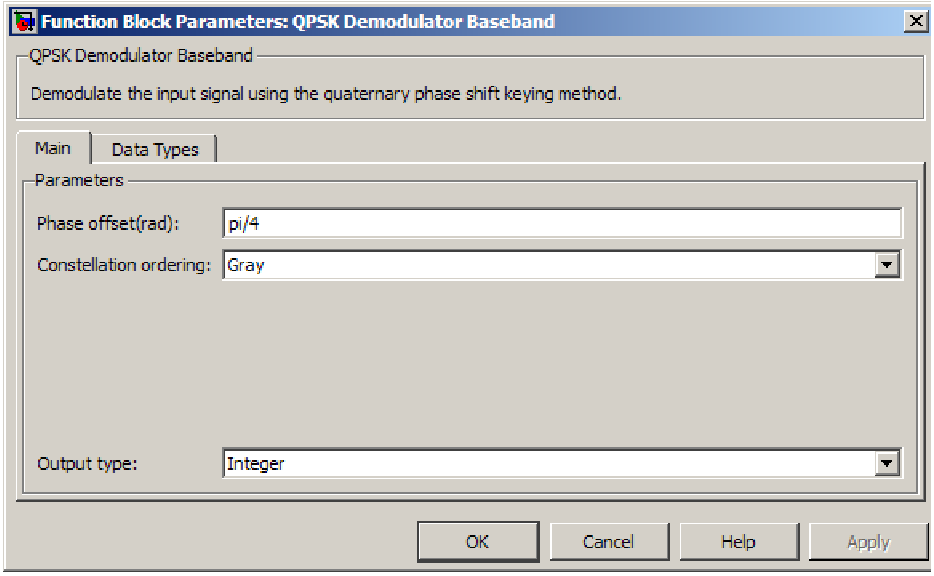
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



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



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Output type: Integer

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

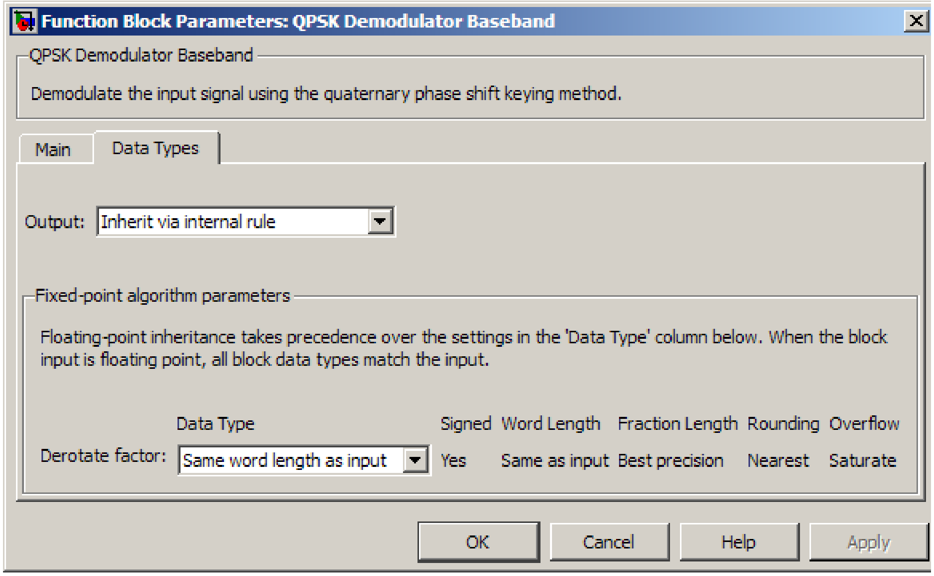
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

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.



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Output: Inherit via internal rule

Fixed-point algorithm parameters

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

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Derotate factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate

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.

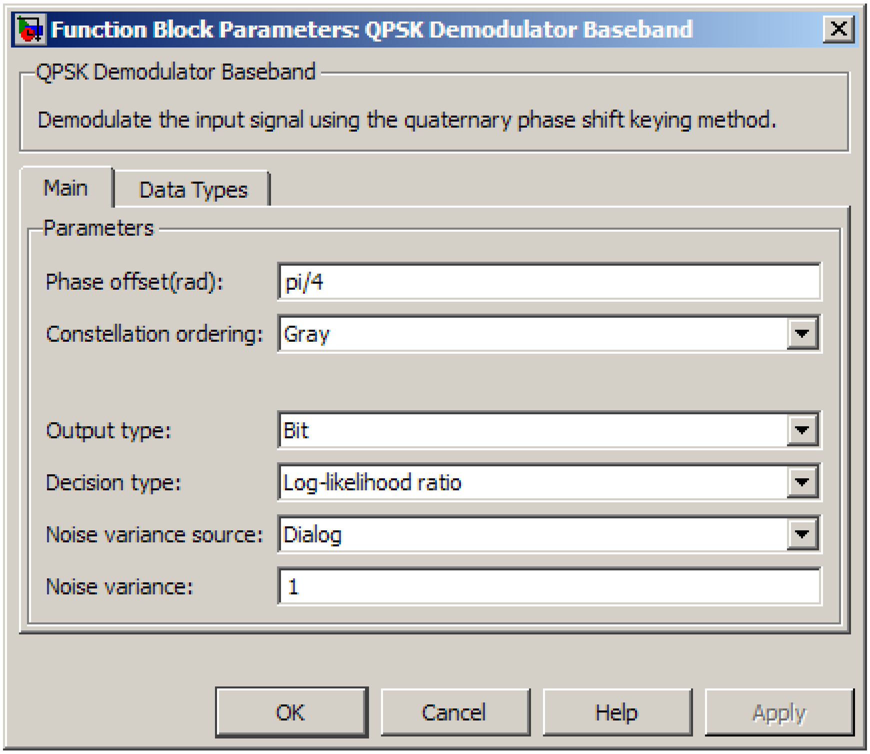
If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, and **Output type** is `Bit`, the output data type is the ideal minimum one-bit size, i.e., `ufix(1)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding 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.



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Output type: Bit

Decision type: Log-likelihood ratio

Noise variance source: Dialog

Noise variance: 1

Data Types Pane for Soft-Decision

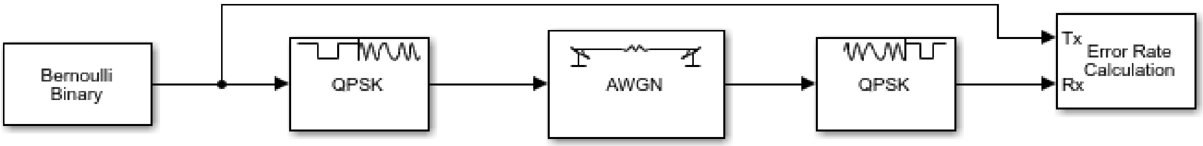
For bit outputs, when **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is inherited from the input (e.g., if the input is of data type `double`, the output is also of data type `double`).

Examples

Demodulate Noisy QPSK Signal

Modulate and demodulate a noisy QPSK signal.

Open the QPSK demodulation model.



Run the simulation. The results are saved to the base workspace, where the variable `ErrorVec` is a 1-by-3 row vector. The BER is found in the first element.

Display the error statistics. For the Eb/No provided, 4.3 dB, the resultant BER is approximately 0.01. Your results may vary slightly.

```
ans =

    0.0112
```

Increase the Eb/No to 7 dB. Rerun the simulation, and observe that the BER has decreased.

```
ans =

   1.0000e-03
```

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`

Port	Supported Data Types
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`

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see QPSK Demodulator Baseband in the HDL Coder documentation.

Pair Block

QPSK Modulator Baseband

See Also

M-PSK Demodulator Baseband, BPSK Demodulator Baseband, DQPSK Demodulator Baseband

Introduced before R2006a

QPSK Modulator Baseband

Modulate using quadrature phase shift keying method

Library

PM in Digital Baseband sublibrary of Modulation

Description

The QPSK Modulator Baseband block modulates using the quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

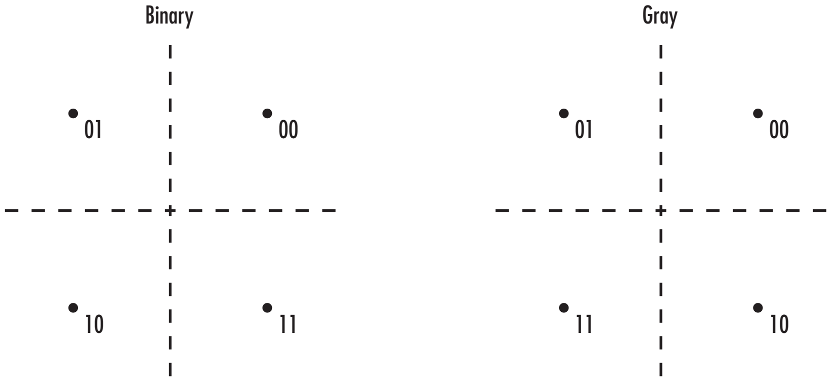
Integer-Valued Signals and Binary-Valued Signals

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`.

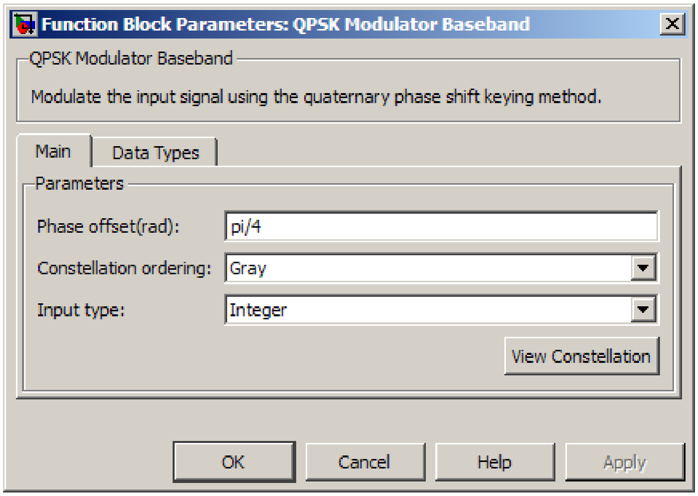


In the previous figure, the most significant bit (i.e. the left-most bit), is the first bit input to the block. For additional information about Gray mapping, see the M-PSK Modulator Baseband help page.

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



Function Block Parameters: QPSK Modulator Baseband

QPSK Modulator Baseband

Modulate the input signal using the quaternary phase shift keying method.

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Input type: Integer

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.

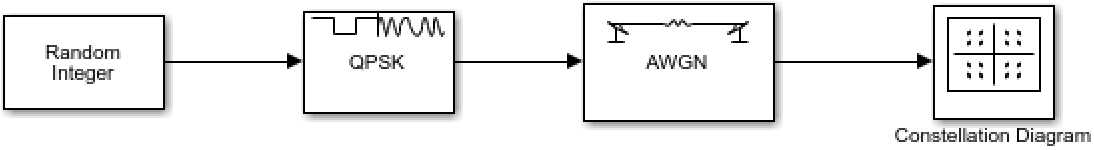
Output fraction length

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.

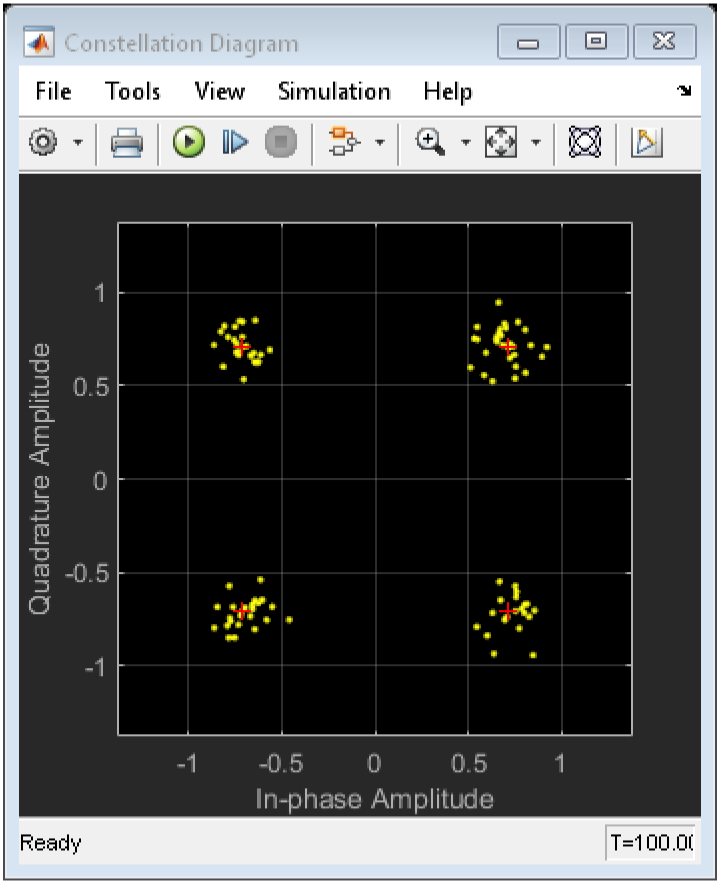
Examples

Plot Noisy QPSK Constellation

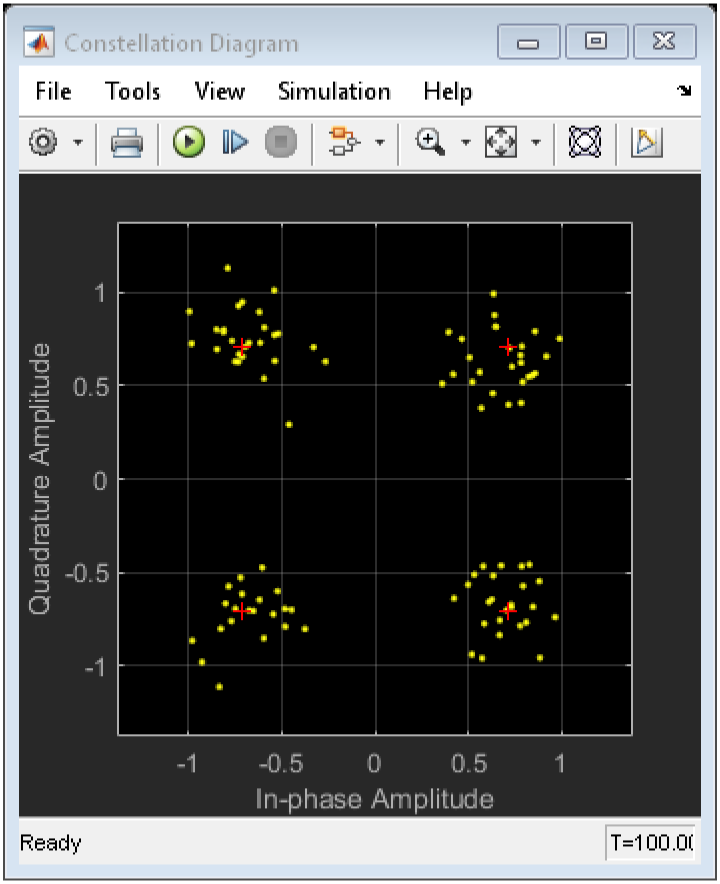
Open the QPSK model. The model generates QPSK data, applies white noise, and displays the resulting constellation diagram.



Run the model.



Change the Eb/No of the AWGN Channel block from 15 dB to 10 dB.


Constellation Diagram
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=100.0(

The noise level increases as shown by the greater distance between the samples.

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

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see QPSK Modulator Baseband in the HDL Coder documentation.

Pair Block

QPSK Demodulator Baseband

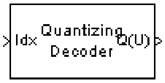
See Also

M-PSK Modulator Baseband, BPSK Modulator Baseband, DQPSK Modulator Baseband

Introduced before R2006a

Quantizing Decoder

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 2-802 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.

Parameters

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

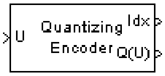
See Also

Scalar Quantizer Decoder (DSP System Toolbox documentation)

Introduced before R2006a

Quantizing Encoder

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 2-804 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 $x \leq P(1)$
- m if $P(\mathrm{m}) < x \leq P(\mathrm{m}+1)$
- n if $P(\mathrm{n}) < x$

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.

Parameters

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

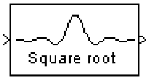
See Also

Scalar Quantizer Encoder (DSP System Toolbox documentation), `lloyds` (Communications System Toolbox documentation)

Introduced before R2006a

Raised Cosine Receive Filter

Apply pulse shaping by downsampling signal 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: *L* * **Filter span in symbols** + 1, where *L* is the value of the **Input samples per symbol** parameter (not the **Output samples per symbol** parameter, as in the case of the Raised Cosine Transmit Filter block).

The block normalizes the filter coefficients to unit energy. If you specify a **Linear amplitude filter gain** other than `1`, then the block scales the normalized filter coefficients using the gain value you specify.

Decimating the Filtered Signal

To have the block decimate the filtered signal, set the **Decimation factor** parameter to a value greater than `1`.

If K represents the **Decimation factor** parameter value, then the block retains 1/*K* of the samples, choosing them as follows:

- If the **Decimation offset** parameter is zero, then the block selects the samples of the filtered signal indexed by 1, *K*+1, 2**K*+1, 3**K*+1, etc.
- If the **Decimation offset** parameter is a positive integer less than M, 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 decimation, set **Decimation factor** to `1`. 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 2-813 table on this page.

If you set **Decimation factor** to `1`, then the input and output signals share the same sampling mode, sample time, and vector length.

If you set **Decimation factor** to *K*, which is greater than `1`, then *K* 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/*K* times that of the input (M_o = M_i/K), In this mode, the input frame size, M_i, must be a multiple of *K*.

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 K times longer than the input sample period ($T_{so} = K*T_{si}$), 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}) K times longer than the input frame period ($T_{fo} = K*T_{fi}$).

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.

Parameters

Filter shape

Specify the filter shape as `Square root` or `Normal`.

Rolloff factor

Specify the rolloff factor of the filter. Use a real number between `0` and `1`.

Filter span in symbols

Specify the number of symbols the filter spans as an even, integer-valued positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that this parameter specifies.

Input samples per symbol

An integer greater than 1 representing the number of samples that represent one symbol in the input signal.

Decimation factor

Specify the decimation factor the block applies to the input signal. The output samples per symbol equals the value of the input samples per symbol divided by the decimation factor. If the decimation factor is one, then the block only applies filtering. There is no decimation.

Decimation offset

Specify the decimation offset in samples. Use a value between `0` and **Decimation factor** `-1`.

Linear amplitude filter gain

Specify a positive scalar value that the block uses to scale the filter coefficients. By default, the block normalizes filter coefficients to provide unit energy gain. If you specify a gain other than `1`, the block scales the normalized filter coefficients using the gain value you specify.

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.

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 *K*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block processes the signal such that the output sample rate is *K* times slower than the input sample rate.

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 (DSP System Toolbox) or "Rounding Mode: Simplest" (Fixed-Point Designer).

Saturate on integer overflow

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" (DSP System Toolbox) 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 **Saturate on integer overflow** 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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 data type settings against changes by the fixed-point tools

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.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Raised Cosine Receive Filter in the HDL Coder documentation.

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

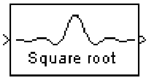
See Also

`rcosdesign`, `comm.RaisedCosineTransmitFilter`

Introduced before R2006a

Raised Cosine Transmit Filter

Apply pulse shaping by upsampling 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 shape** 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.

Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that the **Filter span in symbols** parameter specifies. The **Filter span in symbols**, N, and the **Output samples per symbol**, L, determine the length of the filter's impulse response, which is *L* * **Filter span in symbols** + 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 block normalizes the filter coefficients to unit energy. If you specify a **Linear amplitude filter gain** other than `1`, then the block scales the normalized filter coefficients using the gain value you specify.

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 2-820 table on this page.

The **Rate options** method and the value of the **Output samples per symbol**, *L*, 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 *L* times larger than that of the input ($M_o = M_i*L$), where *L* represents the value of the **Output samples per symbol** parameter.

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/L 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**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 ($T_{fo} = T_{fi}/L$).

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.

Parameters

Filter shape

Specify the filter shape as `Square root` or `Normal`.

Rolloff factor

Specify the rolloff factor of the filter. Use a real number between `0` and `1`.

Filter span in symbols

Specify the number of symbols the filter spans as an even, integer-valued positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that this parameter specifies.

Output samples per symbol

Specify the number of output samples for each input symbol. The default is 8. This property accepts an integer-valued, positive scalar. The number of taps for the raised cosine filter equals the value of this parameter multiplied by the value of the **Filter span in symbols** parameter.

Linear amplitude filter gain

Specify a positive scalar value that the block uses to scale the filter coefficients. By default, the block normalizes filter coefficients to provide unit energy gain. If you specify a gain other than `1`, the block scales the normalized filter coefficients using the gain value you specify.

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.

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.

Export filter coefficients to workspace

Select this check box to create a variable in the MATLAB workspace that contains the filter coefficients.

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 (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturage on integer overflow

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” (DSP System Toolbox) 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 **Saturate on integer overflow** 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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 data type settings against changes by the fixed-point tools

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.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Raised Cosine Transmit Filter in the HDL Coder documentation.

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

`rcosdesign`, `comm.RaisedCosineReceiveFilter`

Introduced before R2006a

Random Deinterleaver

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.

Parameters

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

Introduced before R2006a

Random Integer Generator

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 **Set size** defined in the dialog box.

The **Set size** can be either a scalar or a row vector. If it is a scalar, then all output random variables are independent and identically distributed (i.i.d.). If the **Set size** is a vector, then the length of the vector determines the number of output channels. The channels can have differing output ranges.

If the **Initial seed** parameter is a constant, then the resulting noise is repeatable.

Attributes of Output Signal

The output signal can be a column or row vector, a two-dimensional matrix, or a scalar. The number of rows in the output signal corresponds to the number of samples in one frame and is determined by the **Samples per frame** parameter. The number of columns in the output signal corresponds to the number of channels and is determined by the number of elements in the **Set size** parameter. See “Sources and Sinks” in *Communications System Toolbox User*'s *Guide* for more details.

Parameters

Set size

The set size determines the range of output values. The block generates integers ranging from 0 to **Set size** – 1. Specify the set size as a scalar or row vector of positive integers. The number of elements in **Set size** correspond to the number of independent channels output from the block.

Source of initial seed

The source of the initial seed for the random number generator. Specify the source as either `Auto` or `Parameter`. When set to `Auto`, the block uses the global random number stream.

Note When **Source of initial seed** is `Auto` in `Code generation` mode, the random number generator uses an initial seed of zero. Therefore, the block generates the same random numbers each time it is started. Use `Interpreted execution` to ensure that the model uses different initial seeds. If `Interpreted execution` is run in `Rapid accelerator` mode, then it behaves the same as `Code generation` mode.

Initial seed

The initial seed value for the random number generator. Specify the seed as a nonnegative integer scalar. **Initial seed** is available when the **Source of initial seed** parameter is set to `Parameter`.

Sample time

The time between each sample of a column of the output signal.

Samples per frame

The number of samples per frame in one channel of the output signal. Specify **Samples per frame** as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs an integer every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Output data type

The output type of the block can be specified as a `boolean`, `uint8`, `uint16`, `uint32`, `single`, or `double`. The default is `double`. For Boolean typed outputs, the **Set size** must be 2.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

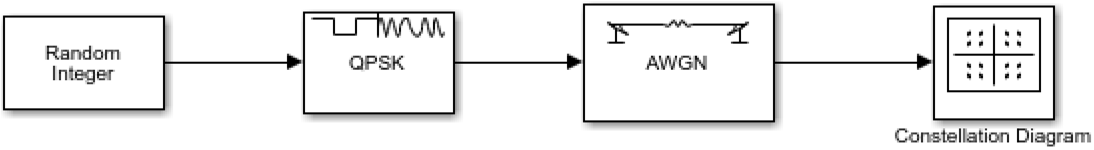
`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

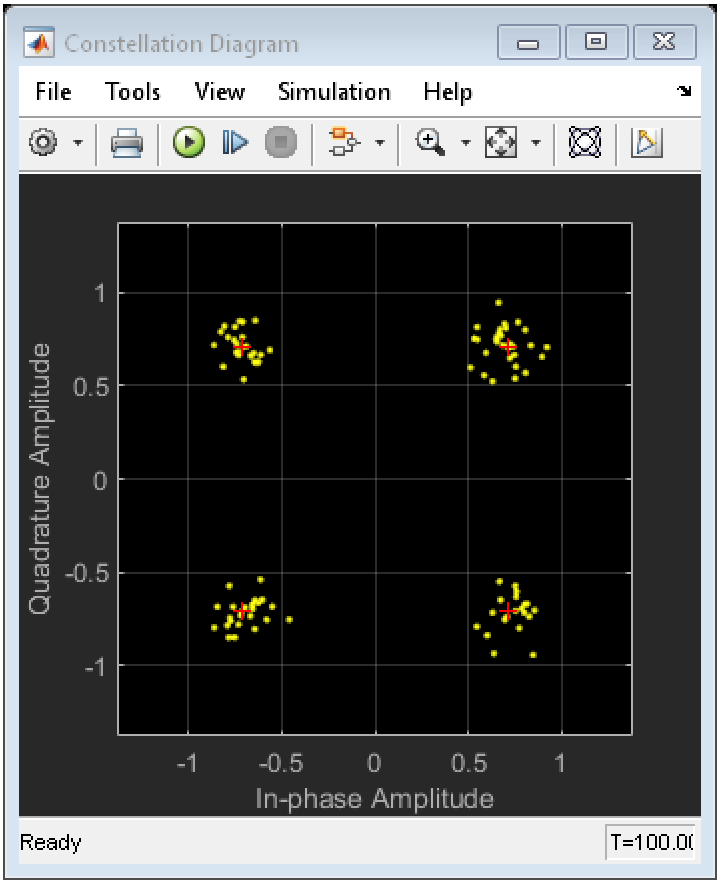
Examples

Plot Noisy QPSK Constellation

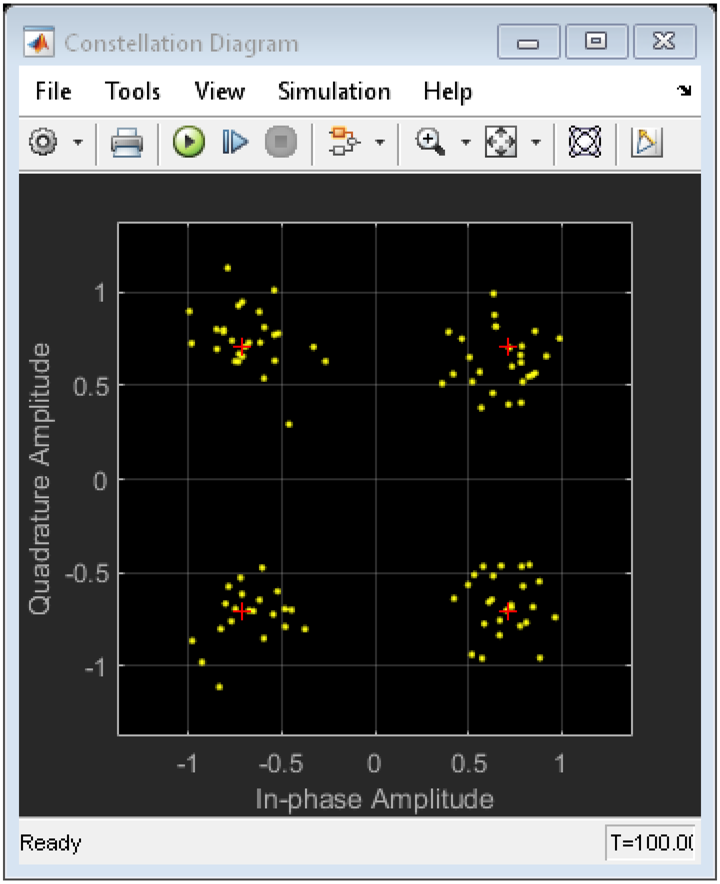
Open the QPSK model. The model generates QPSK data, applies white noise, and displays the resulting constellation diagram.



Run the model.



Change the Eb/No of the AWGN Channel block from 15 dB to 10 dB.



The noise level increases as shown by the greater distance between the samples.

See Also

randi

Introduced before R2006a

Random Interleaver

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.

Parameters

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

Introduced before R2006a

Rayleigh Noise Generator

Generate Rayleigh distributed noise

Note Rayleigh Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

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

$$f(x) = \begin{cases} \frac{x}{\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

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” 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.

Parameters

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.

Blocks

MIMO Fading Channel

Functions

`raylrnd`

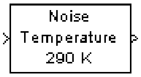
References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

Introduced before R2006a

Receiver Thermal Noise

Apply receiver thermal noise to complex baseband signal



Library

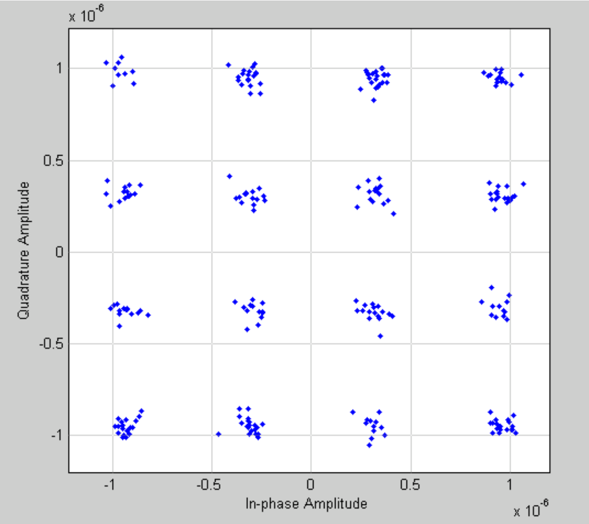
RF Impairments

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 K.
- `Noise figure` specifies the receiver noise in dB for an input noise temperature of 290 K. This is the decibel equivalent of the noise factor.
- `Noise factor` specifies the receiver noise for an input noise temperature of 290 K. This is the linear equivalent of the noise figure.

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`

Parameters

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 K.

Noise figure (dB)

Scalar specifying the amount of noise in dB relative to a noise temperature of 290 degrees K. This parameter specifies the noise contribution of the receiver circuitry only. To add the effects of antenna noise, select the **Add 290K antenna noise** check box.

Noise factor

Scalar specifying the amount of noise relative to a noise temperature of 290 degrees K. This parameter specifies the noise contribution of the receiver circuitry only. To add the effects of antenna noise, select the **Add 290K antenna noise** check box.

Add 290K antenna noise

Select this check box to add 290 K antenna noise to the signal. This parameter is available when **Specification method** is `Noise figure` or `Noise factor`.

Initial seed

The initial seed value for the random number generator that generates the noise.

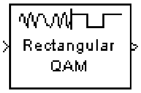
See Also

Free Space Path Loss | `comm.ThermalNoise`

Introduced before R2006a

Rectangular QAM Demodulator Baseband

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.

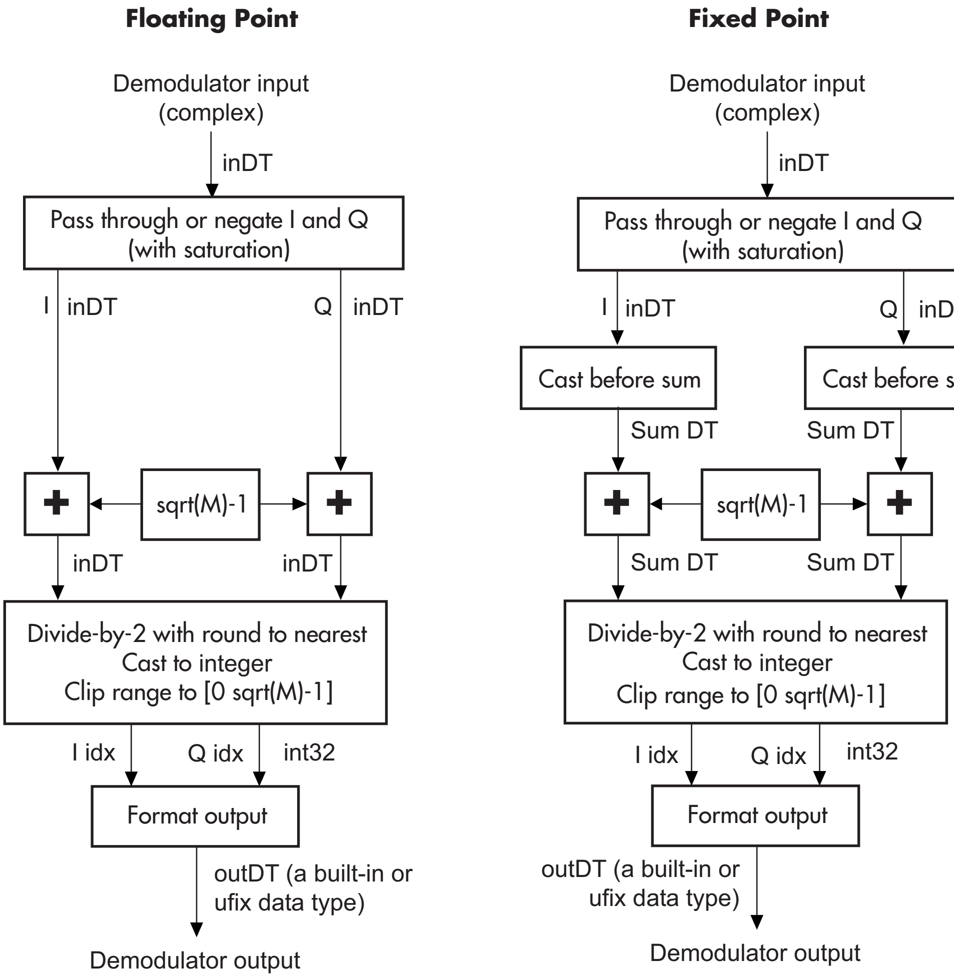
This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 2-846 table on this page.

Hard Decision Algorithm

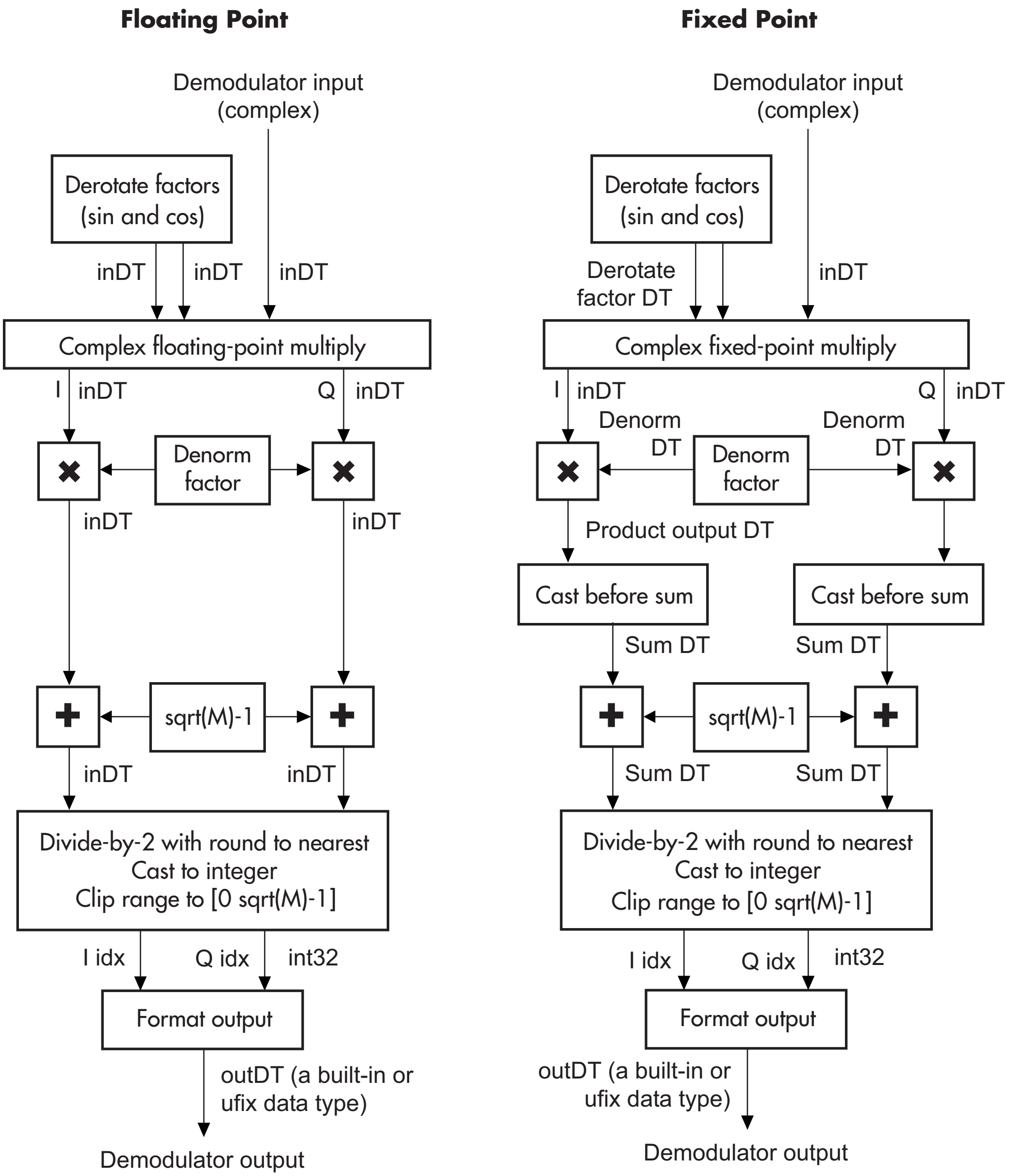
The demodulator algorithm maps received input signal constellation values to M-ary integer I and Q symbol indices between 0 and $\sqrt{M}-1$ and then maps these demodulated symbol indices to formatted output values.

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 π/2, and/or the derived denormalization factor is 1.


Floating Point
Demodulator input (complex)
inDT
Pass through or negate I and Q (with saturation)
I
inDT
Q
inDT
+
sqrt(M)-1
+
inDT
inDT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output
Fixed Point
Demodulator input (complex)
inDT
Pass through or negate I and Q (with saturation)
I
inDT
Q
inDT
Cast before sum
Cast before sum
Sum DT
Sum DT
+
sqrt(M)-1
+
Sum DT
Sum DT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output

Signal-Flow Diagrams with Trivial Phase Offset and Denormalization Factor Equal to 1


Floating Point
Demodulator input (complex)
Derotate factors (sin and cos)
inDT
inDT
inDT
Complex floating-point multiply
I
inDT
Q
inDT
Denorm factor
inDT
inDT
sqrt(M)-1
inDT
inDT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output
Fixed Point
Demodulator input (complex)
Derotate factors (sin and cos)
Derotate factor DT
inDT
Complex fixed-point multiply
I
inDT
Q
inDT
Denorm DT
Denorm factor
Denorm DT
Product output DT
Cast before sum
Cast before sum
Sum DT
Sum DT
sqrt(M)-1
Sum DT
Sum DT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output

Signal-Flow Diagrams with Nontrivial Phase Offset and Nonunity Denormalization Factor

Parameters

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

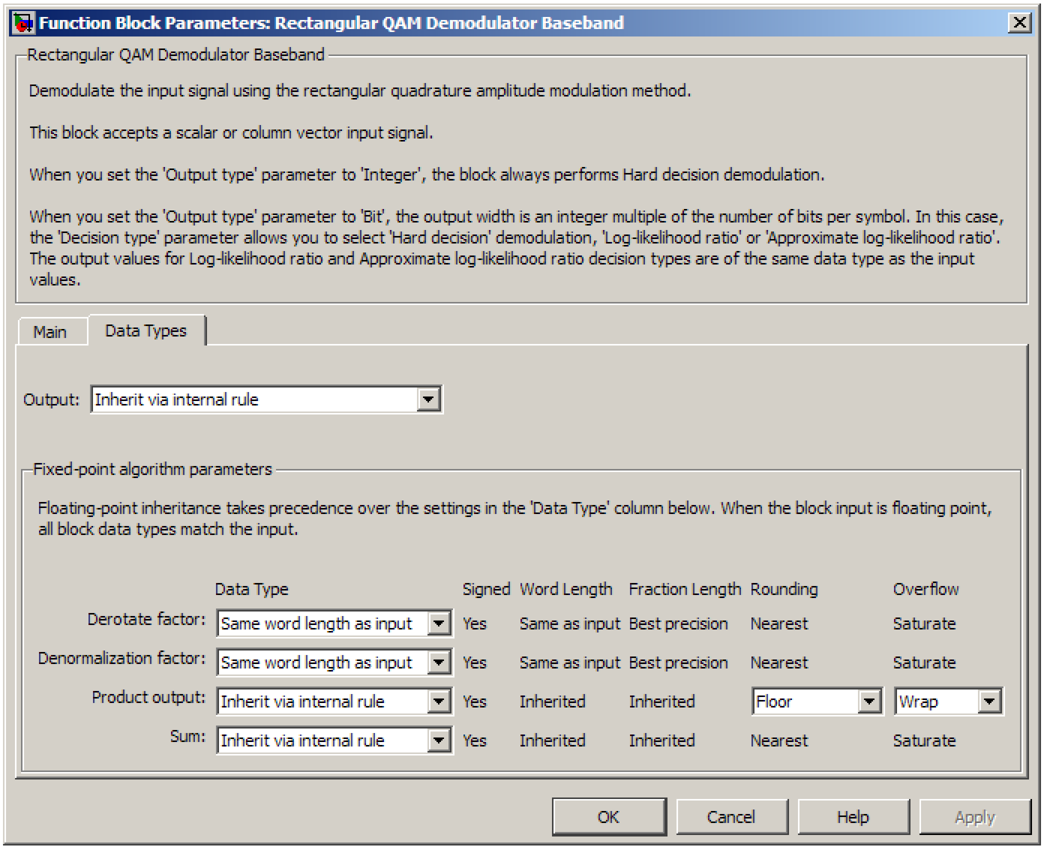
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

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.



Function Block Parameters: Rectangular QAM Demodulator Baseband

Rectangular QAM Demodulator Baseband

Demodulate the input signal using the rectangular quadrature amplitude modulation method.

This block accepts a scalar or column vector input signal.

When you set the 'Output type' parameter to 'Integer', the block always performs Hard decision demodulation.

When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol. In this case, the 'Decision type' parameter allows you to select 'Hard decision' demodulation, 'Log-likelihood ratio' or 'Approximate log-likelihood ratio'. The output values for Log-likelihood ratio and Approximate log-likelihood ratio decision types are of the same data type as the input values.

Output: Inherit via internal rule

Fixed-point algorithm parameters

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

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Derotate factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Denormalization factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Product output:	Inherit via internal rule	Yes	Inherited	Inherited	Floor	Wrap
Sum:	Inherit via internal rule	Yes	Inherited	Inherited	Nearest	Saturate

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 π/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 (DSP System Toolbox) 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” (DSP System Toolbox) or “Rounding Mode: Simplest” (Fixed-Point Designer).

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 on page 2-838 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.

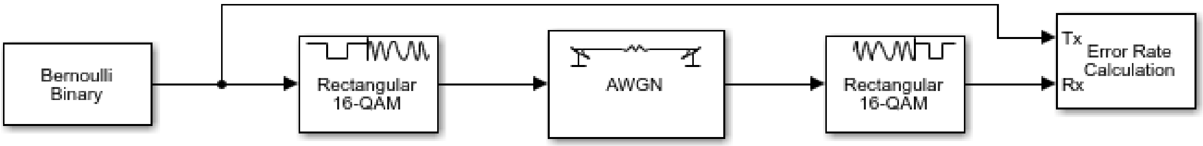
Setting to `Same as product output` allows the Sum data type to be the same as the **Product output** data type (when **Product output** is used). If the **Product output** is not used, then this setting will be ignored and the `Inherit via internal rule` Sum setting will be used.

Examples

Demodulate Noisy QAM Signal

Modulate and demodulate a noisy QAM signal.

Open the QAM demodulation model.



Run the simulation. The results are saved to the base workspace, where the variable `ErrorVec` is a 1-by-3 row vector. The BER is found in the first element.

Display the error statistics. For the Eb/No provided, 2 dB, the resultant BER is approximately 0.1. Your results may vary slightly.

```
ans =

    0.0948
```

Increase the Eb/No to 4 dB. Rerun the simulation, and observe that the BER has decreased.

```
ans =

    0.0167
```

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`

Port	Supported Data Types
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 • 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`

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Rectangular QAM Demodulator Baseband in the HDL Coder documentation.

Pair Block

Rectangular QAM Modulator Baseband

See Also

General QAM Demodulator Baseband

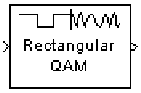
References

[1] Smith, Joel G., “Odd-Bit Quadrature Amplitude-Shift Keying,” *IEEE Transactions on Communications*, Vol. COM-23, March 1975, 385–389.

Introduced before R2006a

Rectangular QAM Modulator Baseband

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 2-855.

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

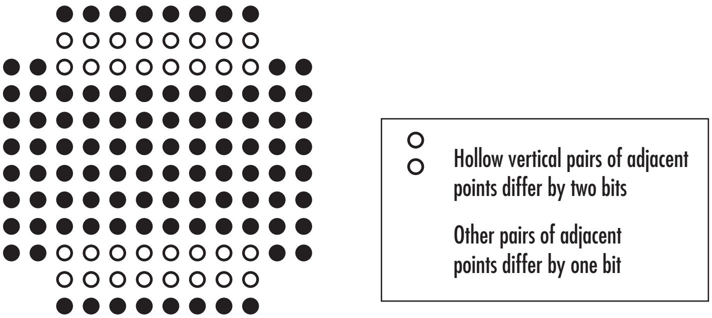
where

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.


Hollow vertical pairs of adjacent points differ by two bits
Other pairs of adjacent points differ by one bit

For details about the Gray coding, see the reference page for the M-PSK Modulator Baseband block and the paper listed in References on page 2-856. 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*.

Parameters

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.

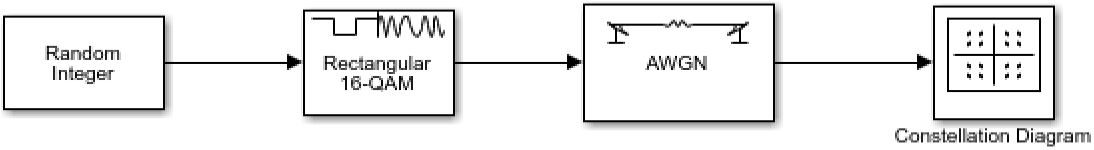
Output fraction length

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.

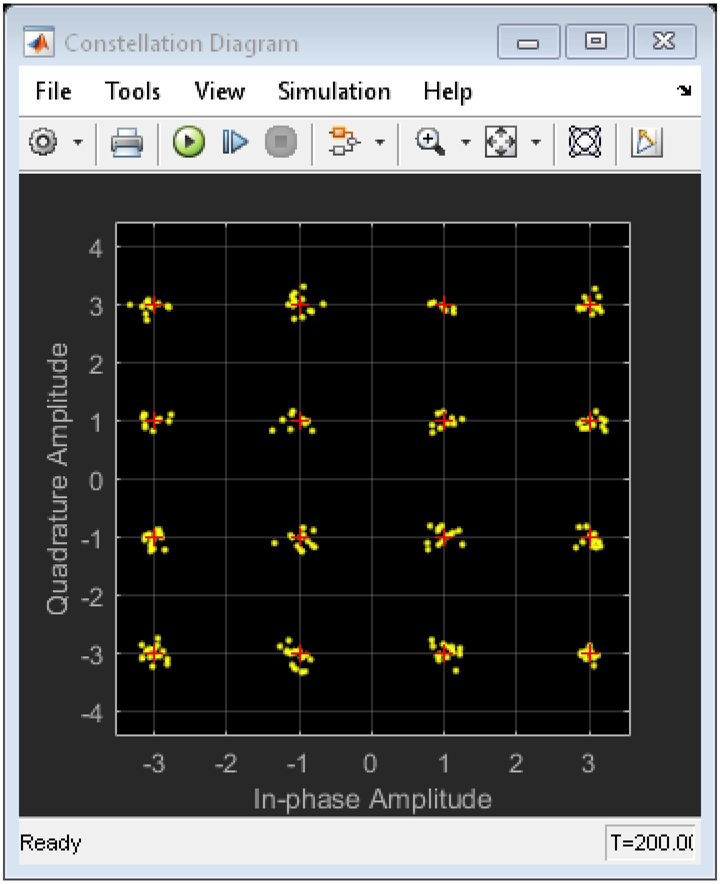
Examples

Plot Noisy 16-QAM Constellation

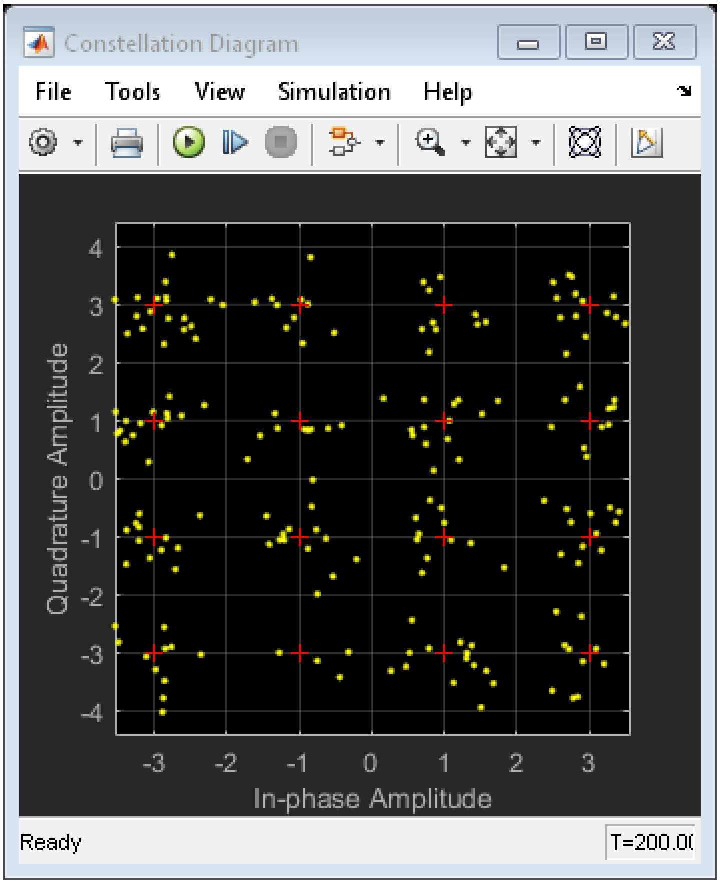
Open the 16-QAM model. The model generates a QAM signal, applies white noise, and displays the resulting constellation diagram.



Run the model.



Change the Eb/No of the AWGN Channel block from 20 dB to 10 dB. Observe the increase in the noise.



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
	• $ufix(\log_2 M)$ when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Rectangular QAM Modulator Baseband in the HDL Coder documentation.

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.

Introduced before R2006a

Rectangular QAM TCM Decoder

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 2-859.

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**.

Parameters

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.

Introduced before R2006a

Rectangular QAM TCM Encoder

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". 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

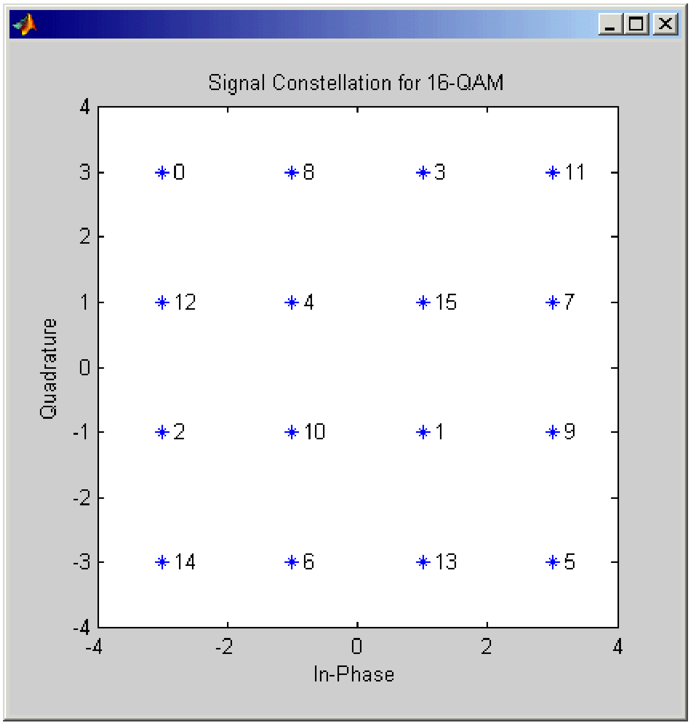
  ```
  poly2trellis(7,[171 133],171)
  ```
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

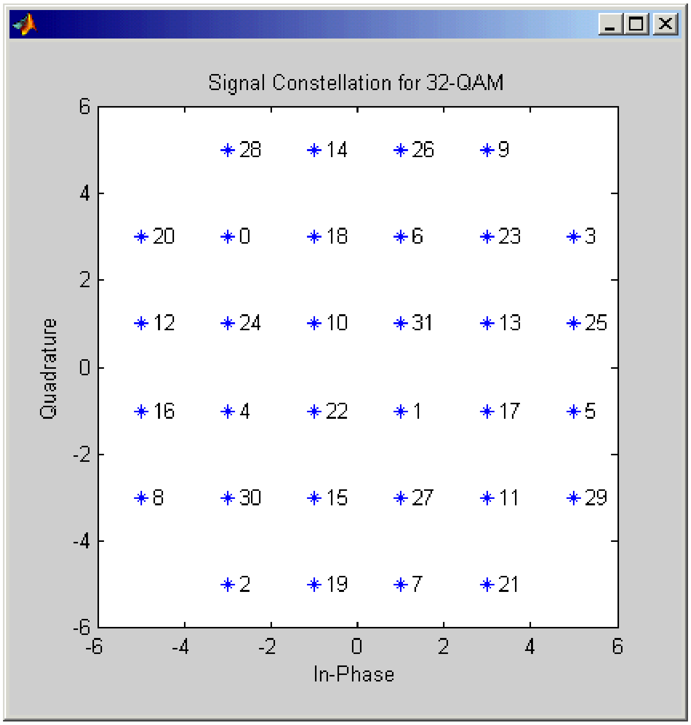
The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation mode` to **Reset on nonzero input via port**. The block then opens a second input port, labeled `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.

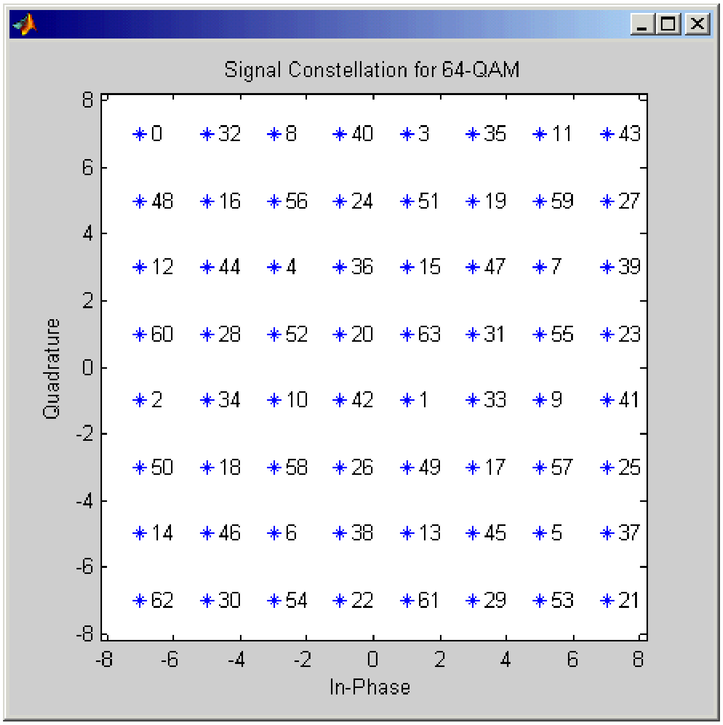
Signal Constellations

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
6
4
2
0
-2
-4
-6
-6
-4
-2
0
2
4
6
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



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.

Parameters

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.

Introduced before R2006a

Repeat

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.

Introduced in R2008a

Rician Noise Generator

Generate Rician distributed noise

Note

Note Rician Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

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.

Parameters

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.

Blocks

MATLAB Function | MIMO Fading Channel

Functions

`randn`

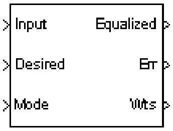
References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

Introduced before R2006a

RLS Decision Feedback Equalizer

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.

Parameters

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.

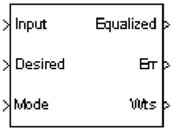
See Also

RLS Linear Equalizer, LMS Decision Feedback Equalizer, CMA Equalizer

Introduced before R2006a

RLS Linear Equalizer

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.

Parameters

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

Introduced before R2006a

Scrambler

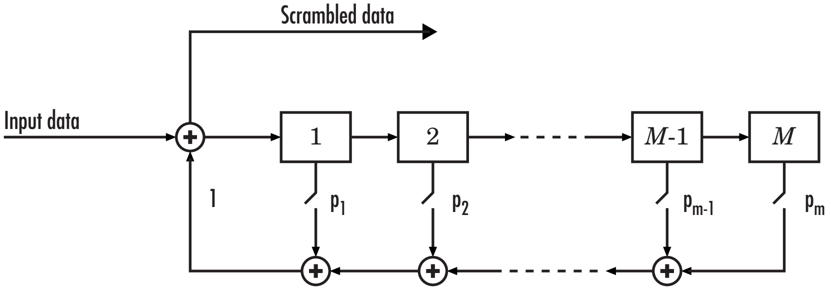
Scramble input signal

Library: Communications System Toolbox / Sequence Operations

Description

The Scrambler block scrambles a scalar or column vector input signal.

One purpose of scrambling is to reduce the length of consecutive 0s or 1s in a transmitted signal. Long sequences of 0s or 1s can cause transmission synchronization problems. This schematic shows the scrambler operation. All adders perform addition modulo *N*, where *N* is the value specified by the Calculation base parameter.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Scramble polynomial parameter, you specify the on or off state for each switch in the scrambler.

To achieve repeatable initial scrambler conditions, you can use one of these optional input ports:

- Select the Reset on nonzero input via port parameter and reset the scrambler with Rst.

- Set the Initial states source parameter to `Input port` and provide the initial states with ISt.

This block can accept input sequences that vary in length during simulation. For more information about sequences that vary in length, see Variable-Size Signal Basics (Simulink).

Ports

Input

`in` — Input data signal

vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal. The input values must be integers from `0` to Calculation base – `1`.

Data Types: `double`

`Rst` — Reset scrambler

scalar

Reset scrambler, specified as a scalar. The scrambler is reset if a nonzero input is applied to the port.

Dependencies

To enable this port, set Initial states source to `Dialog Parameter` and select Reset on nonzero input via port.

`ISt` — Initial states

vector

Initial states of the scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **ISt** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from `0` to Calculation base – `1`.

Dependencies

To enable this port, set Initial states source to `Input port`.

Output

`Out1` — Output scrambled data

vector

Output scrambled data, returned as an N_S-by-1 vector. N_S equals the number of samples in the input signal.

Data Types: `double`

Parameters

`Calculation base` — Calculation base

`4` (default) | nonnegative integer

Calculation base used in the scrambler for modulo operations, specified as a nonnegative integer. The input and output of this block are integers from `0` to **Calculation base** – `1`.

`Scramble polynomial` — Polynomial that defines connections in scrambler

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Polynomial that defines the connections in the scrambler, specified as a character vector, integer vector, or binary vector. The **Scramble polynomial** parameter defines if each switch in the scrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + \ldots$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that has a coefficient of 1. In this case, the order of the scramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

`Initial states source` — Set the source for scrambler initial states

`Dialog Parameter` (default) | `Input port`

- `Dialog Parameter` – Specify scrambler initial states by using the Initial states parameter.
- `Input port` – Specify scrambler initial states by using the ISt port.

`Initial states` — Initial states of scrambler registers

`[0 1 2 3]` (default) | nonnegative integer vector

Initial states of scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **Initial states** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from `0` to Calculation base – `1`.

Dependencies

This parameter is available when Initial states source is set to `Dialog Parameter`.

`Reset on nonzero input via port` — Reset scrambler via input port

off (default) | on

Select this parameter to reset the Scrambler block via input port `Rst`.

Dependencies

This parameter is available when Initial states source is set to `Dialog Parameter`.

See Also

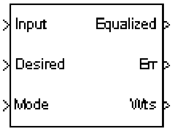
Blocks
Descrambler | PN Sequence Generator

System Objects
`comm.Scrambler`

Introduced before R2006a

Sign LMS Decision Feedback Equalizer

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.

Parameters

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.

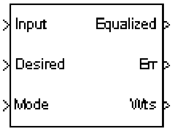
See Also

Sign LMS Linear Equalizer, LMS Decision Feedback Equalizer

Introduced before R2006a

Sign LMS Linear Equalizer

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.

Parameters

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

Introduced before R2006a

SISO Fading Channel

Filter input signal through SISO multipath fading channel
Library: Communications System Toolbox / Channels

Description

The SISO Fading Channel block filters an input signal using a single-input/single-output (SISO) multipath fading channel. This block models both Rayleigh and Rician fading. For processing details, see the Algorithms on page 2-899 section.

Ports

Input

`in` — Input data signal
vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Out1` — Output data signal for fading channel
vector

Output data signal for the fading channel, returned as an N_S-by-1 vector. N_S represents the number of samples in the input signal.

`Gain` — Discrete path gains

matrix

Discrete path gains of the underlying fading process, returned as an N_S-by-N_P matrix.

- N_S represents the number of samples in the input signal.
- N_P represents the number of channel paths.

Dependencies

To enable this port, on the **Main** tab, select Output channel path gains.

`Delay` — Channel filter delay

scalar

Channel filter delay, returned as a scalar.

Dependencies

To enable this port, on the **Main** tab, select Output channel filter delay.

Parameters

Main Tab

Multipath parameters (frequency selectivity)

`Inherit sample rate from input` — Option to inherit the sample rate from input

on (default) | off

Select this parameter to use the sample rate of the input signal when processing. When `Inherit sample rate from input` is selected, the sample rate is N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

`Sample rate (Hz)` — Input signal sample rate

`1` (default) | positive scalar

Input signal sample rate in hertz, specified as a positive scalar. To match the model settings, set the sample rate to N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

Dependencies

This parameter appears when Inherit sample rate from input is not selected.

Data Types: `double`

`Discrete path delays (s)` — Delays for each discrete path

`0` (default) | nonnegative scalar | row vector

Delays for each discrete path in seconds, specified as a nonnegative scalar or row vector.

- When you set **Discrete path delays (s)** to a scalar, the SISO channel is frequency flat.
- When you set **Discrete path delays (s)** to a vector, the SISO channel is frequency selective.

Data Types: `double`

`Average path gains (dB)` — Average gain for each discrete path

`0` (default) | scalar | row vector

Average gain for each discrete path in decibels, specified as a scalar or row vector. **Average path gains (dB)** must have the same size as Discrete path delays (s).

Data Types: `double`

`Normalize average path gains to 0 dB` — Option to normalize average path gains to 0 dB

on (default) | off

Select this parameter to normalize the fading processes so that the total power of the path gains, averaged over time, is 0 dB.

`Fading distribution` — Fading distribution of channel

`Rayleigh` (default) | `Rician`

Select the fading distribution of the channel, either `Rayleigh` or `Rician`.

`K-factors` — K-factor of Rician fading channel

`3` (default) | positive scalar | row vector

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of positive-valued elements. N_P equals the value of the Discrete path delays (s) parameter.

- If you set **K-factors** to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of **K-factors**. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set **K-factors** to a row vector, the discrete path corresponding to a positive element of the **K-factors** 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 **K-factors** vector is a Rayleigh fading process.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path Doppler shifts (Hz)` — Doppler shifts for line-of-sight components

`0` (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path Doppler shifts (Hz)** to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set **LOS path Doppler shifts (Hz)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of **LOS path Doppler shifts (Hz)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path initial phases (rad)` — Initial phases for line-of-sight components

`0` (default) | scalar | row vector

Initial phases for the line-of-sight component of the Rician fading channel in radians, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path initial phases (rad)** to a scalar, it is the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set **LOS path initial phases (rad)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the

elements of **LOS path initial phases (rad)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

Doppler parameters (time dispersion)

`Maximum Doppler shift (Hz)` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

Maximum Doppler shift (Hz) must be smaller than (f_s/10)/f_c for each path. f_s is the sampling rate at the input to the SISO Fading Channel block. f_c is the cutoff frequency factor of the path. For more information, see Cutoff Frequency Factor on page 2-608.

Data Types: `double`

`Doppler spectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this parameter is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.
- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter applies when Maximum Doppler shift (Hz) is greater than zero.

Other parameters

`Initial seed` — Random number generator initial seed
73 (default) | nonnegative integer

Random number generator initial seed for this block, specified as a nonnegative integer.

`Output channel path gains` — Option to output channel path gains
off (default) | on

Select this parameter to add the Gain output port to the block and output the channel path gains of the underlying fading process.

`Output channel filter delay` — Option to output channel filter delay
off (default) | on

Select this parameter to add the Delay output port to the block and output the channel filter delay of the underlying fading process.

`Simulate using` — Compilation type
`Interpreted execution` (default) | `Code generation`

Compilation type, specified as `Interpreted execution` or `Code generation`.

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

Visualization Tab

`Channel visualization` — Select the channel visualization
`Off` (default) | `Impulse response` | `Frequency response` | `Doppler spectrum` | `Impulse and frequency responses`

Select the channel visualization: `Off`, `Impulse response`, `Frequency response`, `Doppler spectrum`, or `Impulse and frequency responses`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

`Percentage of samples to display` — Percentage of samples to display
`25%` (default) | `10%` | `50%` | `100%`

Select the percentage of samples to display: `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This parameter appears when Channel visualization is `Impulse response`, `Frequency response`, or `Impulse and frequency responses`.

`Path for Doppler spectrum display` — Path for which Doppler spectrum is displayed
`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_P, where N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter appears when Channel visualization is `Doppler spectrum`.

Algorithms

The fading process for the SISO channel is described in Methodology for Simulating Multipath Fading Channels.

Cutoff Frequency Factor

The following information explains how the cutoff frequency factor, f_c, is determined for different Doppler spectrum types:

- For any Doppler spectrum type other than Gaussian and BiGaussian, f_c equals `1`.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation•sqrt(2•log(2))`.
- For a `doppler('BiGaussian')` spectrum type:

- If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
- If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)•sqrt(2•log(2))`.
- If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
- In all other cases, f_c equals `1`.

References

[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.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

See Also

Blocks
AWGN Channel | MIMO Fading Channel

Functions
`doppler`

System Objects
`comm.MIMOChannel`

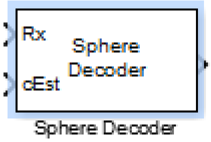
Topics

Channel Visualization

Introduced in R2017b

Sphere Decoder

Decode input using a sphere decoder



Library

MIMO

Description

This block decodes the symbols sent over `Nt` antennas using the sphere decoding algorithm.

Data Type

For information about the data types each block port supports, see the “Supported Data Type” on page 2-903 table on this page. The output signal inherits the data type from the inputs.

Algorithm

This block implements the algorithm, inputs, and outputs described on the `comm.SphereDecoder` System object block reference page. The object properties correspond to the block parameters.

Parameters

Signal constellation

Specify the number of points in the signal constellation to which the bits are mapped. This value must be a complex column vector. The length of the vector must be a power of two. The block uses the same constellation for each transmit antenna. The default setting is a QPSK constellation with an average power of 1.

Bit mapping per constellation point

Specify the bit mapping that the block uses for each constellation point. This value must be a numerical matrix. he matrix size must be [ConstellationLength bitsPerSymbol], where ConstellationLength represents the length of the **Signal constellation** parameter value and bitsPerSymbol represents the number of bits that each symbol encodes. The default matrix size is [0 0; 0 1; 1 0; 1 1], which matches the default value of the **Signal constellation** property.

Initial search radius

Specify the initial search radius for the decoding algorithm as `Infinity` or `ZF solution`.

When you select `Infinity`, the block sets the initial search radius to Inf. When you select `ZF solution`, the block 'sets the initial search radius to the zero-forcing solution. The zero-forcing solution is calculated by 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.

Decision method

Specify the decoding decision method as `Soft` or `Hard`. When you select `Soft` the block outputs log-likelihood ratios (LLRs), or soft bits. When you select set to `Hard`, the block converts the soft LLRs to bits. The hard decision output logical array follows the mapping of a `0` for a negative LLR and `1` for all other values.

Simulation using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

Supported Data Type

Port	Supported Data Types
Rx	• Double-precision floating point
cEst	• Double-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Boolean (Hard-decision method)

Limitations

- The output LLR values are not scaled by the noise variance. For coded links employing iterative coding (LDPC or turbo) or MIMO OFDM with Viterbi decoding, the output LLR values should be scaled by the channel state information to achieve better performance.

Algorithms

This block implements the algorithm, inputs, and outputs described on the Sphere Decoder System object reference page. The object properties correspond to the block parameters.

See Also

OSTBC Encoder | OSTBC Combiner | `comm.SphereDecoder`

Introduced in R2013b

Squaring Timing Recovery

Recover symbol timing phase using squaring method

Note Squaring Timing Recovery has been removed. Use the Symbol Synchronizer block instead.

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.

Parameters

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

$$-\frac{1}{2\pi}\arg\left(\sum_{m=0}^{\mathrm{LN}\text{-}1}\left|x_{m+1}\right|^2\exp(\text{-j}2\pi m/\mathrm{N})\right)$$

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.

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

Symbol Synchronizer

Introduced before R2006a

SSB AM Demodulator Passband

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.

Parameters

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

Introduced before R2006a

SSB AM Modulator Passband

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 Transform 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 $\mathrm{H}\{\}$ 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`.

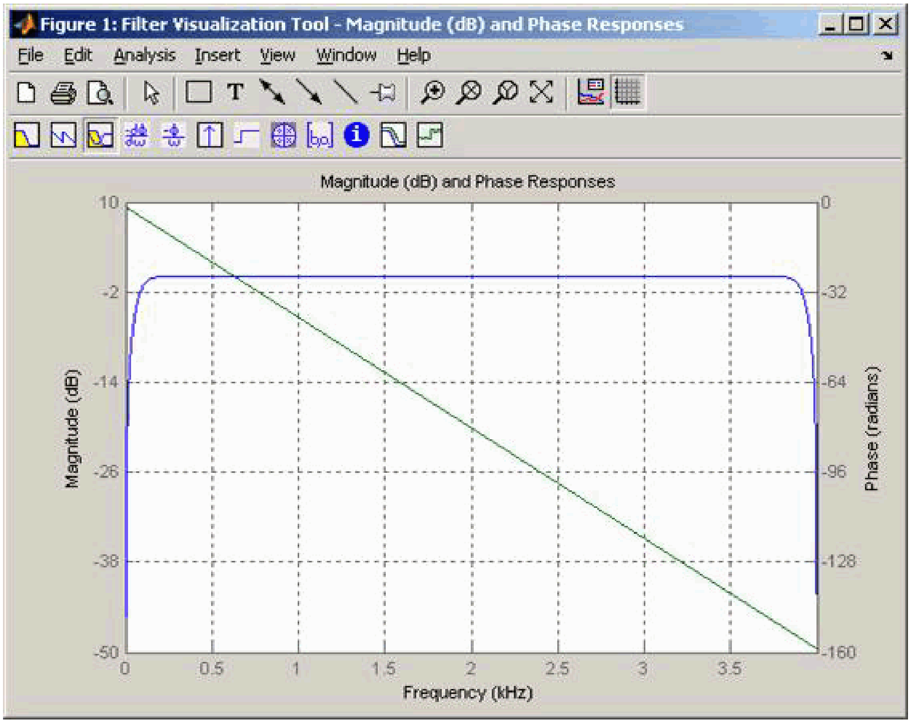


Figure 1: Filter Visualization Tool - Magnitude (dB) and Phase Responses
File Edit Analysis Insert View Window Help
Magnitude (dB) and Phase Responses
Magnitude (dB)
10
-2
-14
-26
-38
-50
Phase (radians)
0
-32
-64
-96
-128
-160
0
0.5
1
1.5
2
2.5
3
3.5
Frequency (kHz)

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.

Parameters

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`

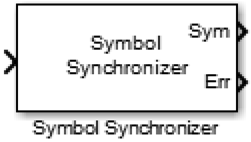
References

[1] Peebles, Peyton Z, Jr. *Communication System Principles*. Reading, Mass.: Addison-Wesley, 1976.

Introduced before R2006a

Symbol Synchronizer

Correct symbol timing clock skew



Library

Synchronization

Description

The Symbol Synchronizer block corrects for symbol timing clock skew for PAM, PSK, or QAM modulation schemes. The block accepts a single input port. To obtain a normalized estimate of the timing error, select the **Normalized timing error output port** check box. The block accepts a complex input signal and returns a complex output signal and a real timing error estimate. The Sym output is variable-size with maximum dimensions of $\left\lceil \frac{N_{samp}}{N_{sps}} \times 1.1 \right\rceil$, where N_{samp} is the number of samples and N_{sps} is the samples per symbol. Output that would exceed this limit is truncated. The Err output has the same dimensions as the input signal.

Parameters

Modulation type

Specify the modulation type as `PAM/PSK/QAM`, or `OQPSK`. The default setting is `PAM/PSK/QAM`.

Timing error detector

Specify the timing error detector as `Zero-Crossing (decision-directed)`, `Gardner (non-data-aided)`, `Early-Late (non-data-aided)`, or `Mueller-`

`Muller (decision-directed)`. The default setting is `Zero-Crossing (decision-directed)`.

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar greater than or equal to 2. The default setting is 2.

Damping factor

Specify the damping factor of the loop filter as a positive real finite scalar. The default setting is 1. This parameter is tunable.

Normalized loop bandwidth

Specify the normalized loop bandwidth as a real scalar between 0 and 1. The bandwidth is normalized by the sample rate of the symbol synchronizer block. The default setting is 0.01. This parameter is tunable.

Note Set **Normalized loop bandwidth** to less than `0.1` to ensure that the symbol synchronizer locks.

Detector gain

Specify the detector gain as a real positive finite scalar. The default setting is 2.7. This parameter is tunable.

Normalized timing error output port

Select this check box to provide the normalized timing error to an output port. The default for this parameter is selected.

Simulate using

Select the type of simulation to run.

- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

The default setting is `Code generation`.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.SymbolSynchronizer` reference page. The object properties correspond to the block parameters.

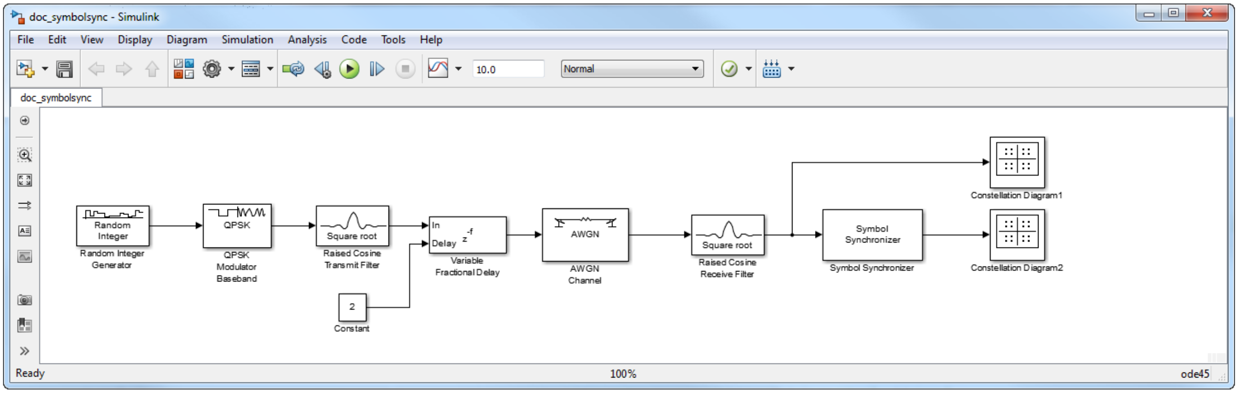
For OQPSK, the in-phase and quadrature signal components are first aligned (as in QPSK) using a buffer (state) to cache the last half symbol of the previous input. After initial alignment, the remaining synchronization processing is QPSK.

Examples

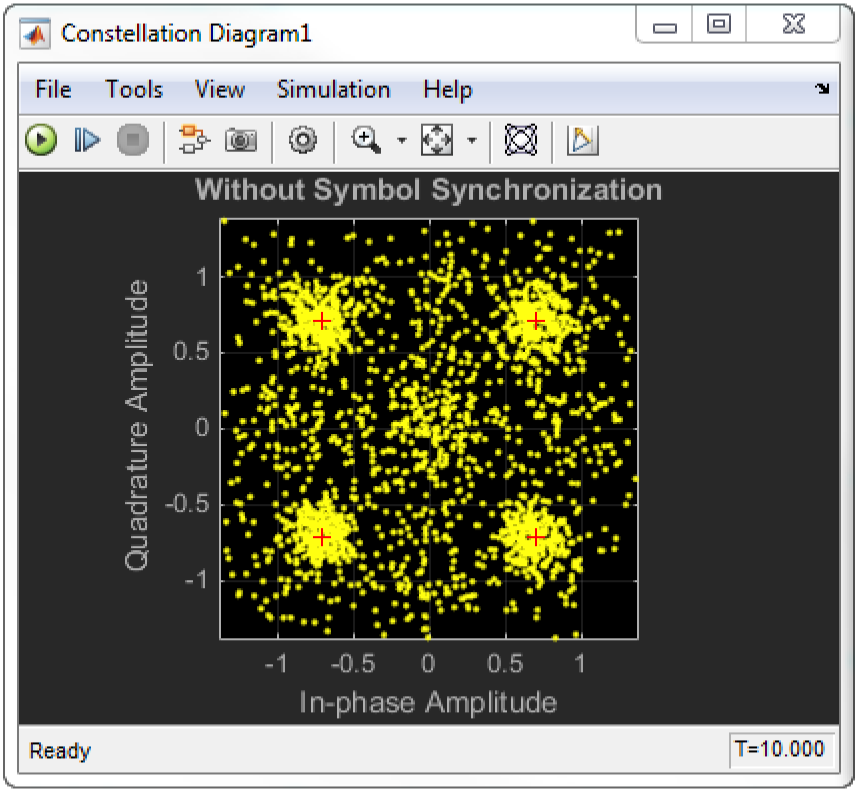
Correct QPSK Signal for Timing Offset

Correct for a fixed symbol timing offset on a noisy QPSK signal.

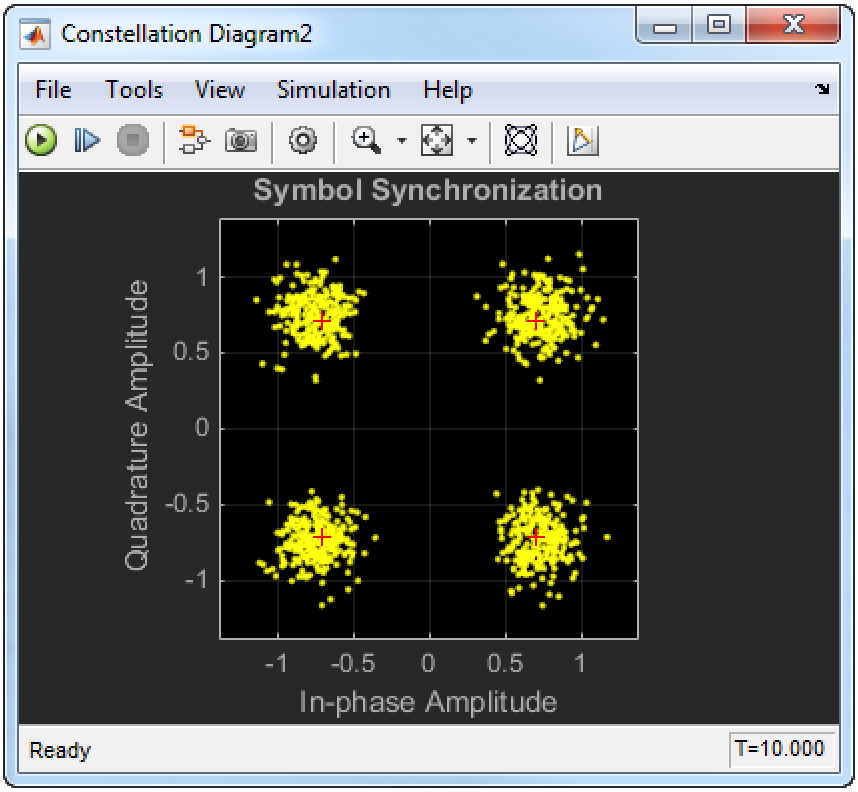
Open the doc_symbolsync model.



Run the model. The Variable Fractional Delay block is used to introduce a timing error of 2 samples. As the Raised Cosine Transmit Filter is configured to have 4 **Output samples per symbol**, the timing delay is 0.5 symbols. The constellation diagram without symbol synchronization shows that the QPSK symbols cannot be successfully resolved.



The constellation diagram for the signal after the synchronizer shows that the QPSK symbols can now be resolved.

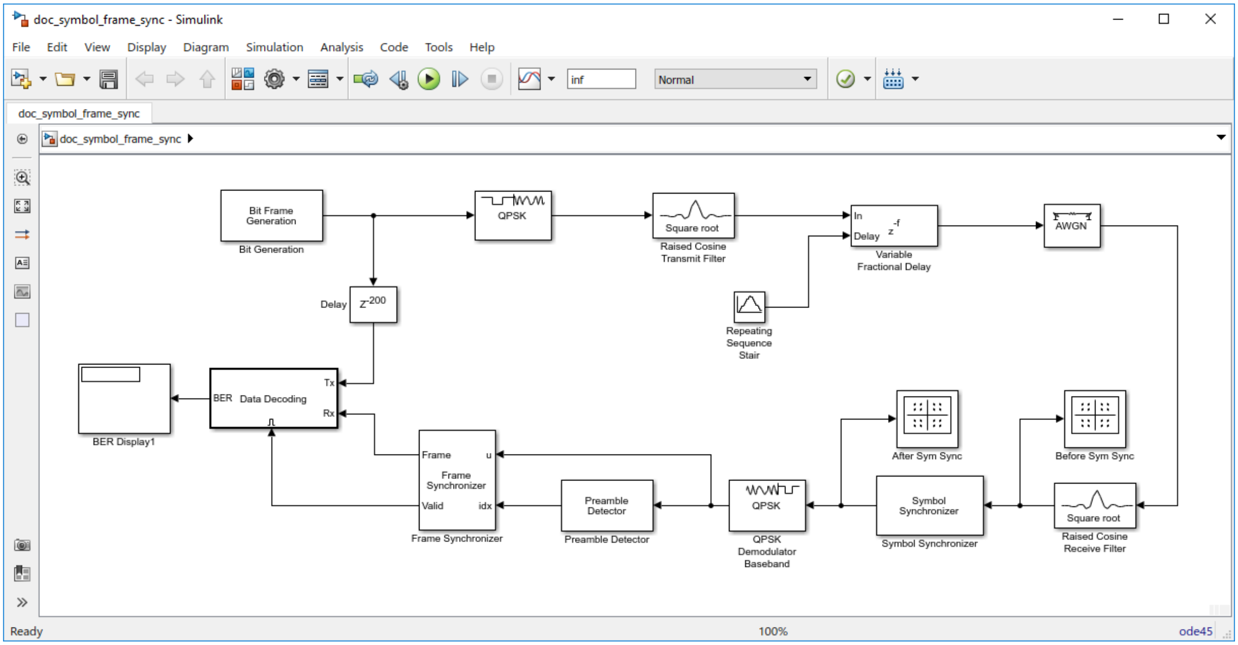


Try to experiment with the model by changing the delay and the **Timing error detector** algorithm.

Symbol and Frame Synchronization

Recover frame synchronization from a QPSK system suffering from a variable timing error.

Load the doc_symbol_frame_sync model.

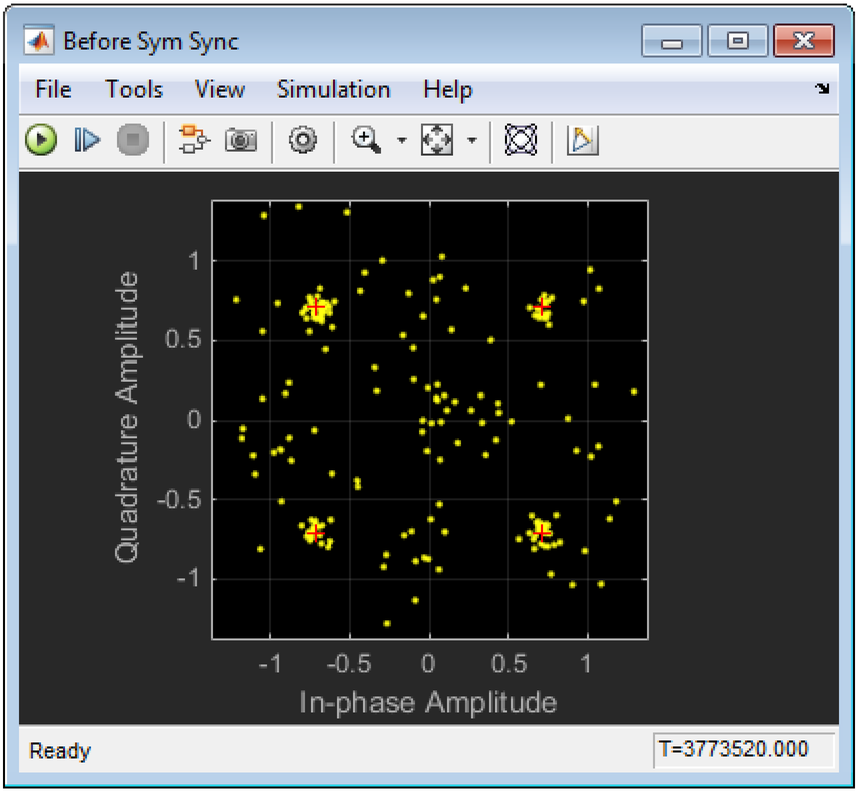


Run the model. The Before Sym Sync and After Sym Sync constellation diagram blocks show the effects of a timing error on the transmitted QPSK constellation. This timing error is introduced as a variable delay that ranges from 0 to 0.9 samples. The Symbol Synchronizer block corrects for clock skew between a single-carrier transmitter and receiver, aligning the output data with a valid clock reference. Depending on the size of the timing error, the output dimensions of the symbol synchronizer vary. In this example, the symbol synchronizer returns a vector containing 99, 100, or 101 samples for a 100-sample input vector.

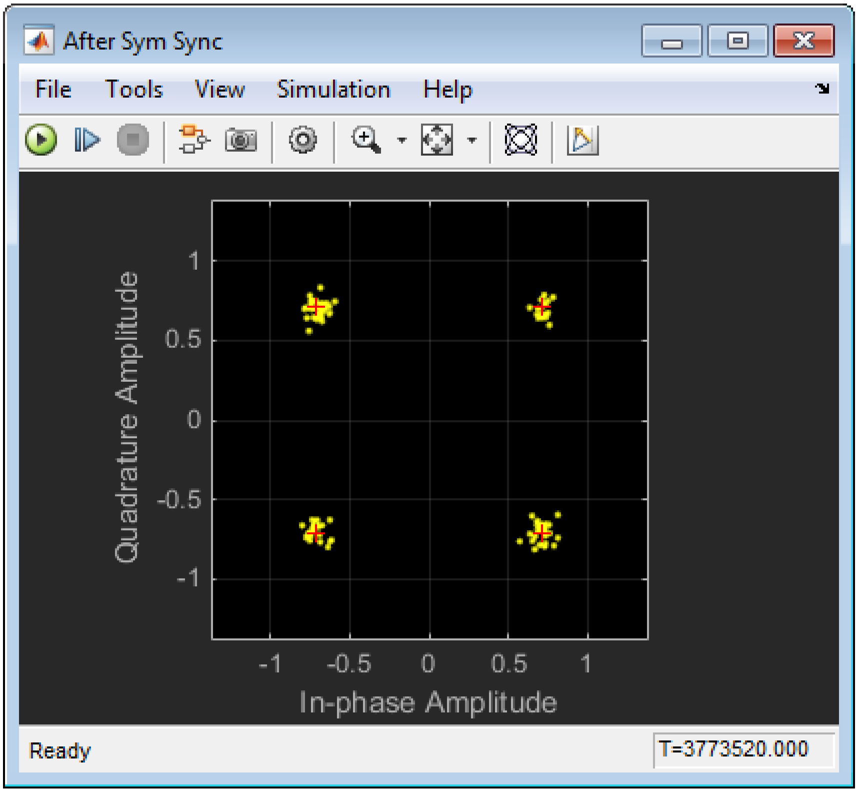
The bit error rate (BER) is calculated by the Data Decoding block. Within that block, the input data is regenerated rather than being taken from the Bit Generation block because the BER is calculated only for valid frames. The Preamble Detector finds the start of packet in the demodulated bit stream. The Frame Synchronizer uses this start index to align the bit stream along correct frame boundaries and also provides a valid frame indicator.

The signal is recovered correctly, as seen by a BER of zero (or less than 10^{-5}) for the 20 dB signal-to-noise ratio used here.

The constellation diagram before the symbol synchronizer shows the effects of the variable timing error. Because the timing error varies over time, the constellation oscillates between corrupted and clean states.



The After Sym Sync constellation diagram shows that the synchronizer removes the effects of the variable timing error.



Experiment with the model by commenting through (Ctrl-Shift-Y) the Symbol Synchronizer block and setting the decimation factor of the Raised Cosine Receive Filter block to 2. Without symbol synchronization, the BER increases significantly because the timing error corrupts the received signal to the point that bit errors occur.

Supported Data Types

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

Port	Supported Data Types
Symbol Output	• Double-precision floating point • Single-precision floating point
Normalized Timing Error	• Double-precision floating point • Single-precision floating point

See Also

- Preamble Detector
- Carrier Synchronizer
- `comm.SymbolSynchronizer`

Selected Bibliography

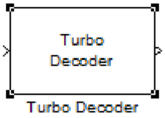
[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 434–513.

[2] Mengali, Umberto and Aldo N. D’Andrea. *Synchronization Techniques for Digital Receivers*. New York: Plenum Press, 1997.

Introduced in R2015a

Turbo Decoder

Decode input signal using parallel concatenated decoding scheme



Library

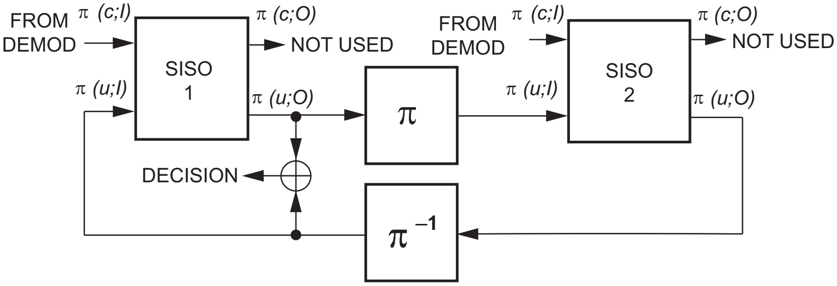
Convolutional sublibrary of Error Detection and Correction

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.

Parameters

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).

Source of interleaver indices

Specify the source of the interleaver indices as `Property` or `Input port`.

When you set this parameter to `Property`, the block uses the **Interleaver indices** parameter to specify the interleaver indices.

When you set this parameter to `Input port`, the block uses the secondary input port, IntrInd, to specify the interleaver indices.

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`. The block iterates and provides updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the block is the hard-decision output of the final LLR update.

Simulate using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

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

`comm.TurboDecoder`

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.

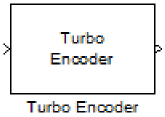
[3] Schlegel, Christian B. and Lance C. Perez. *Trellis and Turbo Coding*, IEEE Press, 2004.

[4] 3GPP TS 36.212 v9.0.0, *3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Introduced in R2011b

Turbo Encoder

Encode binary data using parallel concatenated encoding scheme



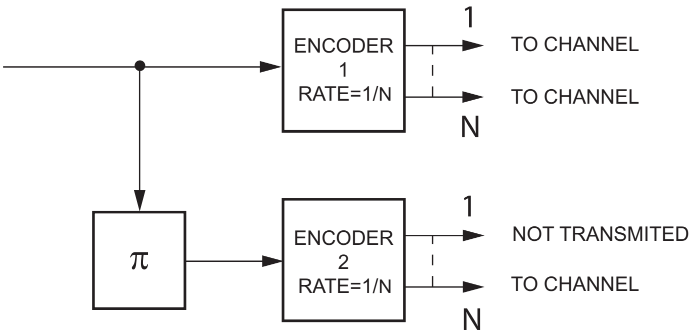
Library

Convolutional sublibrary of Error Detection and Correction

Description

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

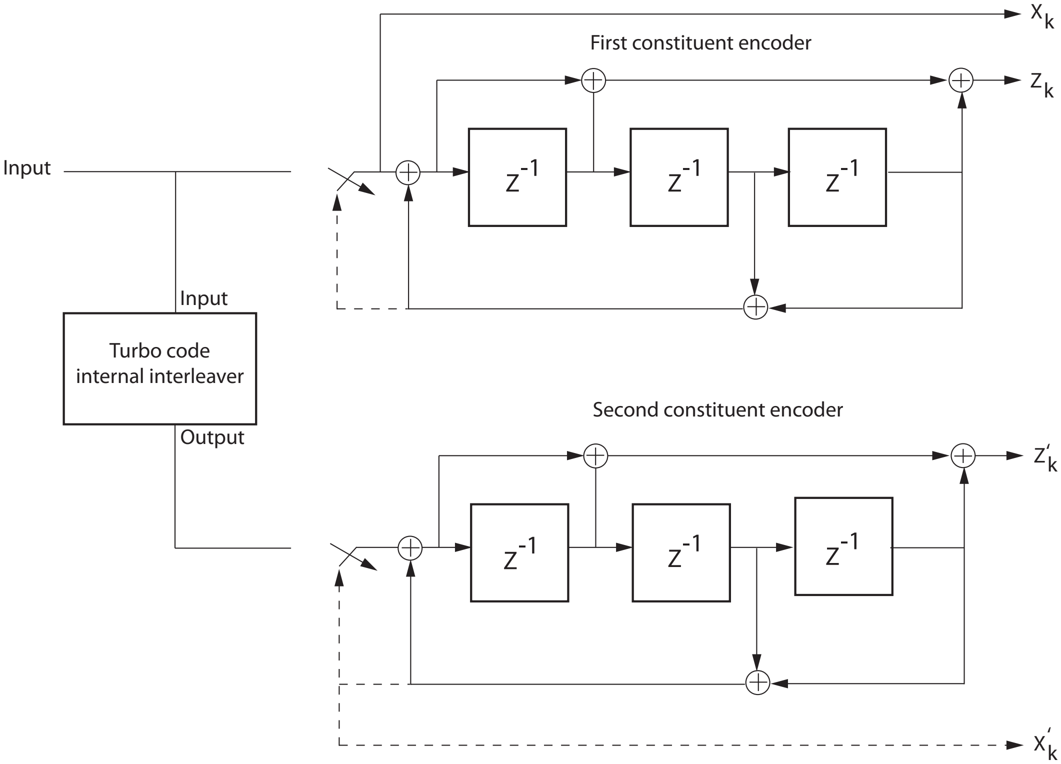
L = encoder input length

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 (three systematic bits and three parity bits) are the output tail bits from the first constituent encoder. The second group of six bits (three systematic bits and three parity bits) are the output tail bits from the second constituent encoder.

Due to the tail bits, the encoder output code rate is slightly less than 1/3.

Parameters

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).

Source of interleaver indices

Specify the source of the interleaver indices as `Property` or `Input port`.

When you set this parameter to `Property`, the block uses the **Interleaver indices** parameter to specify the interleaver indices.

When you set this parameter to `Input port`, the block uses the secondary input port, IntrInd, to specify the interleaver indices.

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.

Simulate using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

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

`comm.TurboEncoder`

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 Access (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Introduced in R2011b

Uniform Noise Generator

Generate uniformly distributed noise between upper and lower bounds

Note Uniform Noise Generator will be removed in a future release. Use the MATLAB Function block and `rand` function instead.

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.

Parameters

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)

Introduced before R2006a

Unipolar to Bipolar Converter

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)

Parameters

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** (Simulink) 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.

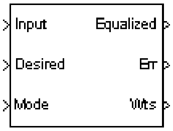
Pair Block

Bipolar to Unipolar Converter

Introduced before R2006a

Variable Step LMS Decision Feedback Equalizer

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.

Parameters

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.

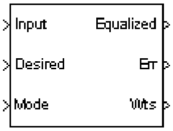
See Also

Variable Step LMS Linear Equalizer, LMS Decision Feedback Equalizer

Introduced before R2006a

Variable Step LMS Linear Equalizer

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.

Parameters

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.

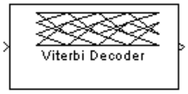
See Also

Variable Step LMS Decision Feedback Equalizer, LMS Linear Equalizer

Introduced before R2006a

Viterbi Decoder

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” (Simulink).

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 2-961 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
5	Third most confident one

Input Value	Interpretation
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.

As a general estimate, the **Traceback depth** value is approximately two to three times (k – 1)/(1 – r), where k is the constraint length of the code and r is the code rate [7]. For example:

- A rate `1/2` code has a **Traceback depth** of 5(k – 1).
- A rate `2/3` code has a **Traceback depth** of 7.5(k – 1).
- A rate `3/4` code has a **Traceback depth** of 10(k – 1).
- A rate `5/6` code has a **Traceback depth** of 15(k – 1).

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:

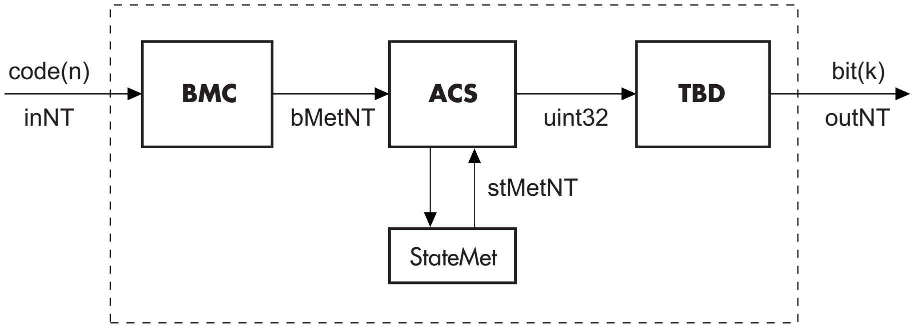
- Sets the all-zeros state metric to zero.
- Sets all other state metrics to the maximum value.
- Sets the traceback memory to zero.

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`.

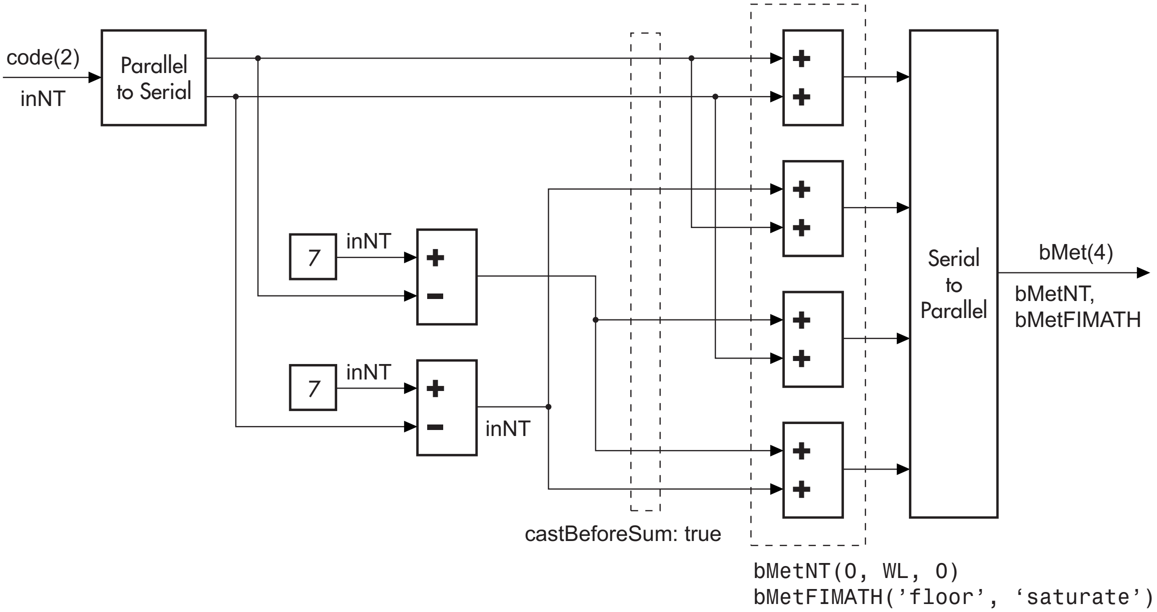
The reset port supports `double` or `boolean` typed signals.

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.

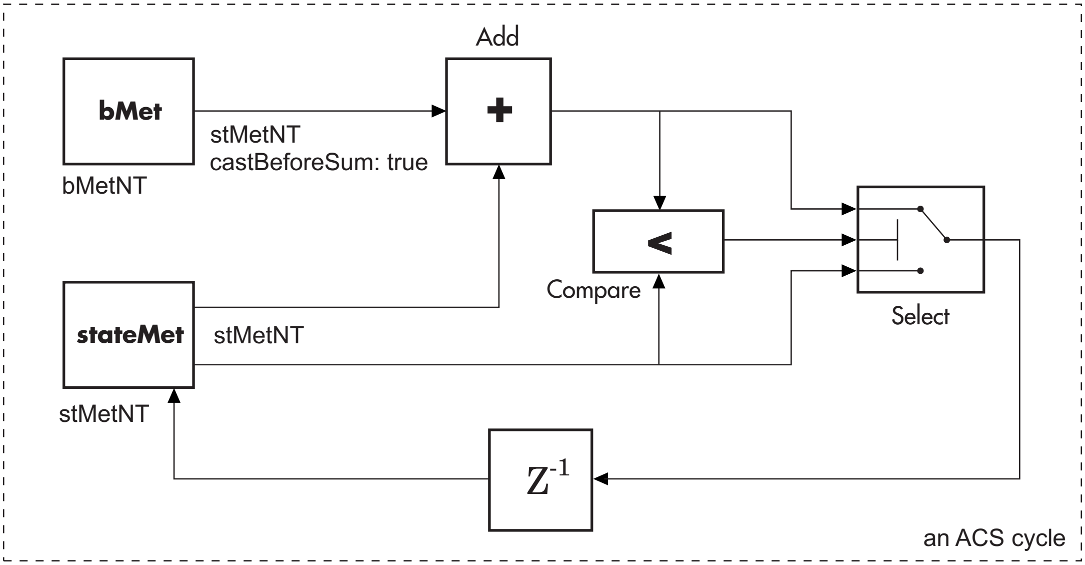


As an example of a BMC diagram, a 1/2 rate, *nsdec* = 3 signal flow would be as follows.



$$WL = nsdec + n - 1$$
$$n = 2 \Rightarrow WL = 4$$

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.

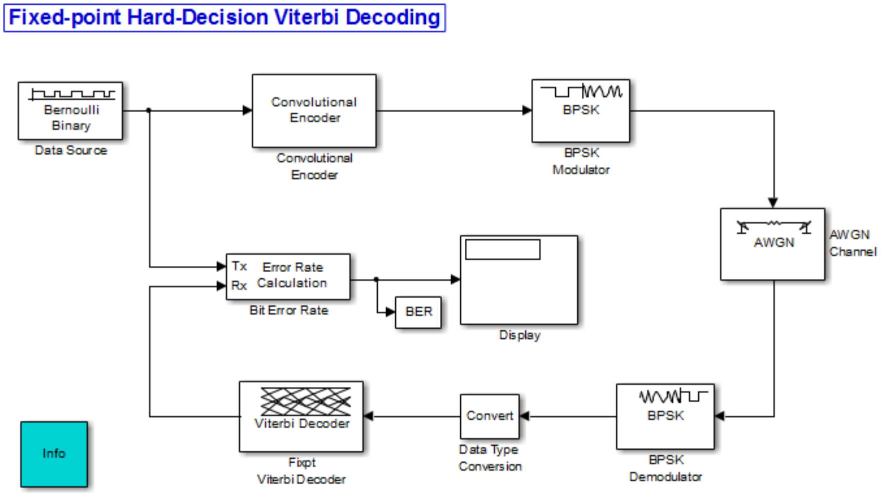
Puncture Pattern Examples

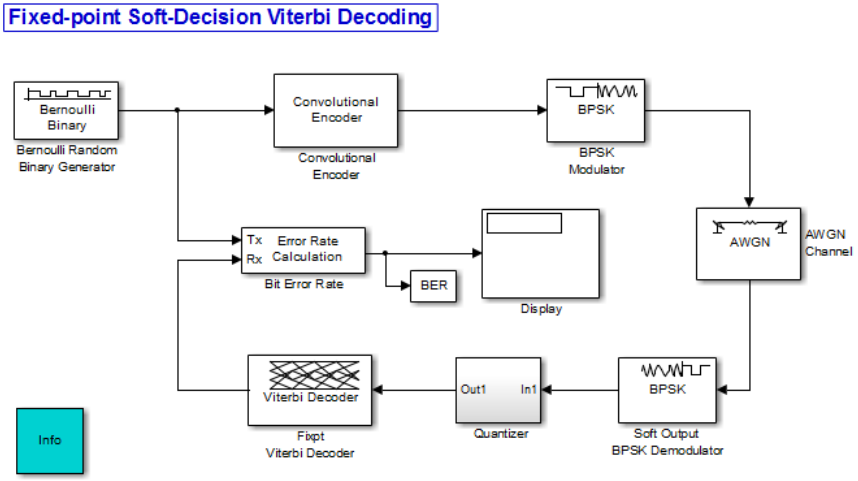
For some commonly used puncture patterns for specific rates and polynomials, see the last three references.

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\comm\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.

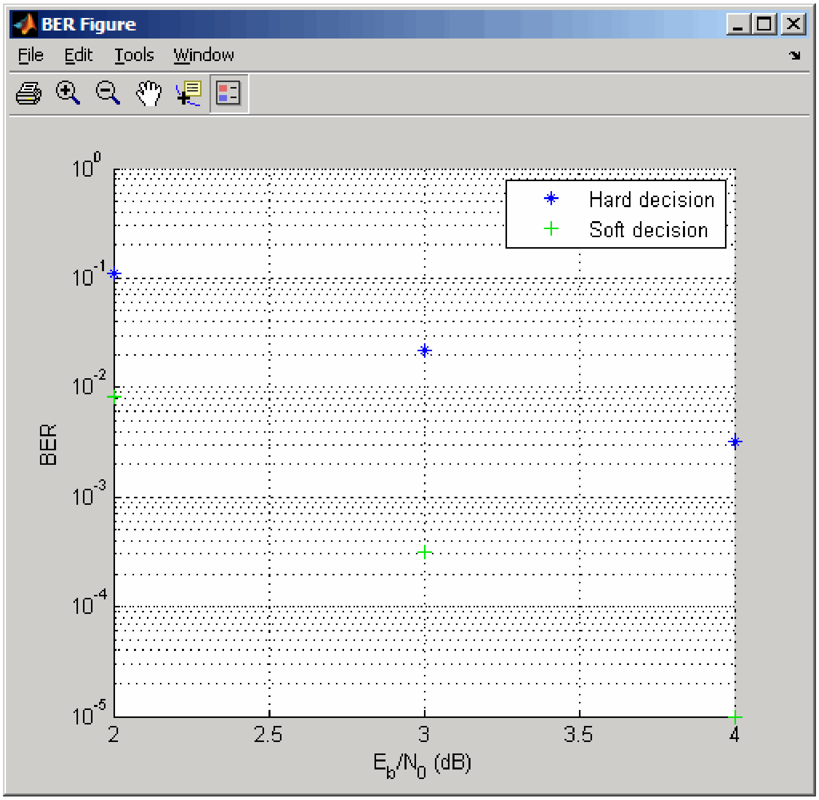
Upon simulating the model, note that the double-precision and fixed-point BER results are the same. They are the same because the fixed-point parameters for the model have been selected to avoid any loss of precision while still being most efficient.

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 is generated.



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.

Parameters

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.

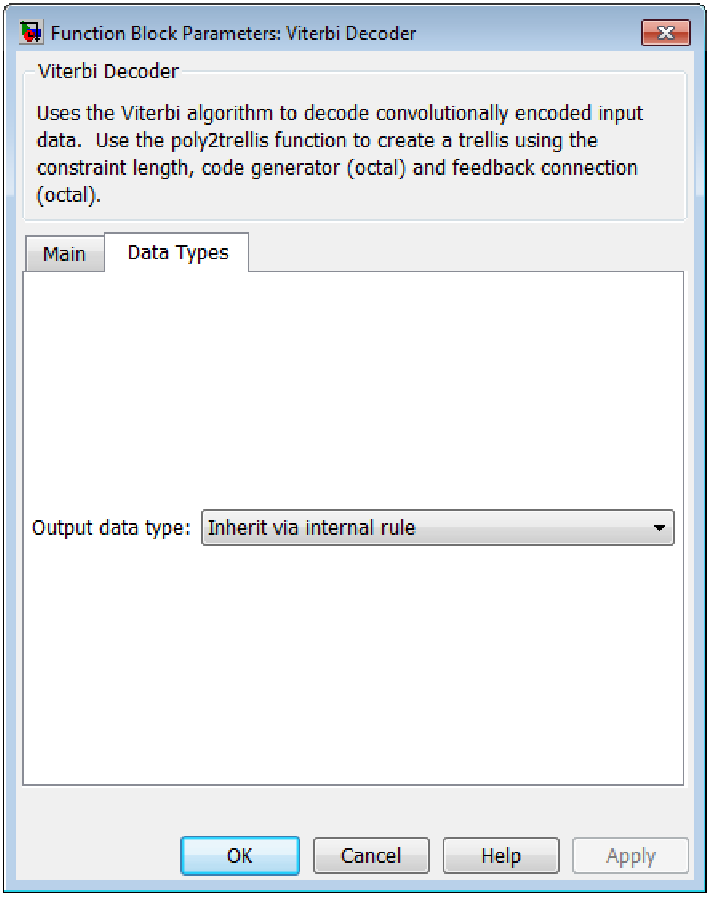
Enable reset input port

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



Function Block Parameters: Viterbi Decoder

Viterbi Decoder

Uses the Viterbi algorithm to decode convolutionally encoded input data. Use the poly2trellis function to create a trellis using the constraint length, code generator (octal) and feedback connection (octal).

Output data type: Inherit via internal rule

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

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Viterbi Decoder in the HDL Coder documentation.

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.

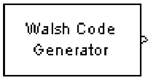
[6] Begin, G., et.al., "Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Transactions on Communications*, Vol. 38, No. 11, pp 1922–1928, Nov. 1990.

[7] Moision, B., "A Truncation Depth Rule of Thumb for Convolutional Codes," *Information Theory and Applications Workshop*, pp. 555–557, 2008.

Introduced before R2006a

Walsh Code Generator

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, ... , *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, ... , *N* - 1.
- $$W_j W_k^T = \begin{cases} 0 & j \neq k \\ N & j = k \end{cases}$$
- Each code W*j* 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, ... , *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.

Parameters

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

The time between each sample of the output signal. Specify as a nonnegative real scalar.

Samples per frame

The number of samples in one column of the output signal. If **Samples per frame** is greater than the **Code length**, the code is cyclically repeated. Specify as a positive integer scalar.

Note The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

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

Introduced before R2006a

Windowed Integrator

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 2-968 table on this page.

Parameters

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.

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" (DSP System Toolbox) or "Rounding Mode: Simplest" (Fixed-Point Designer).

Saturate on integer overflow

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" (DSP System Toolbox) 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 **Saturate on integer overflow** 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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” (DSP System Toolbox) and “Multiplication Data Types” (DSP System Toolbox) 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

Port	Supported Data Types
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)

Introduced before R2006a

3

System Objects — Alphabetical List

comm.ACPR System object

Package: comm

Adjacent Channel Power Ratio measurements

Description

The ACPR System object measures adjacent channel power ratio (ACPR) of an input signal.

To measure adjacent channel power:

1. Define and set up your adjacent channel power object. See "Construction" on page 3-2.
2. Call `step` to measure the adjacent channel power ratio according to the properties of `comm.ACPR`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 MeasurementFilterSource 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 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 `42.5`dB 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

reset	Reset states of ACPR measurement object
step	Adjacent Channel Power Ratio measurements

Common to All System Objects	
`release`	Allow System object property value changes

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 15],5000,1);
y = qammod(x,16);
```

Upsample 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

```
acpr = comm.ACPR(...
    'SampleRate', 3.84e6*8,...
    'MainChannelFrequency', 0,...
    'MainMeasurementBandwidth', 3.84e6,...
    'AdjacentChannelOffset', [-5e6 5e6],...
    'AdjacentMeasurementBandwidth', 3.84e6,...
    'MainChannelPowerOutputPort', true,...
    'AdjacentChannelPowerOutputPort', true);
[ACPR,mainChnlPwr,adjChnlPwr] = acpr(yPulse)
```

```
ACPR = 1×2

  -14.3659  -14.3681


mainChnlPwr = 38.8668

adjChnlPwr = 1×2

   24.5010   24.4988
```

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CCDF` | `comm.EVM` | `comm.MER`

Introduced in R2012a

reset

System object: comm.ACPR
Package: comm

Reset states of ACPR measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ACPR` object, `H`.

step

System object: comm.ACPR
Package: comm

Adjacent Channel Power Ratio measurements

Syntax

```
A = step(H,X)
[A,MAINPOW] = step(H,X)
[A,ADJPOW] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To adaptively adjust gain for constant signal-level output:

1. Define and set up your automatic gain controller object. See “Construction” on page 3-15.
2. Call `step` to adaptively adjust gain and achieve a constant signal level at the output according to the properties of `comm.AGC`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.AGC` creates an 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

`AdaptationStepSize`

Step size for gain updates

Specify the step size as a real positive scalar. The default is `0.01`. Increasing the step size permits the AGC to respond more quickly to changes in the input signal level but increases variation in the output signal level during steady-state operation.

`DesiredOutputPower`

Target output power level

Specify the desired output power level as a real positive scalar. The power is measured in Watts referenced to 1 ohm . The default is `1`.

`AveragingLength`

Length of the averaging window

Specify the length of the averaging window in samples as a positive integer scalar. The default is `100`.

Note If you use the AGC with higher order QAM signals, you might need to reduce the variation in the gain during steady-state operation. Inspect the scatter plot at the output of the AGC and increase the averaging length as needed. An increase in `AveragingLength` reduces execution speed.

`MaxPowerGain`

Maximum power gain in decibels

Specify the maximum gain of the AGC in decibels as a positive scalar. The default is `60`.

If the AGC input signal power is very small, the AGC gain will be very large. This can cause problems when the input signal power suddenly increases. Use `MaxPowerGain` to avoid this by limiting the gain that the AGC applies to the input signal.

Methods

reset	Reset internal states of automatic gain controller
step	Apply adaptive gain to input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Vary AGC Averaging Length

Apply different AGC averaging lengths to QAM modulated signals. Compare the variance and plot of the signals after AGC is applied.

Initialize three AGC Sytem objects with average window length set to 10, 100, and 1000 samples.

```
agc1 = comm.AGC('AveragingLength',10);
agc2 = comm.AGC('AveragingLength',100);
agc3 = comm.AGC('AveragingLength',1000);
```

Generate M-QAM modulated and raised cosine pulse shaped packetized data.

```
M = 16;
d = randi([0 M-1],1000,1);
s = qammod(d,M);
x = 0.1*s;
pulseShaper = comm.RaisedCosineTransmitFilter;
y = awgn(pulseShaper(x),inf);
```

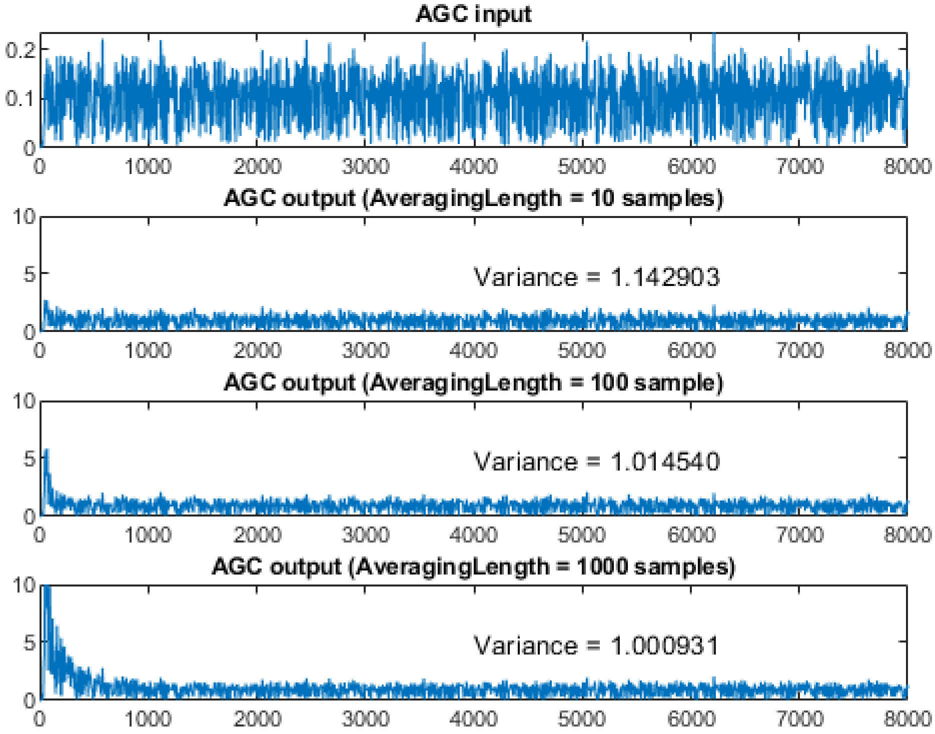
Apply AGC.

```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot and compare the signals.

```
figure(1)
subplot(4,1,1)
```

```
plot(abs(y))
title('AGC input')
subplot(4,1,2)
plot(abs(r1))
axis([0 8000 0 10])
title('AGC output (AveragingLength = 10 samples)')
text(4000, 5, sprintf('Variance = %f',var(r1(3000:end))))
subplot(4,1,3)
plot(abs(r2))
axis([0 8000 0 10])
title('AGC output (AveragingLength = 100 sample)')
text(4000, 5, sprintf('Variance = %f',var(r2(3000:end))))
subplot(4,1,4)
plot(abs(r3))
axis([0 8000 0 10])
title('AGC output (AveragingLength = 1000 samples)')
text(4000, 5, sprintf('Variance = %f',var(r3(3000:end))))
```



As the averaging length increases the variance at the output of the AGC decreases.

Vary AGC Maximum Gain

Apply different AGC maximum gain level to QPSK modulated signals. With the maximum gain too small the signal after AGC will not reach desired amplitude. With the maximum gain too large the signal after AGC will overshoot and can saturate resulting in signal loss at the start of received packets. With the maximum gain at the optimal level, the signal after AGC reaches the desired amplitude without signal loss due to saturation. Compare the plot of the signals after AGC is applied.

Initialize three AGC Sytem objects with maximum gain values set to 10 dB, 20 dB, and 30 dB.

```
agc1 = comm.AGC('MaxPowerGain', 10);
agc2 = comm.AGC('MaxPowerGain', 20);
agc3 = comm.AGC('MaxPowerGain', 30);
```

Generate QPSK modulated and raised cosine pulse shaped packetized data.

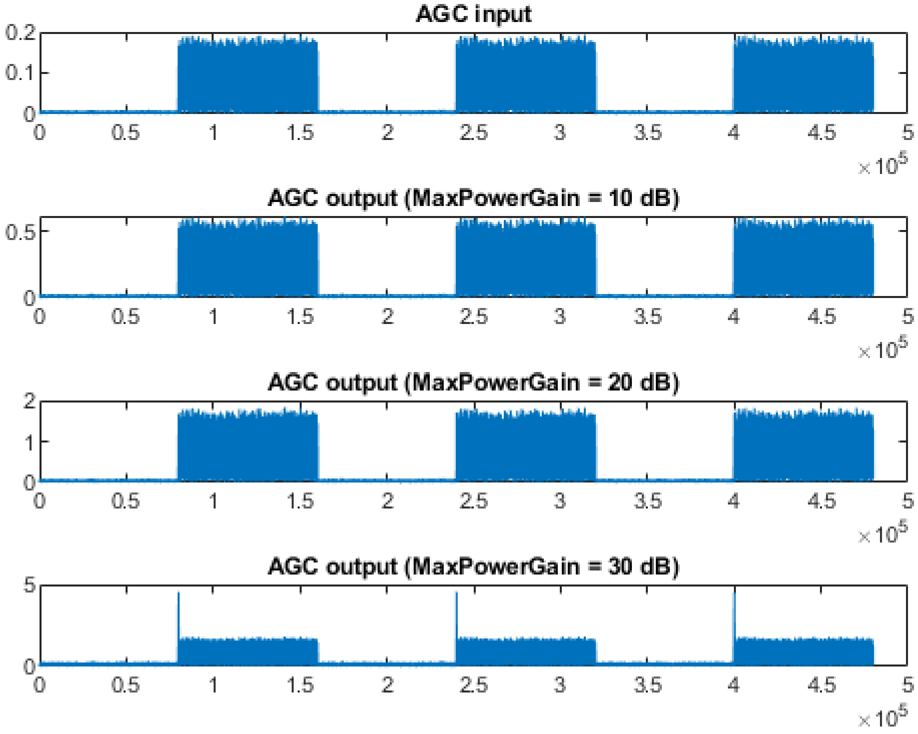
```
M = 4;
pktLen = 10000;
d = randi([0 M-1],pktLen,1);
s = pskmod(d,M,pi/4);
x = repmat([zeros(pktLen,1); 0.3*s],3,1);
pulseShaper = comm.RaisedCosineTransmitFilter;
y = awgn(pulseShaper(x),50);
```

Apply AGC and plot.

```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot and compare the signals.

```
figure(1)
subplot(4,1,1)
plot(abs(y))
title('AGC input')
subplot(4,1,2)
plot(abs(r1))
title('AGC output (MaxPowerGain = 10 dB)')
subplot(4,1,3)
plot(abs(r2))
title('AGC output (MaxPowerGain = 20 dB)')
subplot(4,1,4)
plot(abs(r3))
title('AGC output (MaxPowerGain = 30 dB)')
```



This plot compares the signal input to AGC and output from AGC with various maximum gain levels. With the maximum gain set to 10dB, the AGC output cannot reach the desired output power level. With the maximum gain set to 20dB, the AGC output reaches the desired level without saturating. With the maximum gain set to 30dB, the AGC output overshoots risking signal saturation and data loss. Between packets there is only noise in the input signal.

As shown in the plots, with packet transmissions there may be extended periods when no data is received. This results in AGC increasing to the maximum gain setting. If a packet arrives when the AGC gain is too high, the output power overshoots until the AGC can respond to the change in the input power level and reduce its gain.

Vary AGC Step Size

Apply different AGC step sizes to QPSK modulated signals. Compare the signals after AGC is applied.

Initialize three AGC Sytem objects with step sizes set to 1e-1, 1e-3, and 1e-4.

```
agc1 = comm.AGC('AdaptationStepSize',1e-1);
agc2 = comm.AGC('AdaptationStepSize',1e-3);
agc3 = comm.AGC('AdaptationStepSize',1e-4);
```

Generate QPSK modulated data with raised cosine pulse shaping.

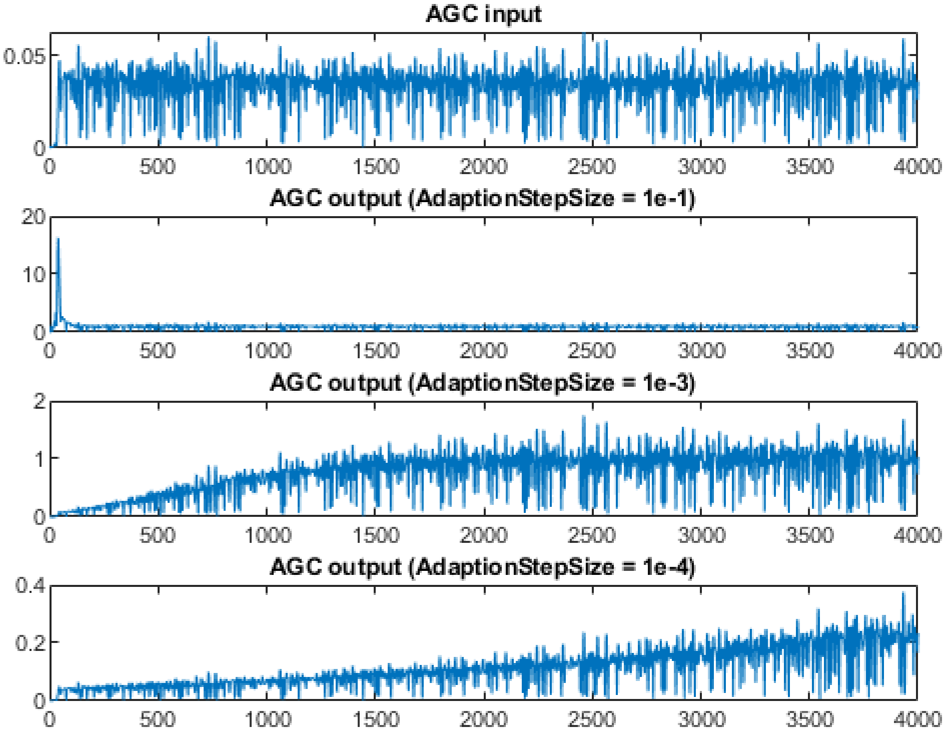
```
d = randi([0 3],500,1);
s = pskmod(d,4,pi/4);
x = 0.1*s;
pulseShaper = comm.RaisedCosineTransmitFilter;
y = pulseShaper(x);
```

Apply AGC.

```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot the signal input to AGC and output after various AGC step sizes.

```
figure
subplot(4,1,1)
plot(abs(y))
title('AGC input')
subplot(4,1,2)
plot(abs(r1))
title('AGC output (AdaptionStepSize = 1e-1)')
subplot(4,1,3)
plot(abs(r2))
title('AGC output (AdaptionStepSize = 1e-3)')
subplot(4,1,4)
plot(abs(r3))
title('AGC output (AdaptionStepSize = 1e-4)')
```



With the step size set to 1e-1, the AGC output overshoot but converges very quickly. With the step size set to 1e-3, the AGC output overshoot disappears and the AGC converges smoothly but more slowly. With the maximum gain set to 1e-4, the AGC takes a very long time to converge.

Adaptively Adjust Received Signal Amplitude Using AGC

Modulate and amplify a QPSK signal. Set the received signal amplitude to approximately 1 volt using an AGC. Then plot the output.

Create a QPSK modulated signal using the QPSK System object.

```
data = randi([0 3],1000,1);
qpsk = comm.QPSKModulator;
modData = qpsk(data);
```

Attenuate the modulated signal.

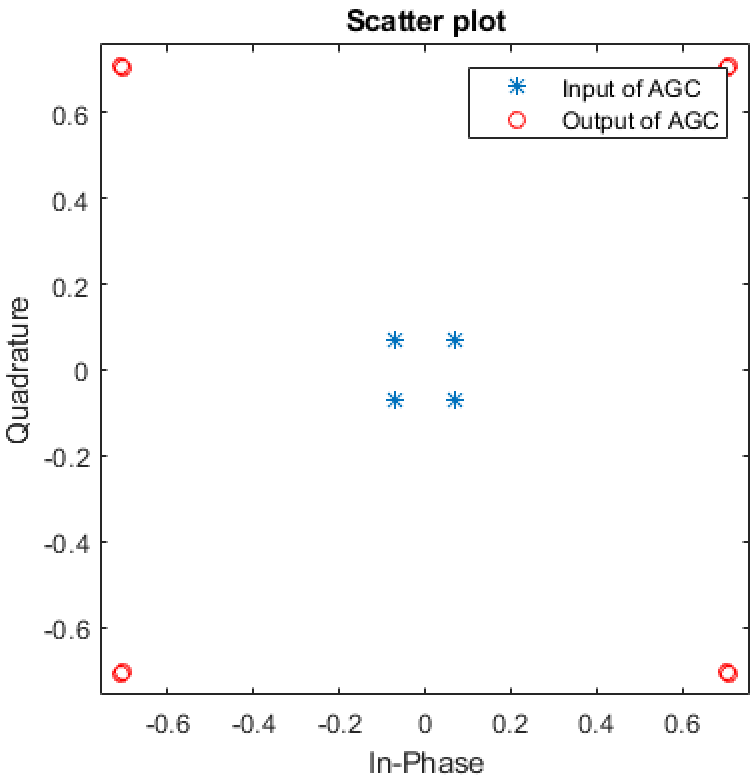
```
txSig = 0.1*modData;
```

Create an AGC System object™ and pass the transmitted signal through it using the `step` function. The AGC adjusts the received signal power to approximately 1 W.

```
agc = comm.AGC;
rxSig = agc(txSig);
```

Plot the signal constellations of the transmit and received signals after the AGC reaches steady-state.

```
h = scatterplot(txSig(200:end),1,0,'*');
hold on
scatterplot(rxSig(200:end),1,0,'or',h);
legend('Input of AGC', 'Output of AGC')
```



Measure and compare the power of the transmitted and received signals after the AGC reaches a steady state. The power of the transmitted signal is 100 times smaller than the power of the received signal.

```
txPower = var(txSig(200:end));
rxPower = var(rxSig(200:end));
[txPower rxPower]
```

```
ans = 1×2

    0.0100    0.9970
```

Plot Effect of Step Size on AGC Performance

Create two AGC System objects™ to adjust the level of the received signal using two different step sizes with identical update periods.

Generate an 8-PSK signal having power equal to 10 W.

```
data = randi([0 7],200,1);
modData = sqrt(10)*pskmod(data,8,pi/8,'gray');
```

Create a pair of raised cosine matched filters with their `Gain` property set so that they have unity output power.

```
txfilter = comm.RaisedCosineTransmitFilter('Gain',sqrt(8));
rxfilter = comm.RaisedCosineReceiveFilter('Gain',sqrt(1/8));
```

Filter the modulated signal through the raised cosine transmit filter object.

```
txSig = txfilter(modData);
```

Create two AGC objects to adjust the received signal level. Select a step size of 0.01 and 0.1, respectively. Set the update period to 10.

```
agc1 = comm.AGC('AdaptationStepSize',0.01);
agc2 = comm.AGC('AdaptationStepSize',0.1);
```

Pass the modulated signal through the two AGC objects.

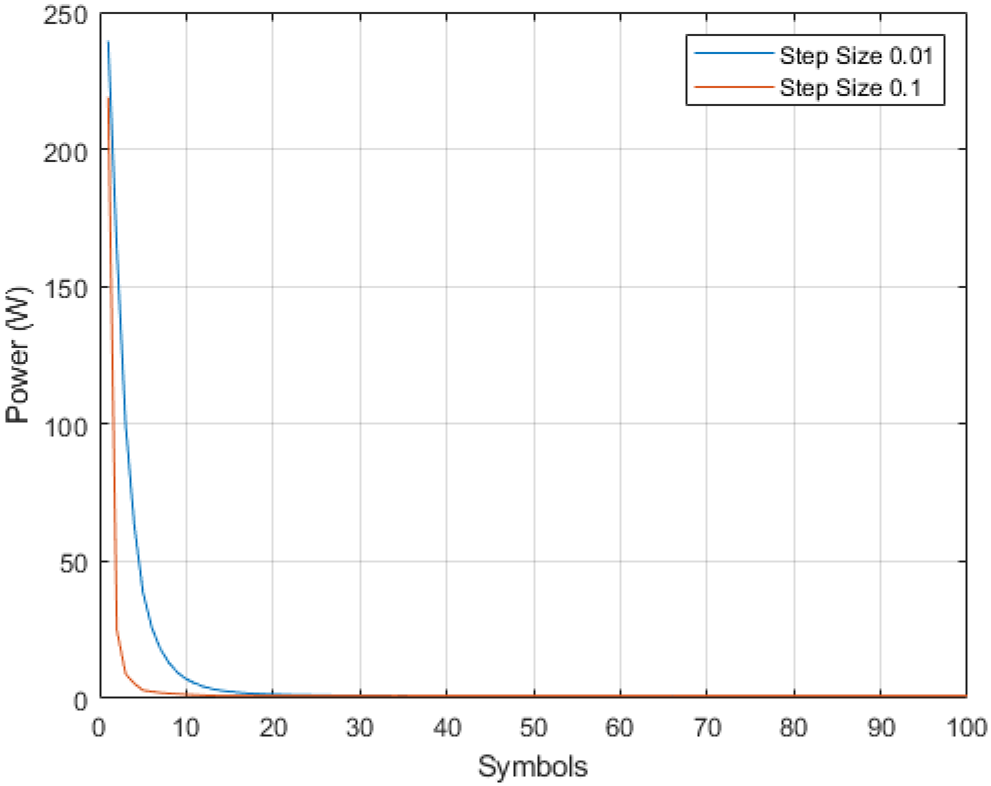
```
agcOut1 = agc1(txSig);
agcOut2 = agc2(txSig);
```

Filter the AGC output signals by using the raised cosine receive filter object.

```
rxSig1 = rxfilter(agcOut1);
rxSig2 = rxfilter(agcOut2);
```

Plot the power of the filtered AGC responses while accounting for the 10 symbol delay through the transmit-receive filter pair.

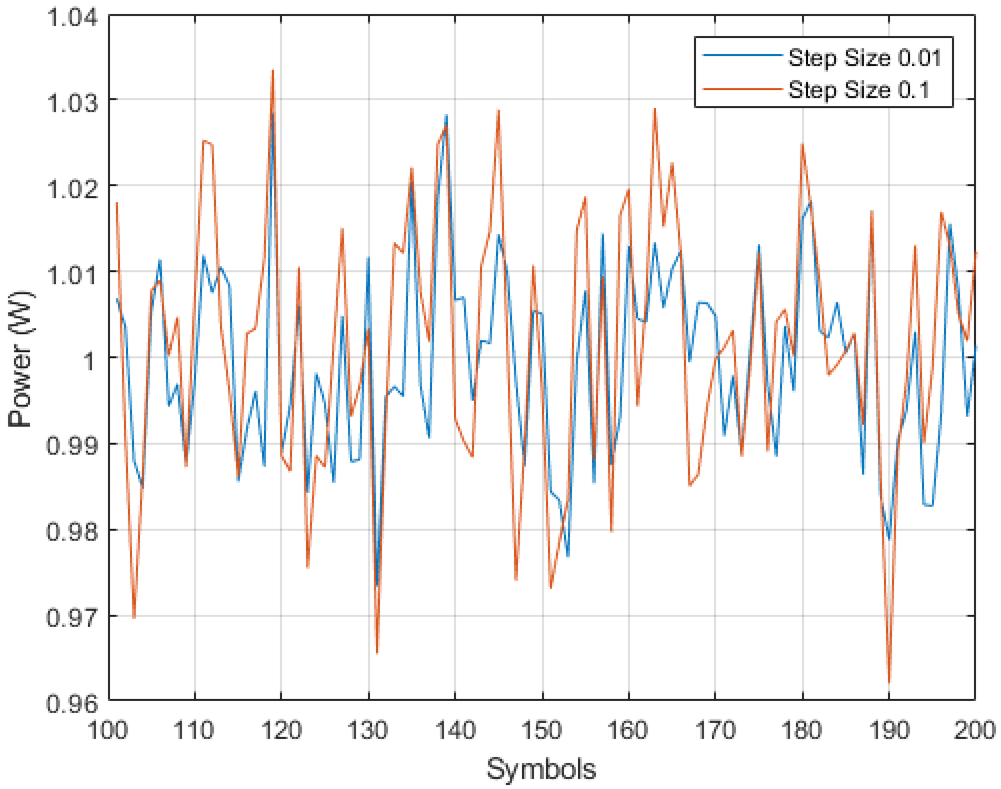
```
plot([abs(rxSig1(11:110)).^2 abs(rxSig2(11:110)).^2])
grid on
xlabel('Symbols')
ylabel('Power (W)')
legend('Step Size 0.01','Step Size 0.1')
```



The signal with the larger step size converges faster to the AGC target power level of 1 W.

Plot the power of the steady-state filtered AGC signals by including only the last 100 symbols.

```
plot((101:200),[abs(rxSig1(101:200)).^2 abs(rxSig2(101:200)).^2])
grid on
xlabel('Symbols')
ylabel('Power (W)')
legend('Step Size 0.01','Step Size 0.1')
```



The larger AGC step size results in less accurate gain correction.

The reduced accuracy suggests a trade off with the `AdaptationStepSize` property. Larger values result in faster convergence at the expense of less accurate gain control.

Demonstrate Effect of Maximum AGC Gain on Packet Data

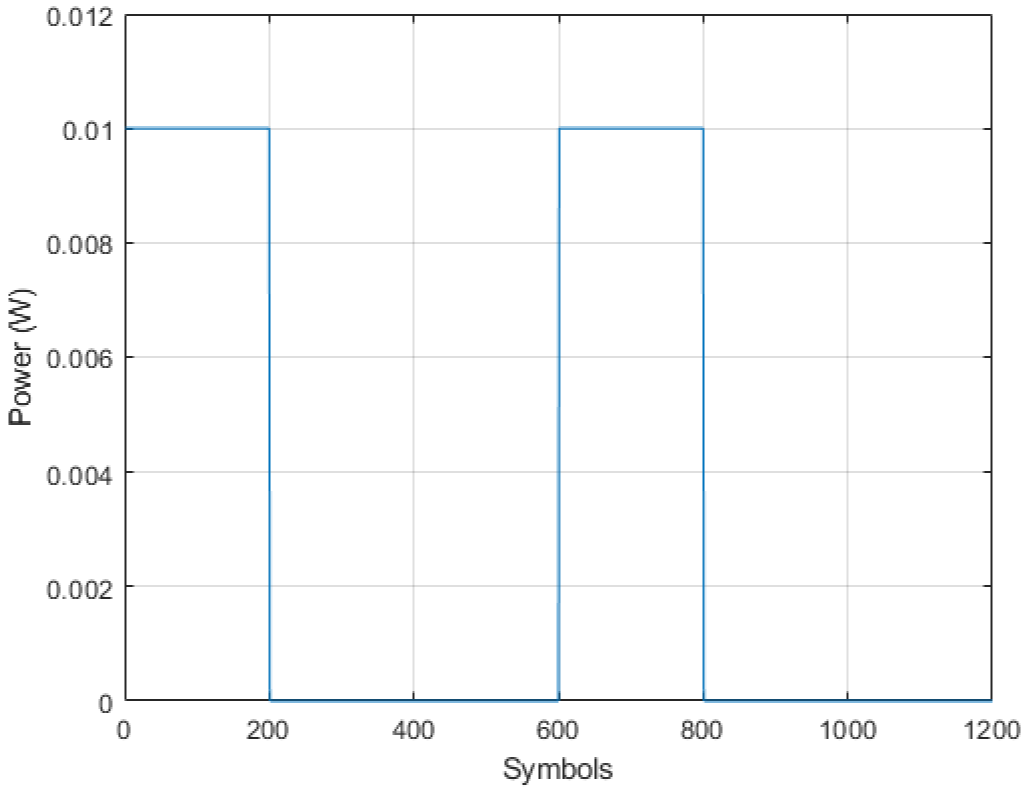
Pass attenuated QPSK packet data to two AGCs having different maximum gains. Then display the results.

Create two, 200-symbol QSPK data packets. Transmit the packets over a 1200-symbol frame.

```
modData1 = pskmod(randi([0 3],200,1),4,pi/4);
modData2 = pskmod(randi([0 3],200,1),4,pi/4);
txSig = [modData1; zeros(400,1); modData2; zeros(400,1)];
```

Attenuate the transmitted burst signal by 20 dB and plot its power.

```
rxSig = 0.1*txSig;
rxSigPwr = abs(rxSig).^2;
plot(rxSigPwr)
grid
xlabel('Symbols')
ylabel('Power (W)')
```



Create two AGCs, where `agc1` has a maximum power gain of 30 dB and `agc2` has a maximum power gain of 24 dB.

```
agc1 = comm.AGC('MaxPowerGain',30, ...
    'AdaptationStepSize',0.02);

agc2 = comm.AGC('MaxPowerGain',24, ...
    'AdaptationStepSize',0.02);
```

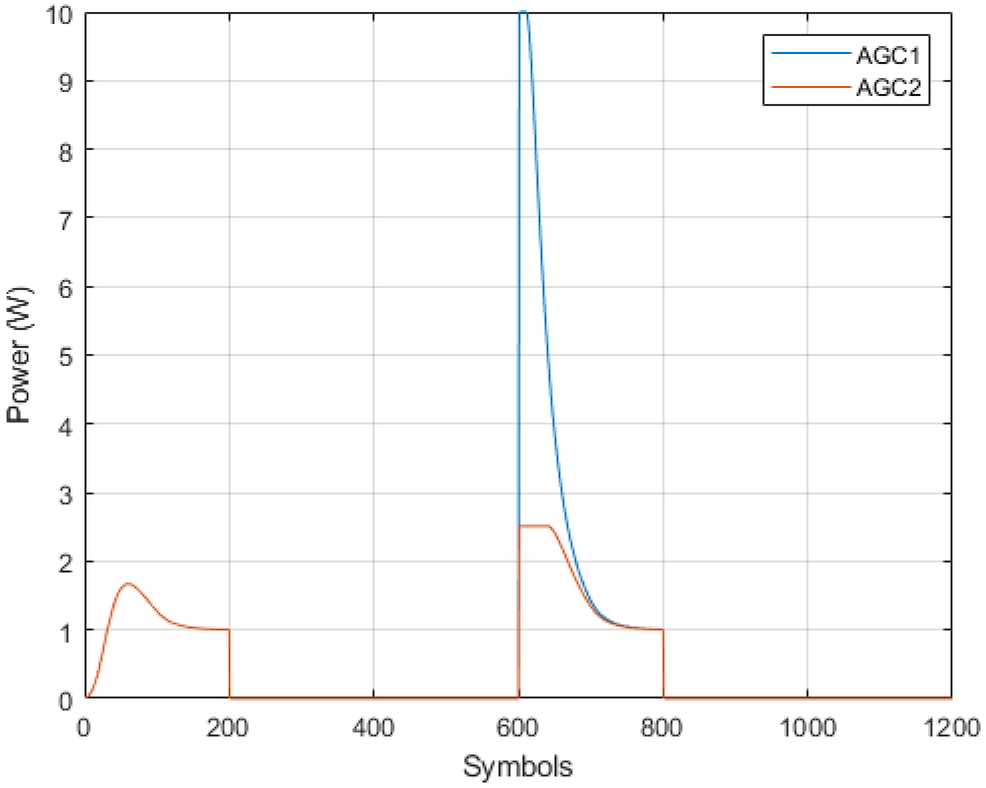
Pass the attenuated signal through the AGCs and calculate the output power.

```
rxAGC1 = agc1(rxSig);
rxAGC2 = agc2(rxSig);
```

```
pwrAGC1 = abs(rxAGC1).^2;
pwrAGC2 = abs(rxAGC2).^2;
```

Plot the output power.

```
plot([pwrAGC1 pwrAGC2])
legend('AGC1','AGC2')
grid
xlabel('Symbols')
ylabel('Power (W)')
```



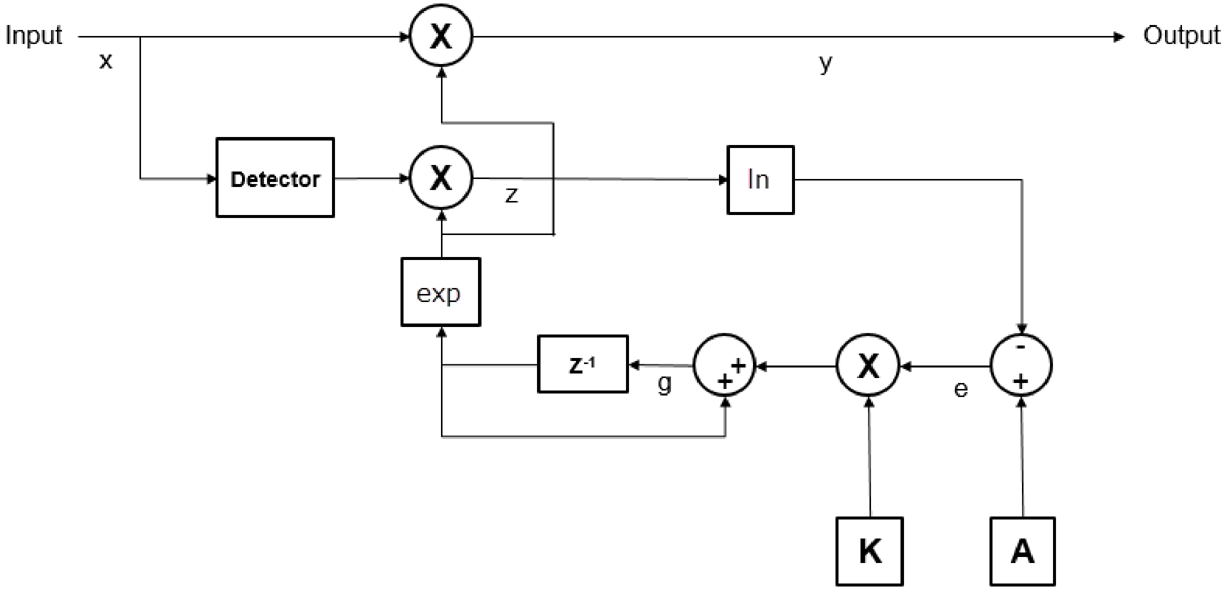
Initially, for the second packet, the `agc1` output signal power is too high because the AGC applied its maximum gain during the period when no data was transmitted. The corresponding `agc2` output signal power does not overshoot the target power level of 1 W by the same amount. It converges to the correct power more quickly due to its smaller maximum gain.

Algorithms

Logarithmic Loop AGC

For the logarithmic loop AGC, the output signal is the product of the input signal and the exponential of the loop gain. The error signal is the difference between the reference level and the product of the logarithm of the detector output and the exponential of the loop gain. After multiplying by the step size, the AGC passes the error signal to an integrator.

The logarithmic loop AGC provides good performance for a variety of signal types, including amplitude modulation. Unlike the previous AGC (R2015a and earlier), the detector is applied to the input signal, which results in faster convergence times and increased signal power variation at the detector input. The larger variation is not a problem for floating point systems. A block diagram of the algorithm is shown.



Mathematically, the algorithm is summarized as

$$
\begin{aligned}
&y(n) = x(n) \cdot \exp(g(n-1)) \\
&z(n) = D(x(n)) \cdot \exp(2g(n-1)) \\
&e(n) = A - \ln(z(n)) \\
&g(n) = g(n-1) + K \cdot e(n),
\end{aligned}
$$

where

- x represents the input signal.
- y represents the output signal.
- g represents the loop gain.
- *D(•)* represents the detector function.
- z represents the detector output.
- A represents the reference value.
- e represents the error signal.
- K represents the step size.

AGC Detector

The AGC uses a square law detector, in which the output of the detector, *z*, is given by

$$z(m) = \frac{1}{N} \sum_{n=mN}^{(m+1)N-1} \left|y(n)\right|^2 ,$$

where N represents the update period.

AGC Performance Criteria

- Attack time — The duration it takes the AGC to respond to an increase in the input amp litude.
- 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

AGC

Introduced in R2013a

reset

System object: comm.AGC
Package: comm

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.

step

System object: comm.AGC
Package: comm

Apply adaptive gain to input signal

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` applies an adaptive gain to the input signal `X`, to achieve a reference signal level at the output, `Y`. You must specify `X` as a double-precision or single-precision column vector. The AGC object uses a square law detector to determine the output signal level.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.AlgebraicDeinterleaver System object

Package: comm

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.

To deinterleave input symbols using an algebraically derived permutation vector:

1. Define and set up your algebraic deinterleaver object. See “Construction” on page 3-37.
2. Call `step` to deinterleave the input symbols according to the properties of `comm.AlgebraicDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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

step	Deinterleave input symbols using algebraically derived permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Algebraic Interleaving and Deinterleaving

Create algebraic interleaver and deinterleaver objects having a length of 16.

```
interleaver = comm.AlgebraicInterleaver('Length',16);
deinterleaver = comm.AlgebraicDeinterleaver('Length',16);
```

Generate 8-ary data. Interleave and deinterleave the data.

```
data = randi([0 7],16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Compare the original, interleaved, and deinterleaved data.

```
[data,intData,deIntData]
```

```
ans = 16×3

     6     3     6
     7     7     7
     1     7     1
     7     7     7
     5     7     5
     0     7     0
     2     1     2
     4     6     4
     7     6     7
     7     7     7
      ⋮
```

Confirm the original and deinterlaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AlgebraicInterleaver` | `comm.BlockInterleaver`

Introduced in R2012a

step

System object: comm.AlgebraicDeinterleaver
Package: comm

Deinterleave input symbols using algebraically derived permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.AlgebraicInterleaver System object

Package: comm

Permute input symbols using algebraically derived permutation vector

Description

The `AlgebraicInterleaver` object rearranges the elements of its input vector using an algebraically derived permutation.

To interleave input symbols using an algebraically derived permutation vector:

1. Define and set up your algebraic interleaver object. See “Construction” on page 3-43.
2. Call `step` to interleave the input symbols according to the properties of `comm.AlgebraicInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

step	Permute input symbols using an algebraically derived permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Algebraic Interleaving and Deinterleaving

Create algebraic interleaver and deinterleaver objects having a length of 16.

```
interleaver = comm.AlgebraicInterleaver('Length',16);
deinterleaver = comm.AlgebraicDeinterleaver('Length',16);
```

Generate 8-ary data. Interleave and deinterleave the data.

```
data = randi([0 7],16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Compare the original, interleaved, and deinterleaved data.

```
[data,intData,deIntData]
```

```
ans = 16×3

     6     3     6
     7     7     7
     1     7     1
     7     7     7
     5     7     5
     0     7     0
     2     1     2
     4     6     4
     7     6     7
     7     7     7
      ⋮
```

Confirm the original and deinterlaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AlgebraicDeinterleaver`

Introduced in R2012a

step

System object: comm.AlgebraicInterleaver
Package: comm

Permute input symbols using an algebraically derived permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.APPDecoder System object

Package: comm

Decode convolutional code using the a posteriori probability method

Description

The `APPDecoder` object performs a posteriori probability (APP) decoding of a convolutional code.

To perform a posteriori probability (APP) decoding of a convolutional code:

1. Define and set up your a posteriori probability decoder object. See “Construction” on page 3-49.
2. Call `step` to perform APP decoding according to the properties of `comm.APPDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 $2^{\text{NumScalingBits}}$ 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

reset	Reset states of APP decoder object
step	Decode convolutional code using the a posteriori probability method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Decode Convolutional Code Using the APP Decoder

This example shows how to use the APP decoder on a convolutionally encoded 8-PSK-modulated bit stream transmitted through an AWGN channel.

Create 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);
```

Create convolutional decoder, PSK demodulator, and error rate System objects.

```
hAPPDec = comm.APPDecoder(...
    'TrellisStructure',poly2trellis(7,[171 133]), ...
    'Algorithm','True APP','CodedBitLLROutputPort',false);
hDemod = comm.PSKDemodulator('BitOutput',true,'PhaseOffset',0, ...
    'DecisionMethod','Approximate log-likelihood ratio', ...
    'Variance',noiseVar);
hError = comm.ErrorRate;
```

Transmit a convolutionally encoded 8-PSK-modulated bit stream through an AWGN channel. Demodulate the received signal using soft-decision. Decode the demodulated signal using the APP decoder.

```
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);
     % The APP decoder assumes a polarization of the soft inputs that is
     % inverse to that of the demodulator soft outputs. Change the sign of
     % demodulated signal.
     receivedSoftBits = step(hAPPDec,zeros(frameLength,1),-demodSignal);
     % Convert from soft-decision to hard-decision.
     receivedBits = double(receivedSoftBits > 0);
     % Count errors
     errorStats = step(hError,data,receivedBits);
end
```

Display the error rate information.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

```
Error rate = 0.000000
Number of errors = 0
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.ViterbiDecoder` | `poly2trellis`

Introduced in R2012a

reset

System object: comm.APPDecoder
Package: comm

Reset states of APP decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `APPDecoder` object, `H`.

step

System object: comm.APPDecoder
Package: comm

Decode convolutional code using the a posteriori probability method

Syntax

`[LUD,LCD] = step(H,LU,LC)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.AWGNChannel System object

Package: comm

Add white Gaussian noise to input signal

Description

comm.AWGNChannel adds white Gaussian noise to the input signal.

When applicable, if inputs to the object have a variable number of channels, the EbNo, EsNo, SNR, BitsPerSymbol, SignalPower, SamplesPerSymbol, and Variance properties must be scalars.

To add white Gaussian noise to an input signal:

1. Create the comm.AWGNChannel object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
awgnchan = comm.AWGNChannel
awgnchan = comm.AWGNChannel(Name,Value)
```

Description

`awgnchan = comm.AWGNChannel` creates an additive white Gaussian noise (AWGN) channel System object, `awgnchan`. This object then adds white Gaussian noise to a real or complex input signal.

`awgnchan = comm.AWGNChannel(Name,Value)` creates a AWGN channel object, `awgnchan`, 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

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`NoiseMethod` — Noise level method

`'Signal to noise ratio (Eb/No)'` (default) | `'Signal to noise ratio (Es/No)'` | `'Signal to noise ratio (SNR)'` | `'Variance'`

Noise level method, specified as `'Signal to noise ratio (Eb/No)'`, `'Signal to noise ratio (Es/No)'`, `'Signal to noise ratio (SNR)'`, or `'Variance'`. For more information, see Specifying the Variance Directly or Indirectly on page 3-78.

Data Types: `char`

EbNo — Ratio of energy per bit to noise power spectral density

`10` (default) | scalar | row vector

Ratio of energy per bit to noise power spectral density (Eb/No) in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`.

Data Types: `double`

EsNo — Ratio of energy per symbol to noise power spectral density

`10` (default) | scalar | row vector

Ratio of energy per symbol to noise power spectral density (Es/No) in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Es/No)'`.

Data Types: `double`

`SNR` — Ratio of signal power to noise power

`10` (default) | scalar | row vector

Ratio of signal power to noise power in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (SNR)'`.

Data Types: `double`

`BitsPerSymbol` — Number of bits per symbol

`1` (default) | positive integer

Number of bits per symbol, specified as a positive integer.

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`.

Data Types: `double`

`SignalPower` — Input signal power

`1` (default) | positive scalar | row vector

Input signal power in watts, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels. The object assumes a nominal impedance of 1 Ω.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`, `'Signal to noise ratio (Es/No)'`, or `'Signal to noise ratio (SNR)'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

`1` (default) | positive integer | row vector

Number of samples per symbol, specified as a positive integer or 1-by-N_C vector. N_C is the number of channels.

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'` or `'Signal to noise ratio (Es/No)'`.

Data Types: `double`

`VarianceSource` — Source of noise variance

`'Property'` (default) | `'Input port'`

Source of noise variance, specified as `'Property'` or `'Input port'`.

- Set `VarianceSource` to `'Property'` to specify the noise variance value using the Variance property.
- Set `VarianceSource` to `'Input port'` to specify the noise variance value using an input to the object, when you call it as a function.

For more information, see Specifying the Variance Directly or Indirectly on page 3-78.

Dependencies

This property applies when NoiseMethod is `'Variance'`.

Data Types: `char`

`Variance` — White Gaussian noise variance

`1` (default) | positive scalar | row vector

White Gaussian noise variance, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Variance'` and VarianceSource is set to `'Property'`.

Data Types: `double`

`RandomStream` — Source of random number stream

`'Global stream'` (default) | `'mt19937ar with seed'`

Source of random number stream, specified as `'Global stream'` or `'mt19937ar with seed'`.

- When you set `RandomStream` to `'Global stream'`, the 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. For example `reset(RandStream.getGlobalStream)`. For more information, see `RandStream`.
- When you set `RandomStream` to `'mt19937ar with seed'`, the object uses the mt19937ar algorithm for normally distributed random number generation. In this scenario, when you call the `reset` function, the object reinitializes the random number stream to the value of the `Seed` property. You can generate reproducible numbers by resetting the object.

For a complex input signal, the object creates the random data as follows:

$\texttt{noise = randn(}N_S\texttt{,}N_C\texttt{)+1i(randn(}N_S\texttt{,}N_C\texttt{))}$

N_S is the number of samples and N_C is the number of channels.

Dependencies

This property applies when NoiseMethod is set to `'Variance'`.

Data Types: `char`

Seed — Initial seed

`67` (default) | nonnegative integer

Initial seed of the mt19937ar random number stream, specified as a nonnegative integer. For each call to the `reset` function, the object reinitializes the mt19937ar random number stream to the `Seed` value.

Dependencies

This property applies when RandomStream is set to `'mt19937ar with seed'`.

Data Types: `double`

Usage

Syntax

```
outsignal = awgnchan(insignal)
outsignal = awgnchan(insignal,var)
```

Description

`outsignal = awgnchan(insignal)` adds white Gaussian noise, as specified by `awgnchan`, to the input signal. The result is returned in `outsignal`.

`outsignal = awgnchan(insignal,var)` specifies the variance of the white Gaussian noise. This syntax applies when you set the NoiseMethod to `'Variance'` and VarianceSource to `'Input port'`.

For example:

```
awgnchan = comm.AWGNChannel('NoiseMethod','Variance','VarianceSource','Input port');
var = 12;
...
outsignal = awgnchan(insignal,var);
```

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`insignal` — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, an N_S-element vector, or an N_S-by-N_C matrix. N_S is the number of samples and N_C is the number of channels.

Data Types: `double`
Complex Number Support: Yes

`var` — Variance of additive white Gaussian noise

positive scalar | row vector

Variance of additive white Gaussian noise, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels, as determined by the number of columns in the input signal matrix.

Output Arguments

`outsignal` — Output signal

matrix

Output signal, returned with the same dimensions as `insignal`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create Default AWGN Channel System Object

Create an AWGN channel System object with the default configuration. Pass signal data through this channel.

Create an AWGN channel object and signal data.

```
awgnchan = comm.AWGNChannel;
insignal = randi([0 1],100,1);
```

Send the input signal through the channel.

```
outsignal = awgnchan(insignal);
```

Add White Gaussian Noise to 8-PSK Signal

Modulate an 8-PSK signal, add white Gaussian noise, and plot the signal to observe the effects of noise.

Create a PSK modulator System object™. The default modulation order for the PSK modulator object is 8.

```
pskModulator = comm.PSKModulator;
```

Modulate the signal.

```
modData = pskModulator(randi([0 7],2000,1));
```

Add white Gaussian noise to the modulated signal by passing the signal through an AWGN channel.

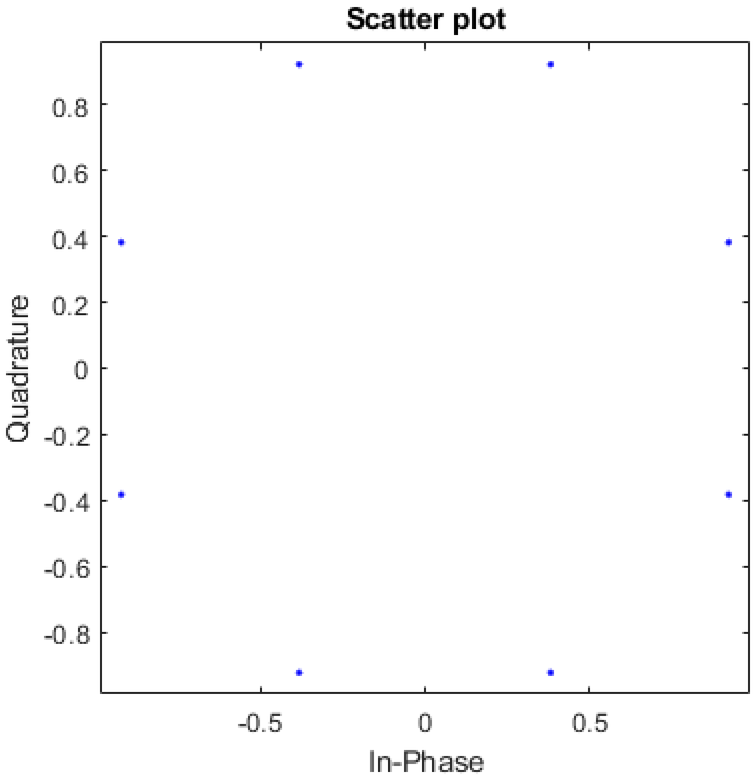
```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',3);
```

Transmit the signal through the AWGN channel.

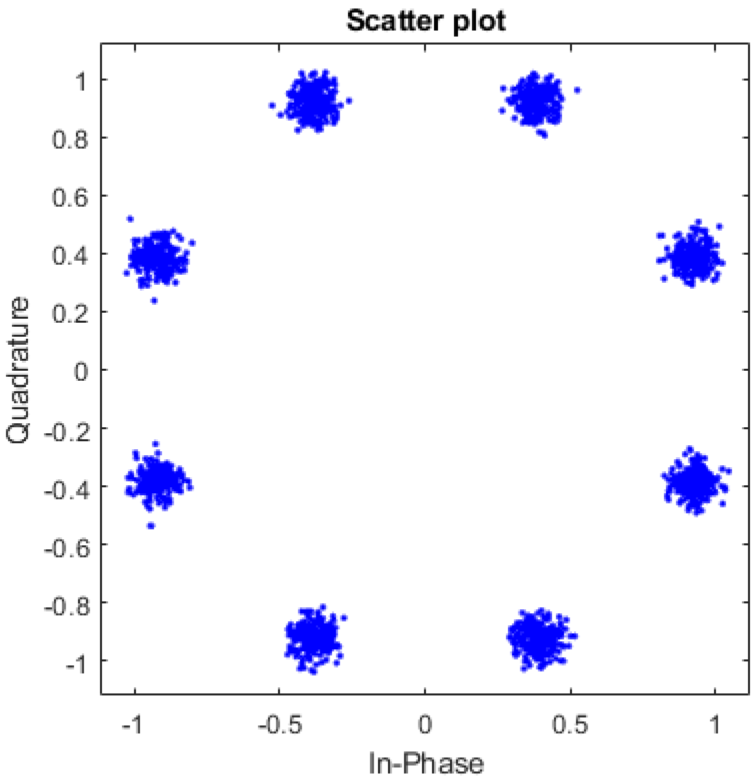
```
channelOutput = channel(modData);
```

Plot the noiseless and noisy data using scatter plots to observe the effects of noise.

```
scatterplot(modData)
```



```
scatterplot(channelOutput)
```



Change the EbNo property to 10 dB to increase the noise.

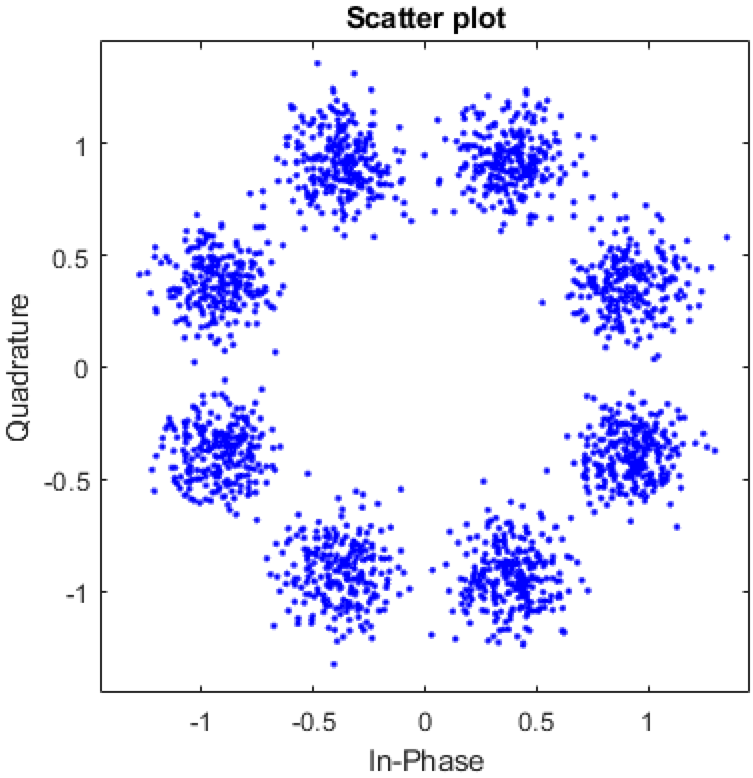
```
channel.EbNo = 10;
```

Pass the modulated data through the AWGN channel.

```
channelOutput = channel(modData);
```

Plot the channel output. You can see the effects of increased noise.

```
scatterplot(channelOutput)
```



Process Signals When Number of Channels Changes

Pass a single-channel and multichannel signal through an AWGN channel System object™.

Create an AWGN channel System object with the Eb/No ratio set for a single channel input. In this case, the `EbNo` property is a scalar.

```
channel = comm.AWGNChannel('EbNo',15);
```

Generate random data and apply QPSK modulation.

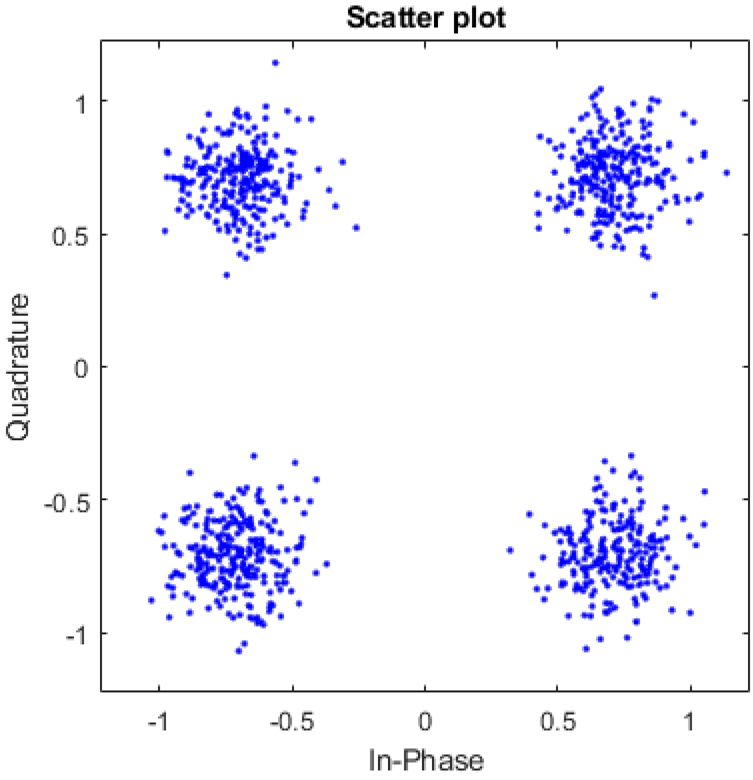
```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Pass the modulated data through the AWGN channel.

```
rxSig = channel(modData);
```

Plot the noisy constellation.

```
scatterplot(rxSig)
```



Generate two-channel input data and apply QPSK modulation.

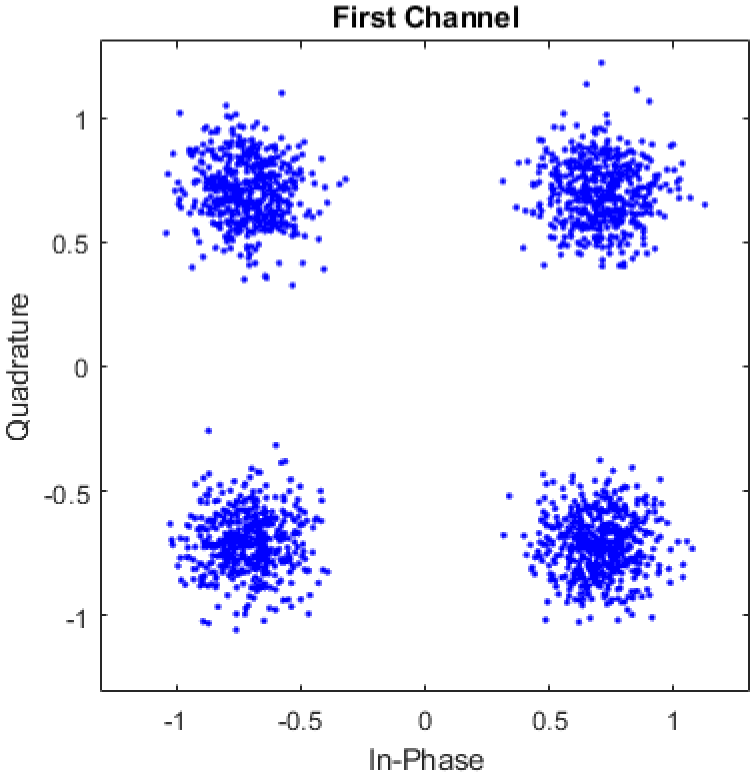
```
data = randi([0 3],2000,2);
modData = pskmod(data,4,pi/4);
```

Pass the modulated data through the AWGN channel.

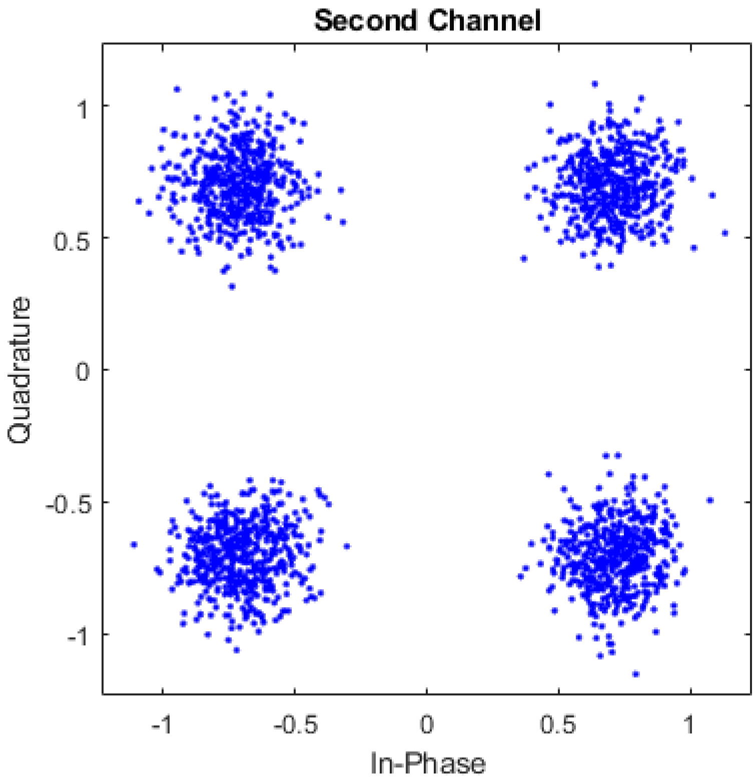
```
rxSig = channel(modData);
```

Plot the noisy constellations. Each channel is represented as a single column in `rxSig`. The plots are nearly identical, becasue the same Eb/No value is applied to both channels.

```
scatterplot(rxSig(:,1))
title('First Channel')
```



```
scatterplot(rxSig(:,2))
title('Second Channel')
```



Modify the AWGN channel object to apply a different Eb/No value to each channel. To apply different values, set the `EbNo` property to a 1-by-2 vector. When changing the dimension of the `EbNo` property, you must release the AWGN channel object.

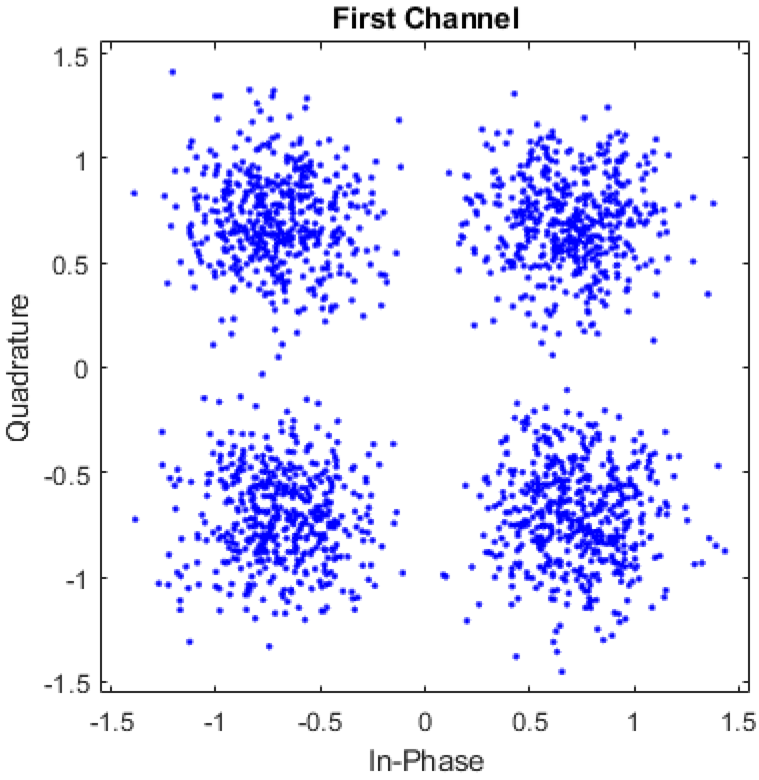
```
release(channel)
channel.EbNo = [10 20];
```

Pass the data through the AWGN channel.

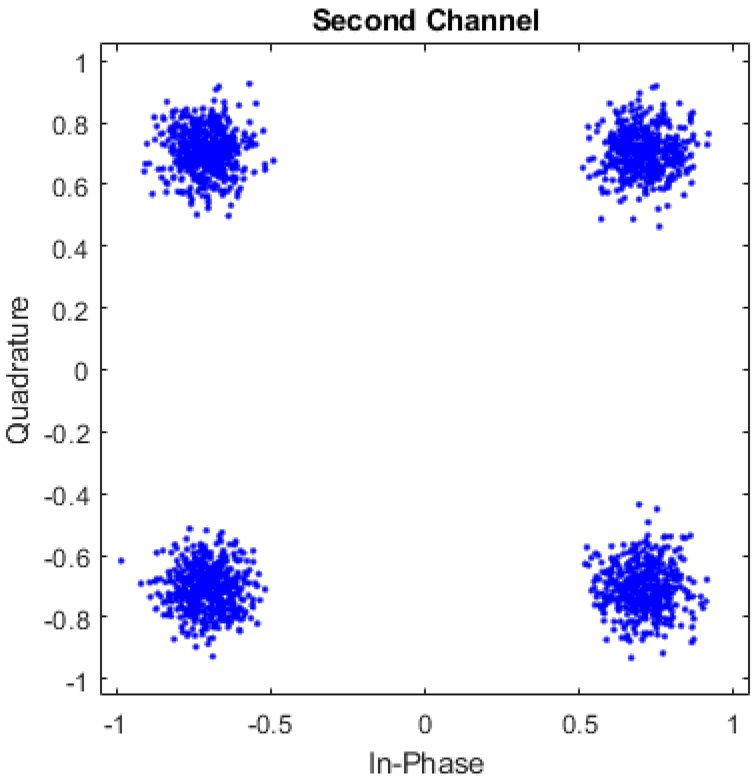
```
rxSig = channel(modData);
```

Plot the noisy constellations. The first channel has significantly more noise due to its lower Eb/No value.

```
scatterplot(rxSig(:,1))
title('First Channel')
```



```
scatterplot(rxSig(:,2))
title('Second Channel')
```



Add AWGN Using Noise Variance Input Port

Apply 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'`.

```
channel = comm.AWGNChannel('NoiseMethod','Variance', ...
'VarianceSource','Input port');
```

Generate random data for two channels and apply 16-QAM modulation.

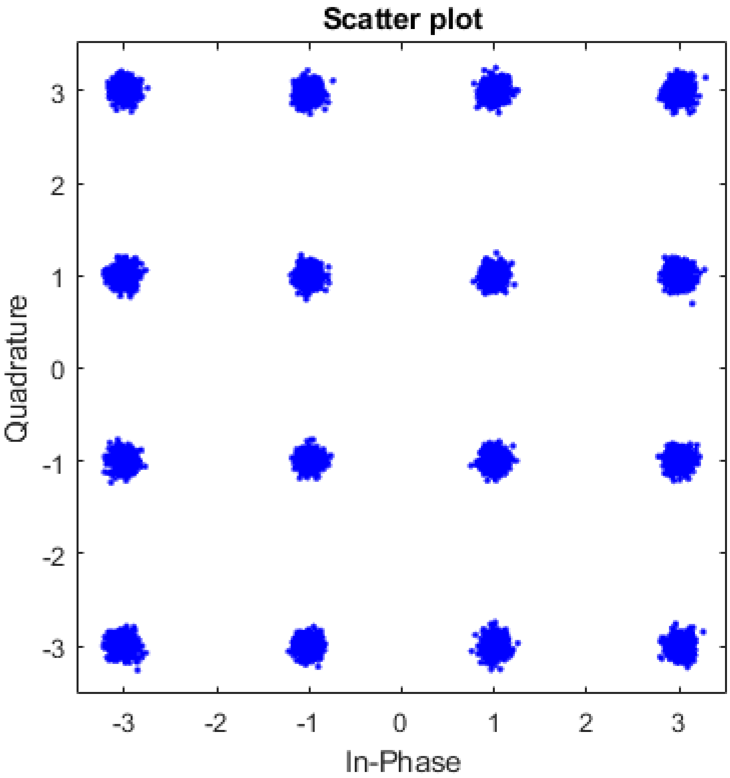
```
data = randi([0 15],10000,2);
txSig = qammod(data,16);
```

Pass the modulated data through the AWGN channel. The AWGN channel object processes data from two channels. The variance input is a 1-by-2 vector.

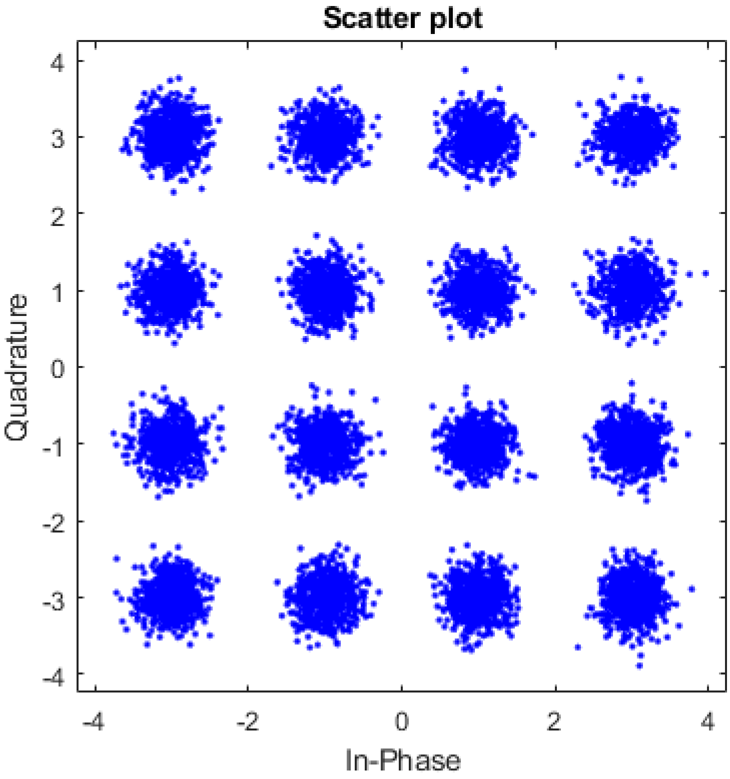
```
rxSig = channel(txSig,[0.01 0.1]);
```

Plot the constellation diagrams for the two channels. The second signal is noisier because its variance is ten times larger.

```
scatterplot(rxSig(:,1))
```

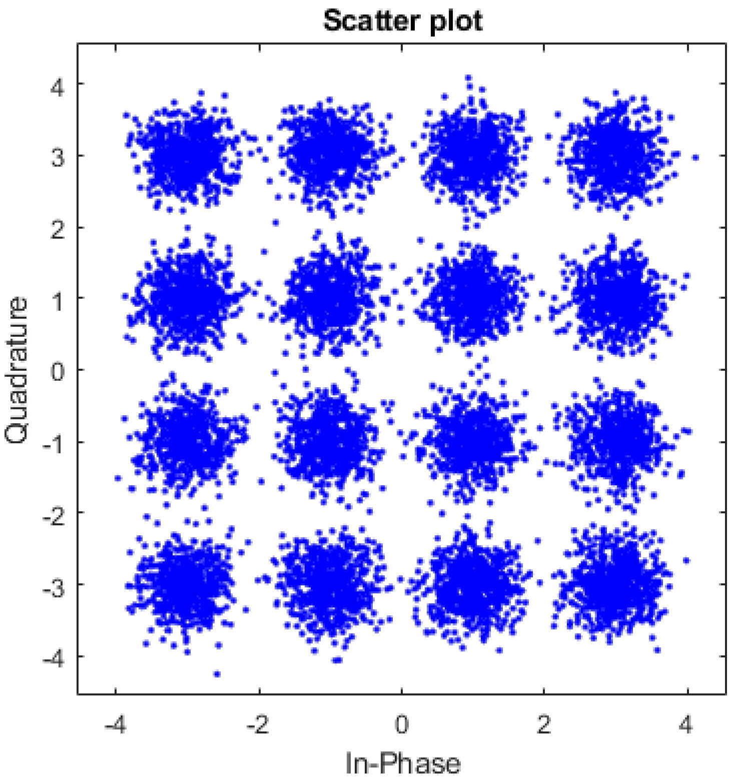


```
scatterplot(rxSig(:,2))
```

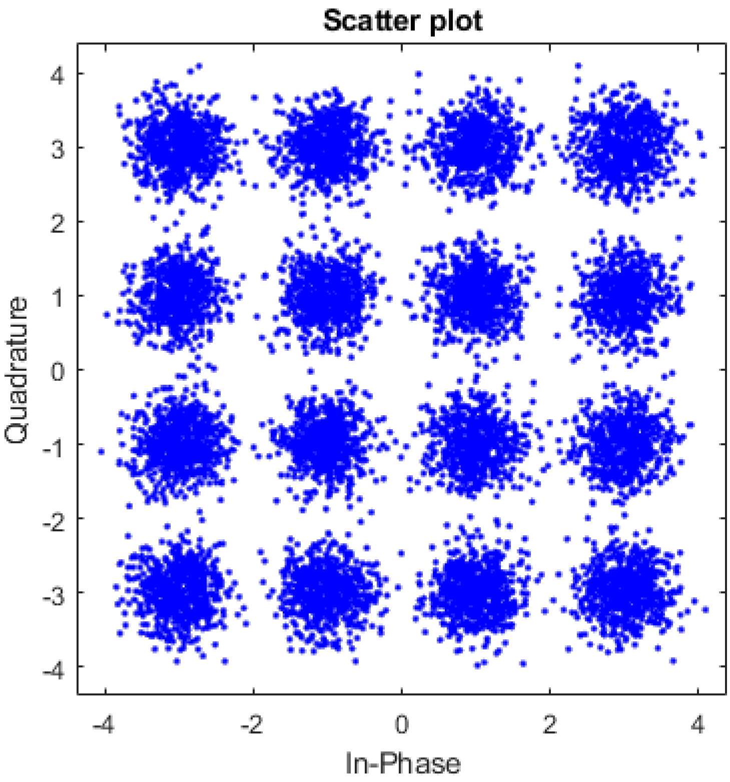


Repeat the process where the noise variance input is a scalar. The same variance is applied to both channels. The constellation diagrams are nearly identical.

```
rxSig = channel(txSig,0.2);
scatterplot(rxSig(:,1))
```



```
scatterplot(rxSig(:,2))
```



Set Random Number Seed for Repeatability

Specify a seed to produce the same outputs when using a random stream in which you specify the seed.

Create an AWGN channel System object™. Set the `NoiseMethod` property to `'Variance'`, the `RandomStream` property to `'mt19937ar with seed'`, and the `Seed` property to `99`.

```
channel = comm.AWGNChannel( ...
    'NoiseMethod','Variance', ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',99);
```

Pass data through the AWGN channel.

```
y1 = channel(zeros(8,1));
```

Pass another all-zeros vector through the channel.

```
y2 = channel(zeros(8,1));
```

Because the seed changes between function calls, the output is different.

```
isequal(y1,y2)
```

```
ans = logical
   0
```

Reset the AWGN channel object by calling the `reset` function. The random data stream is reset to the initial seed of `99`.

```
reset(channel);
```

Pass the all-zeros vector through the AWGN channel.

```
y3 = channel(zeros(8,1));
```

Confirm that the two signals are identical.

```
isequal(y1,y3)
```

```
ans = logical
   1
```

Algorithms

Relationship Among Eb/No, Es/No, and SNR Modes

For uncoded complex input signals, comm.AWGNChannel relates $E_b/N_{0,}$ E_s/N_0, and SNR according to these equations:

$$E_s/N_0 = N_{sps} \times SNR$$
$$E_s/N_0 = E_b/N_0 + 10\log_{10}(k) \text{ in dB}$$

where

- E_s represents the signal energy in joules.
- E_b represents the bit energy in joules.
- N_0 represents the noise power spectral density in watts/Hz.
- N_{sps} represents the number of samples per symbol, SamplesPerSymbol.
- k represents the number of information bits per input symbol, BitsPerSymbol.

For real signal inputs, the comm.AWGNChannel relates E_s/N_0 and SNR according to this equation:

$$E_s/N_0 = 0.5\ (N_{sps}) \times SNR$$

Note

- All values of power assume a nominal impedance of 1 ohm.
- The equation for the real case differs from the corresponding equation for the complex case by a factor of 2. Specifically, the object 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, see AWGN Channel Noise Level.

Specifying the Variance Directly or Indirectly

To directly specify the variance of the noise generated by comm.AWGNChannel, specify VarianceSource as:

- `'Property'`, then set NoiseMethod to `'Variance'` and specify the variance with the Variance property.
- `'Input port'` then specify the variance level for the object as an input with an input argument, var.

To specify variance indirectly, that is, to have it calculated by comm.AWGNChannel, specify VarianceSource as `'Property'` and the NoiseMethod as:

- `'Signal to noise ratio (Eb/No)'`, where the object uses these properties to calculate the variance:
  - EbNo, the ratio of bit energy to noise power spectral density
  - BitsPerSymbol
  - SignalPower, the actual power of the input signal samples
  - SamplesPerSymbol
- `'Signal to noise ratio (Es/No)'`, where the object uses these properties to calculate the variance:
  - EsNo, the ratio of signal energy to noise power spectral density
  - SignalPower, the actual power of the input signal samples
  - SamplesPerSymbol
- `'Signal to noise ratio (SNR)'`, where the object uses these properties to calculate the variance:
  - SNR, the ratio of signal power to noise power
  - SignalPower, the actual power of the input signal samples

Changing the number of samples per symbol (SamplesPerSymbol) affects the variance of the noise added per sample, which also causes a change in the final error rate.

$$\mathit{NoiseVariance} = \text{SignalPower} \times \text{SamplesPerSymbol} \times 10^{(\text{EsNo})/10}$$

Tip Select the number of samples per symbol based on what constitutes a symbol and the oversampling applied to it. For example, a symbol could have 3 bits and be oversampled by 4. For more information, see AWGN Channel Noise Level.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Blocks

AWGN Channel | MIMO Fading Channel

System Objects

`comm.BinarySymmetricChannel` | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Topics

AWGN Channel

Introduced in R2012a

comm.BarkerCode System object

Package: comm

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.

To synchronize using a Barker code:

1. Define and set up your Barker code object. See “Construction” on page 3-81.
2. Call `step` to synchronize according to the properties of `comm.BarkerCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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` on page 3-0 property.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `int8`. The default is `double`.

Methods

reset	Reset states of Barker code generator object
step	Generate Barker code

Common to All System Objects	
release	Allow System object property value changes

Examples

Generate Barker Code Sequence

Create a Barker code object having 10 samples per frame.

```
barker = comm.BarkerCode('SamplesPerFrame',10)
```

```
barker = 
  comm.BarkerCode with properties:

             Length: 7
    SamplesPerFrame: 10
     OutputDataType: 'double'
```

Generate the Barker code sequence.

```
seq = barker()
```

```
seq = 10×1

    -1
    -1
    -1
     1
     1
    -1
     1
    -1
    -1
    -1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HadamardCode` | `comm.OVSFCode`

Introduced in R2012a

reset

System object: comm.BarkerCode
Package: comm

Reset states of Barker code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `BarkerCode` object, `H`.

step

System object: comm.BarkerCode
Package: comm

Generate Barker code

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BasebandFileReader System object

Package: comm

Read baseband signals from file

Description

The comm.BasebandFileReader object reads a baseband signal from a specific type of binary file written by comm.BasebandFileWriter. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. The SampleRate and CenterFrequency properties are saved when the file is created. The comm.BasebandFileReader object automatically reads the sample rate, center frequency, number of channels, and any descriptive data and saves them to its read-only properties.

To create an input signal from a saved baseband file:

1. Create a comm.BasebandFileReader object and set the properties of the object.
2. Call step to generate a baseband signal from saved data.
3. Call release to close the file.

Note Alternatively, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj) and y = obj() perform equivalent operations.

Construction

bbr = comm.BasebandFileReader returns a baseband reader object, bbr, using the default properties.

bbr = comm.BasebandFileReader(fname) returns a baseband reader object and sets fname as the Filename property.

bbr = comm.BasebandFileReader(fname,spf) also sets spf as the SamplesPerFrame property.

bbr = comm.BasebandFileReader(___ ,Name,Value) specifies additional properties using Name,Value pairs. Unspecified properties have default values.

Example:

```
bbr = comm.BasebandFileReader('recorded_data',100);
```

Properties

Filename — Name of the baseband file to read

'example.bb' (default) | character vector

Name of the baseband file to read, specified as a character vector. Specify the absolute path only if the file is not on the MATLAB path. Only the absolute path is saved and displayed.

SampleRate — Sample rate of the saved baseband signal

1 (default) | positive scalar

This property is read-only.

Sample rate of the saved baseband signal in Hz.

CenterFrequency — Center frequency of the saved baseband signal

100000000 (default) | positive scalar | row vector

This property is read-only.

Center frequency of the saved baseband signal in Hz. When this property is a row vector, each element represents the center frequency of a channel in a multichannel signal.

NumChannels — Number of channels of the saved baseband signal

1 (default) | positive integer

This property is read-only.

Number of channels of the saved baseband signal.

Metadata — Data describing the baseband signal

struct() (default) | structure

This property is read-only.

Data describing the baseband signal. If the file has no descriptive data, this property is an empty structure.

SamplesPerFrame — Number of samples per output frame

100 (default) | positive integer

Number of samples per output frame, specified as a positive integer.

Data Types: double

CyclicRepetition — Flag to repeatedly read baseband file

false (default) | true

Flag to repeatedly read baseband file, specified as a logical scalar. To repeatedly read the baseband file specified by Filename, set this property to true.

Methods

info	Characteristic information about baseband file reader
isDone	Read status of baseband file samples
reset	Reset states of baseband file reader object
step	Generate baseband signal from file

Common to All System Objects	
release	Allow System object property value changes

Examples

Read Baseband Data from File

Create a baseband file reader object.

```
bbr = comm.BasebandFileReader
```

```
bbr = 
  comm.BasebandFileReader with properties:
```

```
            Filename: 'B:\matlab\toolbox\comm\comm\example.bb'
          SampleRate: 1
     CenterFrequency: 100000000
         NumChannels: 1
            Metadata: [1x1 struct]
     SamplesPerFrame: 100
    CyclicRepetition: false
```

Use the `info` method to gain additional information about `bbr`. The file contains 10000 samples of type `'double'`. No samples have been read.

```
info(bbr)
```

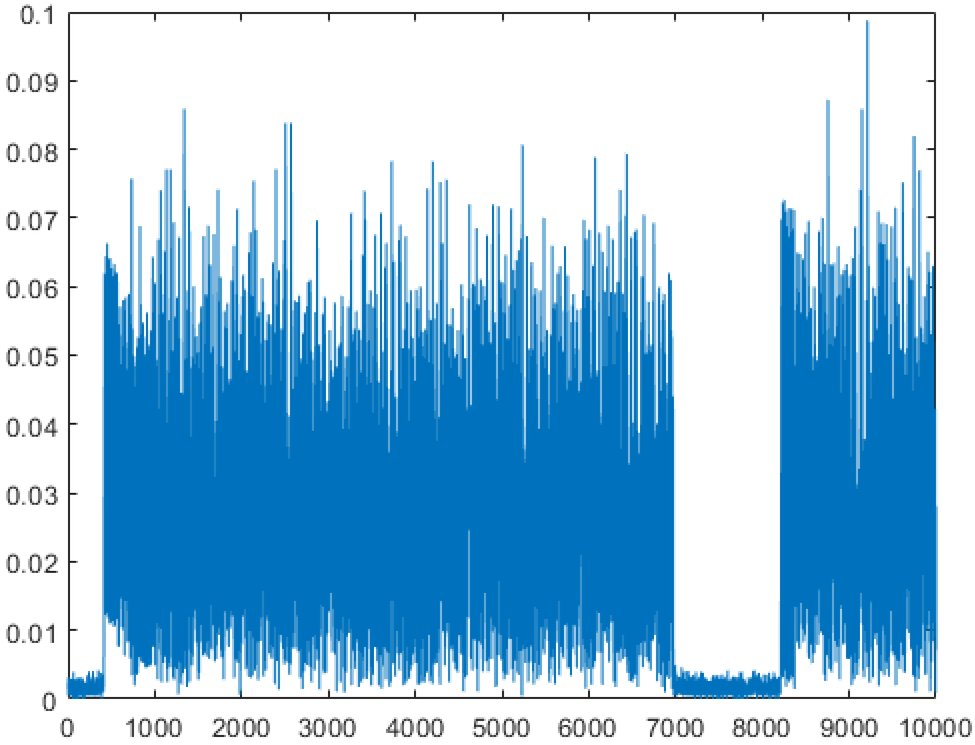
```
ans = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 0
```

Read the entire contents of the `example.bb` file by using the `isDone` method to terminate the loop.

```
y = [];

while ~isDone(bbr)
    x = bbr();
    y = cat(1,y,x);
end
```

Plot the absolute magnitude of the baseband data.

```
plot(abs(y))
```



Confirm that all the samples have been read.

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 10000
```

The total number of samples and the number of samples read are the same.

Release the baseband file reader resources.

```
release(bbr)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BasebandFileWriter`

Introduced in R2016b

info

System object: comm.BasebandFileReader
Package: comm

Characteristic information about baseband file reader

Syntax

```
s = info(bbr)
```

Description

`s = info(bbr)` returns a structure, `s`, containing characteristic information for the `BasebandFileReader` System object, `bbr`. `s` has these fields:

- `NumSamplesInData` is the total number of baseband data samples in the file, returned as a positive integer.
- `DataType` is the data type of the baseband signal in the file.
- `NumSamplesRead` is the number of samples that have been read from the file, returned as a positive integer. It cannot exceed the `NumSamplesInData` property when `CyclicRepetition` is `false`.

Introduced in R2016b

isDone

System object: comm.BasebandFileReader
Package: comm

Read status of baseband file samples

Syntax

```
rs = isDone(bbr)
```

Description

`rs = isDone(bbr)` returns the read status, `rs`, of baseband file reader `bbr`. This status is `true` when `CyclicRepetition` is `false` and the last sample of the file specified by `Filename` has been read.

Introduced in R2016b

reset

System object: comm.BasebandFileReader
Package: comm

Reset states of baseband file reader object

Syntax

```
reset(bbr)
```

Description

`reset(bbr)` resets the states of the `BasebandFileReader` object, `bbr`.

Introduced in R2016b

step

System object: comm.BasebandFileReader
Package: comm

Generate baseband signal from file

Syntax

```
y = step(bbr)
y = bbr()
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`y = step(bbr)` generates a baseband signal, `y`, from the data in the file specified by the `Filename` property of the `BasebandFileReader` object, `bbr`. `y` can be a complex column vector or matrix. If `y` is a matrix, each column represents a separate channel.

`y = bbr()` is equivalent to the first syntax.

Note `bbr` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2016b

comm.BasebandFileWriter System object

Package: comm

Write baseband signals to file

Description

A baseband file is a specific type of binary file written by `comm.BasebandFileWriter`. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. The `SampleRate` and `CenterFrequency` properties are saved when the file is created.

To save a baseband signal to a file:

1. Create a `comm.BasebandFileWriter` object and set the properties of the object.
2. Call `step` to save a baseband signal to a file.
3. Call `release` to save the baseband signal to a file and to close the file.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`bbw = comm.BasebandFileWriter` returns a baseband writer object, `bbw`, using the default properties.

`bbw = comm.BasebandFileWriter(fname)` returns `bbw` and sets `fname` as the `Filename` property.

`bbw = comm.BasebandFileWriter(fname,fs)` also sets `fs` as the `SampleRate` property.

`bbw = comm.BasebandFileWriter(fname,fs,fc)` also sets `fc` as the `CenterFrequency` property.

bbw = comm.BasebandFileWriter(fname,fs,fc,md) also sets structure md as the MetaData property.

bbw = comm.BasebandFileWriter(___ ,Name,Value) specifies additional properties using Name,Value pairs. Unspecified properties have default values.

Example:

```
bbw = comm.BasebandFileWriter('qpsk_data.bb',10e6,2e9);
```

Properties

Filename — Name of saved file

'untitled.bb' (default) | character vector

Name of saved file, specified as a character vector. The filename can include a relative or an absolute path.

SampleRate — Sample rate of output signal

1 (default) | positive scalar

Sample rate of the output signal, specified in Hz as a positive scalar.

CenterFrequency — Center frequency of the baseband signal

100000000 (default) | positive integer scalar | row vector

Center frequency of the baseband signal, specified in Hz as a positive integer scalar or row vector. If CenterFrequency is a row vector, each element corresponds to a channel.

Metadata — Data describing the baseband signal

empty structure (default) | structure

Data describing the baseband signal, specified as a structure. The structure can have any number of fields and any field name. The field values can be of any numeric, logical, or character data type and have any number of dimensions.

NumSamplesToWrite — Number of samples to save

Inf (default) | positive integer

Number of samples to save, specified as a positive integer.

- To write all the baseband signal samples to a file, set NumSamplesToWrite to Inf.
- To write only the last NumSamplesToWrite samples to a file, set NumSamplesToWrite to a finite number.

Data Types: double

Methods

info	Characteristic information about baseband file writer
reset	Reset states of baseband file writer object
step	Write baseband signal to file

Common to All System Objects	
release	Allow System object property value changes

Examples

Write Baseband Signal to File

Create a baseband file writer object having a sample rate of 1 kHz and a 0 Hz center frequency.

```
bbw = comm.BasebandFileWriter('baseband_data.bb',1000,0);
```

Save today's date in the Metadata structure.

```
bbw.Metadata = struct('Date',date);
```

Generate two channels of QPSK-modulated data.

```
d = randi([0 3],1000,2);
x = pskmod(d,4,pi/4,'gray');
```

Write the baseband data to file 'baseband_data.bb'.

```
bbw(x)
```

Display information about bbw. Release the object.

```
info(bbw)
```

```
ans = struct with fields:
             Filename: 'C:\TEMP\Bdoc18a_815039_17936\ib0BF173\26\tpabc23b83\comm-ex6649
      SamplesPerFrame: 1000
          NumChannels: 2
             DataType: 'double'
    NumSamplesWritten: 1000
```

```
release(bbw)
```

Create a baseband file reader object to read the saved data. Read the metadata from the file.

```
bbr = comm.BasebandFileReader('baseband_data.bb','SamplesPerFrame',100);
bbr.Metadata
```

```
ans = struct with fields:
    Date: '26-Feb-2018'
```

Read the data from the file.

```
z = [];

while ~isDone(bbr)
    y = bbr();
    z = cat(1,z,y);
end
```

Display information about `bbr`. Release `bbr`.

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 1000
            DataType: 'double'
      NumSamplesRead: 1000
```

```
release(bbr)
```

Confirm the original modulated data, `x`, matches the data read from file `'baseband_data.bb'`, `z`.

```
isequal(x,z)

ans = logical
   1
```

Tips

- `comm.BasebandFileWriter` writes baseband signals to uncompressed binary files. To share these files, you can compress them to a zip file using the `zip` function. For more information, see “Create and Extract from Zip Archives” (MATLAB).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BasebandFileReader`

Introduced in R2016b

info

Characteristic information about baseband file writer

Syntax

```
s = info(bbw)
```

Description

`s = info(bbw)` returns a structure, `s`, containing characteristic information for the `BasebandFileWriter` System object, `bbw`. `s` has these fields:

- `Filename` is the name of the baseband data file, returned as a character vector. The filename shows the absolute path.
- `SamplesPerFrame` is the number of samples in each frame, returned as a positive integer.
- `NumChannels` is the number of channels, returned as a positive integer greater than or equal to 1.
- `DataType` is the input data type.
- `NumSamplesWritten` is the number of samples written to the file, returned as a positive integer. This field returns the smaller of the total number of samples processed by the object and the `NumSamplesWritten` property.

Note All fields are available when the object is locked. When the object is unlocked, only the `Filename` and `NumSamplesWritten` fields are available.

reset

System object: comm.BasebandFileWriter
Package: comm

Reset states of baseband file writer object

Syntax

`reset(bbw)`

Description

`reset(bbw)` resets the states of the `BasebandFileWriter` object, `bbw`.

Introduced in R2016b

step

System object: comm.BasebandFileWriter
Package: comm

Write baseband signal to file

Syntax

```
step(bbw,x)
bbw(x)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(bbw,x)` writes a baseband signal, `x`, to the file specified by the `Filename` property of the `BasebandFileWriter` object, `bbw`. The number of samples written to the file is determined by the `NumSamplesToWrite` property of `bbw`.

`bbw(x)` is equivalent to the first syntax.

Note `bbw` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2016b

comm.BCHDecoder System object

Package: comm

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 `comm.BCHEncoder` System object.

To decode a binary message from a BCH codeword:

1. Define and set up your BCH decoder object. See “Construction” on page 3-107.
2. Call `step` to recover a binary message vector from a binary BCH codeword vector according to the properties of `comm.BCHDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dec = comm.BCHDecoder` creates a BCH decoder System object, `dec`, that performs BCH decoding.

`dec = comm.BCHDecoder(N,K)` creates a BCH decoder object, `dec`, with the `CodewordLength` property set to `N` and the `MessageLength` property set to `K`.

`dec = comm.BCHDecoder(N,K,GP)` creates a BCH decoder object, `dec`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, and the `GeneratorPolynomial` property set to `GP`.

dec = comm.BCHDecoder(N,K,GP,S) creates a BCH decoder object, dec, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, and the ShortMessageLength property set to S.

dec = comm.BCHDecoder(N,K,GP,S,Name,Value) creates a BCH decoder object, dec, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, the ShortMessageLength property set to S, and each specified property Name set to the specified Value.

dec = comm.BCHDecoder(Name,Value) creates a BCH decoder object, dec, 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 on page 3-0 properties must produce a valid narrow-sense BCH code. For a full-length BCH code, the value of this property must take the form $2^M - 1$, where M is an integer such that $3 \le M \le 16$. The default is 15.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar. The values of the CodewordLength on page 3-0 and MessageLength properties must produce a valid narrow-sense BCH code. The default is 5.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as either Auto or Property. When this property is set to Auto, the BCH code is defined by the CodewordLength on page 3-0 , MessageLength on page 3-0 , GeneratorPolynomial on page 3-0 , and PrimitivePolynomial on page 3-0 properties. When ShortMessageLengthSource is set to Property, you must specify the ShortMessageLength on page 3-0 property, which is used with the other properties to define the BCH code. The default is Auto.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to MessageLength on page 3-0 . When ShortMessageLength < MessageLength, the BCH code is shortened. The default is 5.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as either Auto or Property. Set this property to Auto to create the generator polynomial automatically. Set GeneratorPolynomialSource to Property to specify a generator polynomial using the GeneratorPolynomial on page 3-0 property. The default is Auto.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial as a binary double-precision row vector, a binary Galois field row vector that represents the coefficients of the generator polynomial in order of descending powers, or as a polynomial character vector. The length of the generator polynomial requires a value of CodewordLength on page 3-0 –MessageLength on page 3-0 +1. This property applies when you set GeneratorPolynomialSource on page 3-0 to Property. The default is 'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1', which is the result of bchgenpoly(15,5,[],'double') and 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 the specified generator polynomial is valid. 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 GeneratorPolynomialSource on page 3-0 to Property. The default is true.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as Auto or Property. Set this property to Auto to create a primitive polynomial of degree *M*=ceil(log2(CodewordLength on page 3-0 +1)). Set PrimitivePolynomialSource to Property to specify a polynomial using the PrimitivePolynomial on page 3-0 property. The default is Auto.

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 or a polynomial character vector. This property applies when you set the PrimitivePolynomialSource on page 3-0 property to Property. The default is 'X^4 + X + 1', which is the result of fliplr(de2bi(primpoly(4))).

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as None or Property. Set this property to None to disable puncturing. Set it to Property to decode punctured codewords. This decoding is based on a puncture pattern vector you specify in the PuncturePattern on page 3-0 property. The default is None.

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 on page 3-0 – MessageLength on page 3-0 . 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 PuncturePatternSource on page 3-0 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 erasures vector is a double-precision or logical binary column vector that indicates which bits of the input codewords to erase or ignore. Values of `1` in the erasures vector correspond to erased bits in the same position of the (possibly punctured) input codewords. Set this property to `false` to disable erasures. The default is `false`.

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`.

Input and Output Signal Lengths in BCH and RS System Objects

The notation `y = c * x` denotes that `y` is an integer multiple of `x`.

The number of punctures equals the number of zeros in the puncture vector.

M is the degree of the primitive polynomial. Each group of *M* bits represents an integer between `0` and 2^M–`1` that belongs to the finite Galois field `GF(`2^M`)`.

ShortMessageLengthSource	comm.BCHEncoder comm.RSEncoder (BitInput = false)	comm.BCHDecoder comm.RSDecoder (BitInput = false)	comm.RSEncoder (BitInput = true)	comm.RSDecoder (BitInput = true)
Auto	**Input Length:** c * MessageLength **Output Length:** c * (CodewordLength – number of punctures)	**Input Length:** c * (CodewordLength – number of punctures) **Output Length:** c * MessageLength **Erasures Length:** c * (CodewordLength – number of punctures)	**Input Length:** c * (MessageLength * M) **Output Length:** c * ((CodewordLength – number of punctures) * M)	**Input Length:** c * ((CodewordLength – number of punctures) * M) **Output Length:** c * (MessageLength * M) **Erasures Length:** c * (CodewordLength – number of punctures)

ShortMessageLengthSource	comm.BCHEncoder comm.RSEncoder (BitInput = false)	comm.BCHDecoder comm.RSDecoder (BitInput = false)	comm.RSEncoder (BitInput = true)	comm.RSDecoder (BitInput = true)
Property	**Input Length:** c * ShortMessageLength **Output Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)	**Input Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures) **Output Length:** c * ShortMessageLength **Erasures Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)	**Input Length:** c * (ShortMessageLength * M) **Output Length:** c * ((CodewordLength - MessageLength + ShortMessageLength - number of punctures) * M)	**Input Length:** c * ((CodewordLength - MessageLength + ShortMessageLength - number of punctures) * M) **Output Length:** c * (ShortMessageLength * M) **Erasures Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)

Methods

step	Decode data using a BCH decoder

Common to All System Objects	
release	Allow System object property value changes

Examples

Transmit and decode an 8-DPSK-modulated signal, then count errors

```
% The following code transmits a BCH-encoded, 8-DPSK-modulated bit stream
% through an AWGN channel.  Then, the example demodulates, decodes, and counts errors.

enc = comm.BCHEncoder;
mod = comm.DPSKModulator('BitInput',true);
chan = comm.AWGNChannel(...
         'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
demod = comm.DPSKDemodulator('BitOutput',true);
dec = comm.BCHDecoder;
errorRate = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(enc, data);
  modSignal = step(mod, encodedData);
  receivedSignal = step(chan, modSignal);
  demodSignal = step(demod, receivedSignal);
  receivedBits = step(dec, demodSignal);
  errorStats = step(errorRate, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))

Error rate = 0.015075
Number of errors = 9
```

Transmit and receive a BPSK-modulated signal

Transmit and receive a BPSK-modulated signal encoded with a shortened BCH code, then count errors.

Specify the codeword, message, and shortened message lengths.

```
N = 255;
K = 239;
S = 63;
```

Create a BCH (255,239) generator polynomial. Use the generator polynomial to create a BCH encoder and decoder pair. The BCH code is based on the AMR standard.

```
gp = bchgenpoly(255,239);
bchEncoder = comm.BCHEncoder(N,K,gp,S);
bchDecoder = comm.BCHDecoder(N,K,gp,S);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Main processing loop.

```
for counter = 1:20
  data = randi([0 1],630,1);                 % Generate binary data
  encodedData = bchEncoder(data);            % BCH encode data
  modSignal = pskmod(encodedData,2);         % BPSK modulate
  receivedSignal = awgn(modSignal,5);        % Pass through AWGN channel
  demodSignal = pskdemod(receivedSignal,2);  % BSPK demodulate
  receivedBits = bchDecoder(demodSignal);    % BCH decode data
  errorStats = errorRate(data,receivedBits); % Compute error statistics
end
```

Display the error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.000318
Number of errors = 4
```

Shorten a BCH Code

Shorten a (31,26) BCH code to an (11,6) BCH code and use it to encode and decode random binary data.

Create a BCH encoder and decoder pair for a (31,26) code. Specify the generator polynomial, $x^5 + x^2 + 1$, and a shortened message length of 6.

```
enc = comm.BCHEncoder(31,26,'x5+x2+1',6);
dec = comm.BCHDecoder(31,26,'x5+x2+1',6);
```

Encode and decode random binary data and verify that the decoded bit stream matches the original data.

```
x = randi([0 1],60,1);
y = step(enc,x);
z = step(dec,y);
isequal(x,z)
```

```
ans = logical
   1
```

Selected Bibliography

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

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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`bchdec` | `bchgenpoly` | `comm.BCHEncoder` | `comm.RSDecoder` | `primpoly`

step

System object: comm.BCHDecoder
Package: comm

Decode data using a BCH decoder

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
Y = step(H,X,ERASURES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111.

`[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 *i*-th 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 *i*-th 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 obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Encode data using BCH encoder

Description

The BCHEncoder object creates a BCH code with specified message and codeword lengths.

To encode data using a BCH coding scheme:

1. Define and set up your BCH encoder object. See "Construction" on page 3-120.
2. Call step to create a BCH code with message and codeword lengths specified according to the properties of comm.BCHEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

enc = comm.BCHEncoder creates a BCH encoder System object, enc, that performs BCH encoding.

enc = comm.BCHEncoder(N,K) creates a BCH encoder object, enc, with the CodewordLength property set to N and the MessageLength property set to K.

enc = comm.BCHEncoder(N,K,GP) creates a BCH encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K and the GeneratorPolynomial property set to GP.

enc = comm.BCHEncoder(N,K,GP,S) creates a BCH encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, the

GeneratorPolynomial property set to GP and the ShortMessageLength property set to S.

enc = comm.BCHEncoder(N,K,GP,S,Name,Value) creates a BCH encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, the ShortMessageLength property set to S, and each specified property Name set to the specified Value.

enc = comm.BCHEncoder(Name,Value) creates a BCH encoder object, enc, 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

Note The input and output signal lengths are listed in "Input and Output Signal Lengths in BCH and RS System Objects" on page 3-111 on the comm.BCHDecoder reference page.

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 on page 3-0 properties must produce a valid narrow-sense BCH code. For a full-length BCH code, the value of the property must use the form $2^M - 1$, where M is an integer such that $3 \leq M \leq 16$. The default is 15.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar. The values of the CodewordLength on page 3-0 and MessageLength properties must produce a valid narrow-sense BCH code. The default is 5.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as either `Auto` or `Property`. When this property is set to `Auto`, the BCH code is defined by the `CodewordLength` on page 3-0 , `MessageLength` on page 3-0 , `GeneratorPolynomial` on page 3-0 , and `PrimitivePolynomial` on page 3-0 properties. When `ShortMessageLengthSource` is set to `Property`, you must specify the `ShortMessageLength` on page 3-0 property that is used with the other properties to define the RS code. The default is `Auto`.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to `MessageLength` on page 3-0 . When `ShortMessageLength` < `MessageLength`, the BCH code is shortened. The default is `5`.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as either `Auto` or `Property`. Set this property to `Auto` to create the generator polynomial automatically. Set it to `Property` to specify a generator polynomial using the `GeneratorPolynomial` on page 3-0 property. The default is `Auto`.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial as a binary double-precision row vector, a binary Galois row vector that represents the coefficients of the generator polynomial in order of descending powers, or as a polynomial character vector. The length of the generator polynomial requires a value of `CodewordLength` on page 3-0 -`MessageLength` on page 3-0 +`1`. This property applies when you set `GeneratorPolynomialSource` on page 3-0 to `Property`. The default is `'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'`, which is the result of `bchgenpoly(15,5,[],'double')` and 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. This check verifies that the specified generator polynomial is valid. 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 `GeneratorPolynomialSource` on page 3-0 to `Property`. The default is `true`.

`PrimitivePolynomialSource`

Source of primitive polynomial

Specify the source of the primitive polynomial as one of `Auto` or `Property`. Set this property to `Auto` to create a primitive polynomial of degree *M*=`ceil(log2(CodewordLength on page 3-0 +1))`. Set it to `Property` to specify a polynomial using the `PrimitivePolynomial` on page 3-0 property. The default is `Auto`

`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 or as a polynomial character vector. This property applies when you set the `PrimitivePolynomialSource` on page 3-0 property to `Property`. The default is `'X^4 + X + 1'`, which is the result of `fliplr(de2bi(primpoly(4)))`.

`PuncturePatternSource`

Source of puncture pattern

Specify the source of the puncture pattern as one of `None` or `Property`. Set this property to `None` to disable puncturing. Set it to `Property` to decode punctured codewords. This decoding is based on a puncture pattern vector you specify in the `PuncturePattern` on page 3-0 property. The default is `None`.

`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` on page 3-0 - `MessageLength` on page 3-0 . 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 `PuncturePatternSource` on page 3-0 to `Property`. The default is `[ones(8,1); zeros(2,1)]`.

Methods

step	Encode data using a BCH encoder

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit and decode an 8-DPSK-modulated signal, then count errors

```
% The following code transmits a BCH-encoded, 8-DPSK-modulated bit stream
% through an AWGN channel.  Then, the example demodulates, decodes, and counts errors.

enc = comm.BCHEncoder;
mod = comm.DPSKModulator('BitInput',true);
chan = comm.AWGNChannel(...
         'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
demod = comm.DPSKDemodulator('BitOutput',true);
dec = comm.BCHDecoder;
errorRate = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(enc, data);
  modSignal = step(mod, encodedData);
  receivedSignal = step(chan, modSignal);
  demodSignal = step(demod, receivedSignal);
  receivedBits = step(dec, demodSignal);
  errorStats = step(errorRate, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.015075
Number of errors = 9
```

Transmit and receive a BPSK-modulated signal

Transmit and receive a BPSK-modulated signal encoded with a shortened BCH code, then count errors.

Specify the codeword, message, and shortened message lengths.

```
N = 255;
K = 239;
S = 63;
```

Create a BCH (255,239) generator polynomial. Use the generator polynomial to create a BCH encoder and decoder pair. The BCH code is based on the AMR standard.

```
gp = bchgenpoly(255,239);
bchEncoder = comm.BCHEncoder(N,K,gp,S);
bchDecoder = comm.BCHDecoder(N,K,gp,S);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Main processing loop.

```
for counter = 1:20
  data = randi([0 1],630,1);                 % Generate binary data
  encodedData = bchEncoder(data);            % BCH encode data
  modSignal = pskmod(encodedData,2);         % BPSK modulate
  receivedSignal = awgn(modSignal,5);        % Pass through AWGN channel
  demodSignal = pskdemod(receivedSignal,2);  % BSPK demodulate
  receivedBits = bchDecoder(demodSignal);    % BCH decode data
  errorStats = errorRate(data,receivedBits); % Compute error statistics
end
```

Display the error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.000318
Number of errors = 4
```

Shorten a BCH Code

Shorten a (31,26) BCH code to an (11,6) BCH code and use it to encode and decode random binary data.

Create a BCH encoder and decoder pair for a (31,26) code. Specify the generator polynomial, $x^5 + x^2 + 1$, and a shortened message length of 6.

```
enc = comm.BCHEncoder(31,26,'x5+x2+1',6);
dec = comm.BCHDecoder(31,26,'x5+x2+1',6);
```

Encode and decode random binary data and verify that the decoded bit stream matches the original data.

```
x = randi([0 1],60,1);
y = step(enc,x);
z = step(dec,y);
isequal(x,z)
```

```
ans = logical
   1
```

Selected Bibliography

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

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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`bchenc` | `bchgenpoly` | `comm.BCHDecoder` | `comm.RSEncoder` | `primpoly`

step

System object: comm.BCHEncoder
Package: comm

Encode data using a BCH encoder

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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`. The input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BitToInteger System object

Package: comm

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.

To map bits to integers:

1. Define and set up your bit to integer object. See "Construction" on page 3-129.
2. Call `step` to map groups of bits in the input vector to integers in the output vector according to the properties of `comm.BitToInteger`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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` on page 3-0 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

step	Convert vector of bits to vector of integers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convert random 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)
```

```
intData = 3×1

    13
     9
    13
```

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

`bi2de` | `bin2dec` | `comm.IntegerToBit`

Introduced in R2012a

step

System object: comm.BitToInteger
Package: comm

Convert vector of bits to vector of integers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Introduce binary errors

Description

The `BinarySymmetricChannel` object introduces binary errors to the signal transmitted through this channel.

To introduce binary errors into the transmitted signal:

1. Define and set up your binary symmetric channel object. See "Construction" on page 3-135.
2. Call `step` to introduces binary errors into the signal transmitted through this channel according to the properties of `comm.ACPR`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Introduce binary errors

Common to All System Objects	
release	Allow System object property value changes

Examples

Add Errors to Binary Input Signal

Add binary errors with a probability of 0.2 to a binary input signal

```
binSymChan = comm.BinarySymmetricChannel('ErrorProbability',0.2);
data = randi([0 1],1000,1);
[~,err] = binSymChan(data);
```

Confirm that the number errors is approximately equal to the 0.2 multiplied by the number of symbols.

```
[sum(err) 0.2*length(data)]
```

```
ans = 1×2

   188   200
```

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`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AWGNChannel`

Introduced in R2012a

step

System object: comm.BinarySymmetricChannel
Package: comm

Introduce binary errors

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BlockDeinterleaver System object

Package: comm

Deinterleave input symbols using permutation vector

Description

The `BlockDeinterleaver` object, which can process variable-sized signals, rearranges the elements of its input vector without repeating or omitting any elements. The input can be real or complex.

To deinterleave the input vector:

1. Define and set up your block deinterleaver object. See “Construction” on page 3-141.
2. Call `step` to rearrange the elements of the input vector according to the properties of `comm.BlockDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

`PermutationVectorSource`

Permutation vector source

Specify the source of the permutation vector as either `Property` or `Input port`. The default value is `Property`.

`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. The `PermutationVector` property is available only when the `PermutationVectorSource` property is set to `Property`.

Methods

step	Deinterleave input symbols using permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Block Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data, intData, deIntData]
```

```
ans = 4×3

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver(permVec);
deinterleaver = comm.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BlockInterleaver` | `comm.MatrixDeinterleaver`

Introduced in R2012a

step

System object: comm.BlockDeinterleaver
Package: comm

Deinterleave input symbols using permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BlockInterleaver System object

Package: comm

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. This object processes variable-size signals.

To interleave the input signal:

1. Define and set up your block interleaver object. See “Construction” on page 3-146.
2. Call `step` to reorder the input symbols according to the properties of `comm.BlockInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

`PermutationVectorSource`

Permutation vector source

Specify the source of the permutation vector as either `Property` or `Input port`. The default value is `Property`.

`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. The `PermutationVector` property is available only when the `PermutationVectorSource` property is set to `Property`.

Methods

step	Permute input symbols using a permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Block Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data, intData, deIntData]
```

```
ans = 4×3

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver(permVec);
deinterleaver = comm.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)

ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BlockDeinterleaver` | `comm.MatrixInterleaver`

Introduced in R2012a

step

System object: comm.BlockInterleaver
Package: comm

Permute input symbols using a permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BPSKDemodulator System object

Package: comm

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.

To demodulate a binary phase shift signal:

1. Define and set up your BPSK demodulator object. See “Construction” on page 3-151.
2. Call `step` to demodulate a signal according to the properties of `comm.BPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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 on page 3-0 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` on page 3-0 property to `Hard decision`. Thus, when you set the `OutputDataType` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Hard decision` and the `DerotateFactorDataType` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using BPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate BPSK Signal and Calculate Errors

Generate a BPSK signal, pass it through an AWGN channel, demodulate the signal, and compute the error statistics.

Create BPSK modulator and demodulator System objects.

```
bpskModulator = comm.BPSKModulator;
bpskDemodulator = comm.BPSKDemodulator;
```

Create an error rate calculator System object.

```
errorRate = comm.ErrorRate;
```

Generate 50-bit random data frames, apply BPSK modulation, pass the signal through an AWGN channel, demodulate the received data, and compile the error statistics.

```
for counter = 1:100
    % Transmit a 50-symbol frame
    txData = randi([0 1],50,1);           % Generate data
    modSig = bpskModulator(txData);       % Modulate
    rxSig = awgn(modSig,5);               % Pass through AWGN
    rxData = bpskDemodulator(rxSig);      % Demodulate
    errorStats = errorRate(txData,rxData); % Collect error stats
end
```

Display the cumulative error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 0.005600
Number of errors = 28
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BPSKModulator` | `comm.PSKDemodulator`

Introduced in R2012a

constellation

System object: comm.BPSKDemodulator
Package: comm

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 Reference Signal Constellation for BPSK Demodulator

Create BPSK Demodulator System object™ and calculate its reference constellation.

Create a `comm.BPSKDemodulator` System object.

```
h = comm.BPSKDemodulator;
```

Calculate and display the reference signal constellation by calling the `constellation` function.

```
refC = constellation(h)
```

```
refC = 2×1 complex
```

```
   1.0000 + 0.0000i
  -1.0000 + 0.0000i
```

Plot BPSK Demodulator Reference Signal Constellation

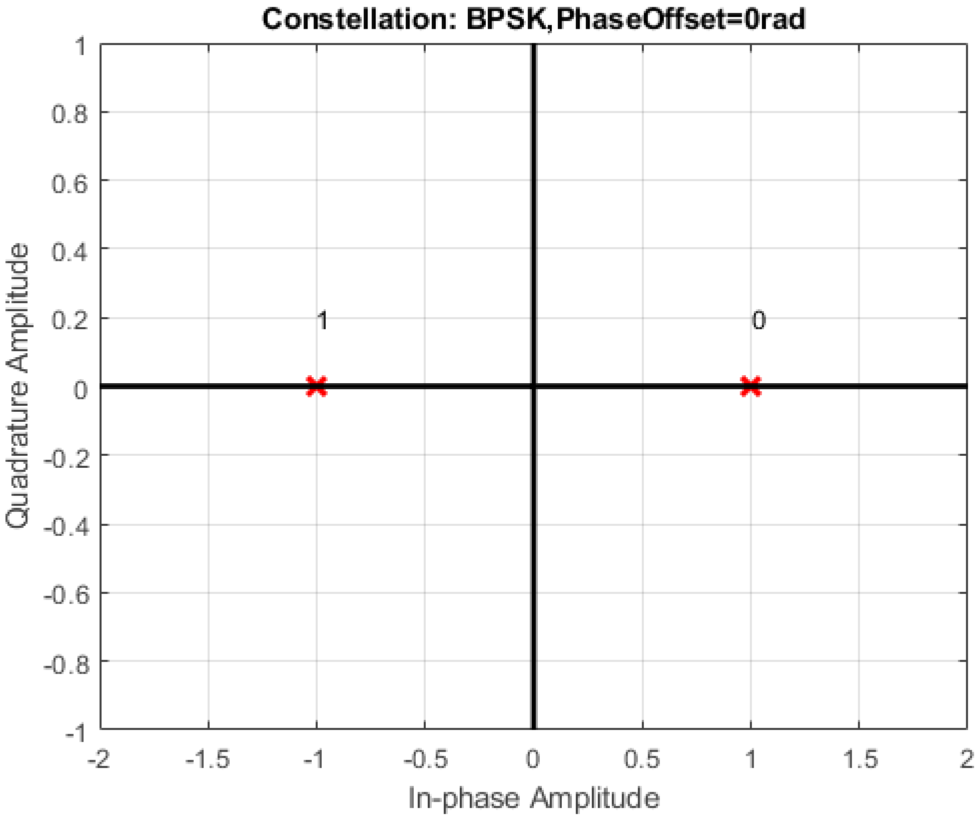
Create a BPSK Demodulator System object™ and then plot the reference signal constellation.

Create a `comm.BPSKDemodulator` System object.

```
h = comm.BPSKDemodulator;
```

Plot the reference constellation by calling the `constellation` function.

```
constellation(h)
```


Constellation: BPSK,PhaseOffset=0rad
Quadrature Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
In-phase Amplitude
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
0

step

System object: comm.BPSKDemodulator
Package: comm

Demodulate using BPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.IQImbalanceCompensator System object

Package: comm

Compensate for I/Q imbalance

Description

The `IQImbalanceCompensator` System object compensates for the imbalance between the in-phase and quadrature components of a modulated signal.

To compensate for I/Q imbalance:

1. Define and set up the `IQImbalanceCompensator` object. See “Construction” on page 3-162.
2. Call `step` to compensate for the I/Q imbalance according to the properties of `comm.IQImbalanceCompensator`. The behavior of `step` is specific to each object in the toolbox.

The adaptive algorithm inherent to the I/Q imbalance compensator is compatible with M-PSK, M-QAM, and OFDM modulation schemes, where *M>2*.

Note The output of the compensator might be scaled and rotated, that is, multiplied by a complex number, relative to the reference constellation. In practice, this is not an issue as receivers correct for this prior to demodulation through the use of channel estimation.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.IQImbalanceCompensator` creates a compensator System object, `H`, that compensates for the imbalance between the in-phase and quadrature components of the input signal.

`H = comm.IQImbalanceCompensator(Name,Value)` creates an I/Q imbalance compensator 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

`CoefficientSource`

Source of compensator coefficients

Specify either `Estimated from input signal` or `Input port`. If the `CoefficientSource` property is set to `Estimated from input signal`, the compensator uses an adaptive algorithm to estimate the compensator coefficient from the input signal. If the `CoefficientSource` property is set to `Input port`, all other properties are disabled and the compensator coefficients must be provided to the `step` function as an input argument. The default value is `Estimated from input signal`. This property is nontunable.

`InitialCoefficent`

Initial coefficient used to compensate for I/Q imbalance

The initial coefficient is a complex scalar that can be either single or double precision. The default value is `0+0i`. This property is nontunable.

`StepSizeSource`

Source of step size for coefficient adaptation

Specify either `Property` or `Input port`. If `StepSizeSource` is set to `Property`, you specify the step size through the `StepSize` property. Otherwise, the step size is provided to the `step` function as an input argument. The default value is `Property`. This property is nontunable.

StepSize

Adaptation step size

Specifies the step size used by the algorithm in estimating the I/Q imbalance. This property is accessible only when `StepSizeSource` is set to `Property`. The default value is `1e-5`. This property is tunable.

AdaptInputPort

Creates input port to control compensator coefficient adaptation

When this logical property is `true`, an input port is created to enable or disable coefficient adaptation. If `AdaptInputPort` is `false`, the coefficients update after each output sample. The default value is `false`. This property is nontunable.

CoefficientOutputPort

Create port to output compensator coefficients

When this logical property is `true`, the I/Q imbalance compensator coefficients are made available through an output argument of the `step` function. The default value is `false`. This property is nontunable.

Methods

step	Compensate I/Q Imbalance
reset	Reset states of the `IQImbalanceCompensator` System object

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Remove I/Q Imbalance from a QPSK Signal

Mitigate the impacts of amplitude and phase imbalance on a QPSK modulated signal by using the comm.IQImbalanceCompensator System object™.

Create a constellation diagram object. Specify name-value pairs to ensure that the constellation diagram displays only the last 100 data symbols.

```
constDiagram = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',100);
```

Create an I/Q imbalance compensator.

```
iqImbComp = comm.IQImbalanceCompensator;
```

Generate random data symbols and apply QPSK modulation.

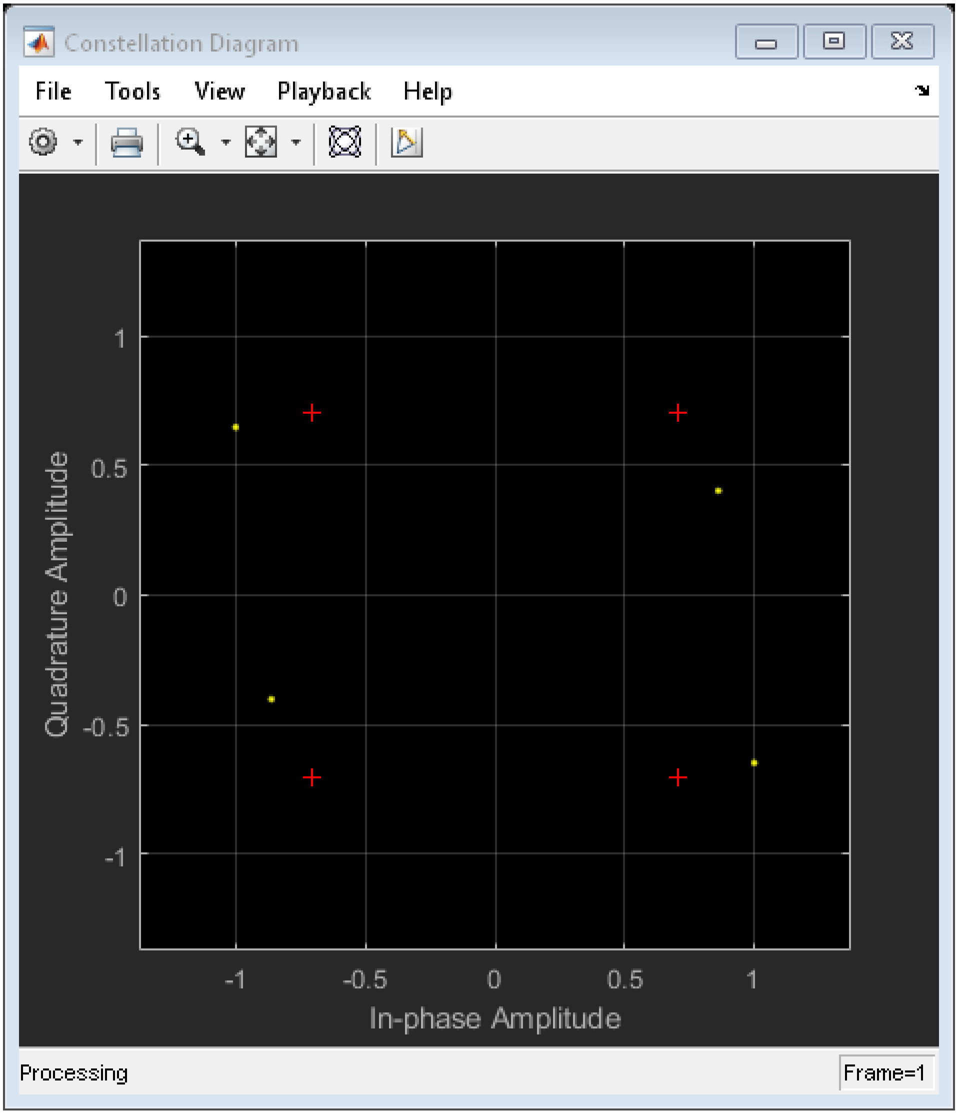
```
data = randi([0 3],1e7,1);
txSig = pskmod(data,4,pi/4);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

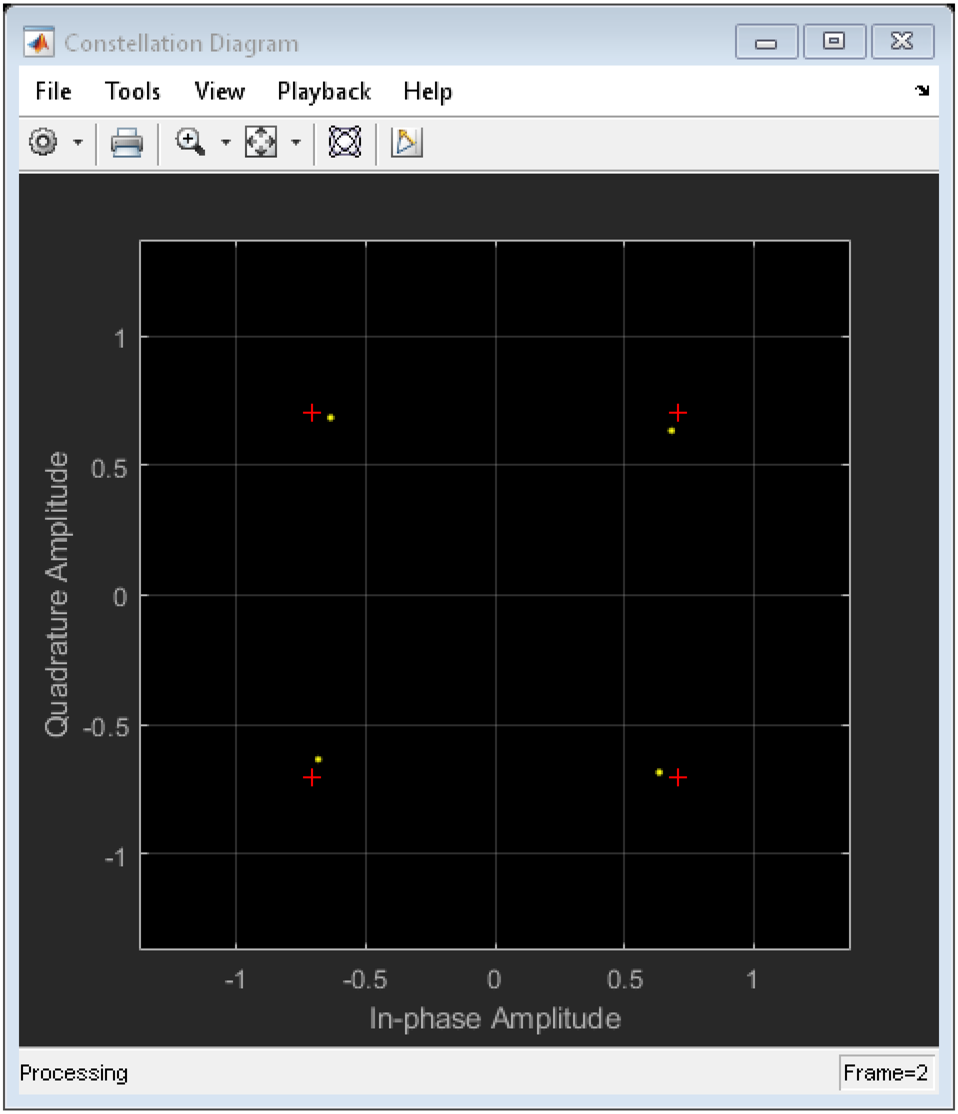
Plot the constellation diagram of the received signal. Observe that the received signal experienced an amplitude and phase shift.

```
constDiagram(rxSig)
```



Apply the I/Q compensation algorithm and view the constellation. The compensated signal constellation is nearly aligned with the reference constellation.

```
compSig = iqImbComp(rxSig);
constDiagram(compSig)
```



Constellation Diagram
File
Tools
View
Playback
Help
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=2

Remove I/Q Imbalance from an 8-PSK Signal using External Coefficients

Compensate for an amplitude and phase imbalance on an 8-PSK signal by using the `comm.IQImbalanceCompensator` System object™ with external coefficients.

Create 8-PSK modulator and constellation diagram System objects. Use name-value pairs to ensure that the constellation diagram displays only the last 100 data symbols and to provide the reference constellation.

```
hMod = comm.PSKModulator(8);
refC = constellation(hMod);
hScope = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',100, ...
    'ReferenceConstellation',refC);
```

Create an I/Q imbalance compensator object with an input port for the algorithm coefficients.

```
hIQComp = comm.IQImbalanceCompensator('CoefficientSource','Input port');
```

Generate random data symbols and apply 8-PSK modulation.

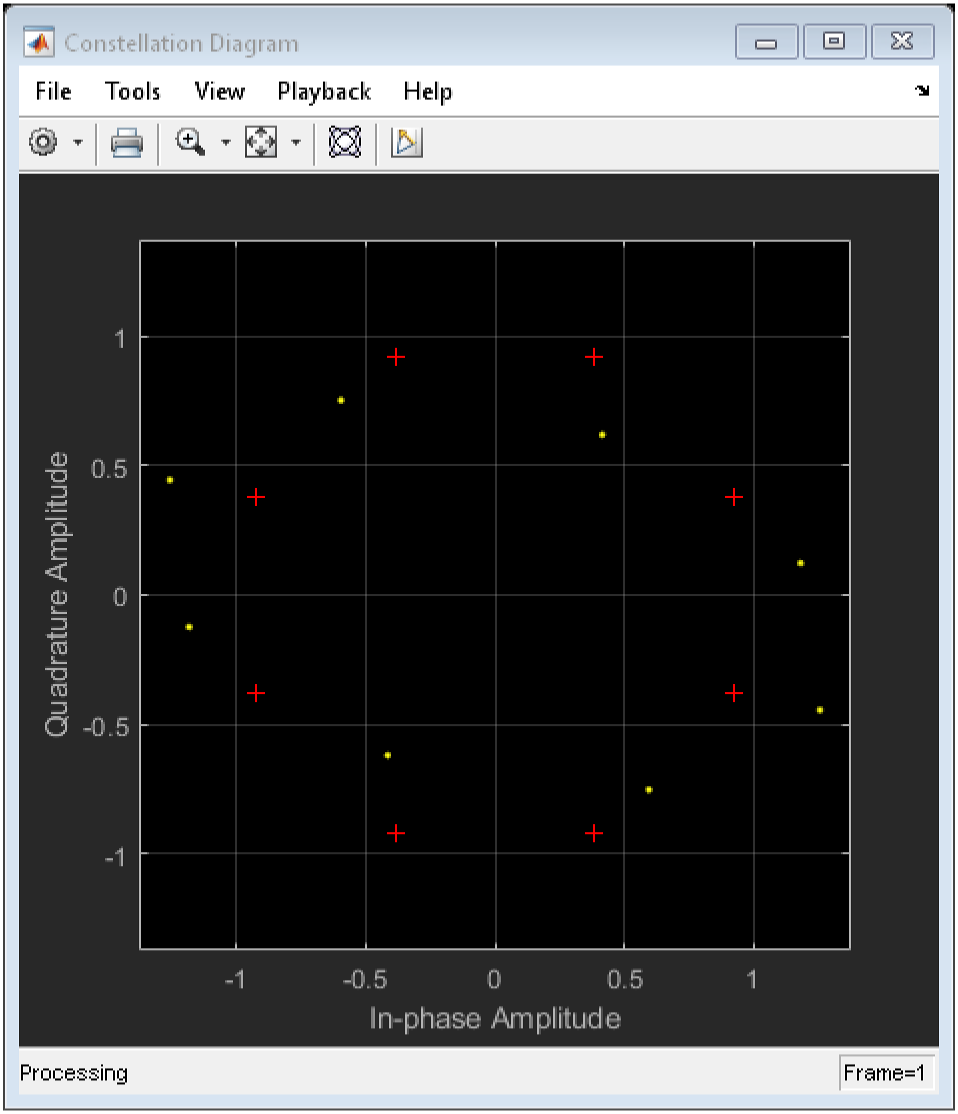
```
data = randi([0 7],1000,1);
txSig = step(hMod,data);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the constellation diagram of the received signal. Observe that the received signal experienced an amplitude and phase shift.

```
step(hScope,rxSig);
```

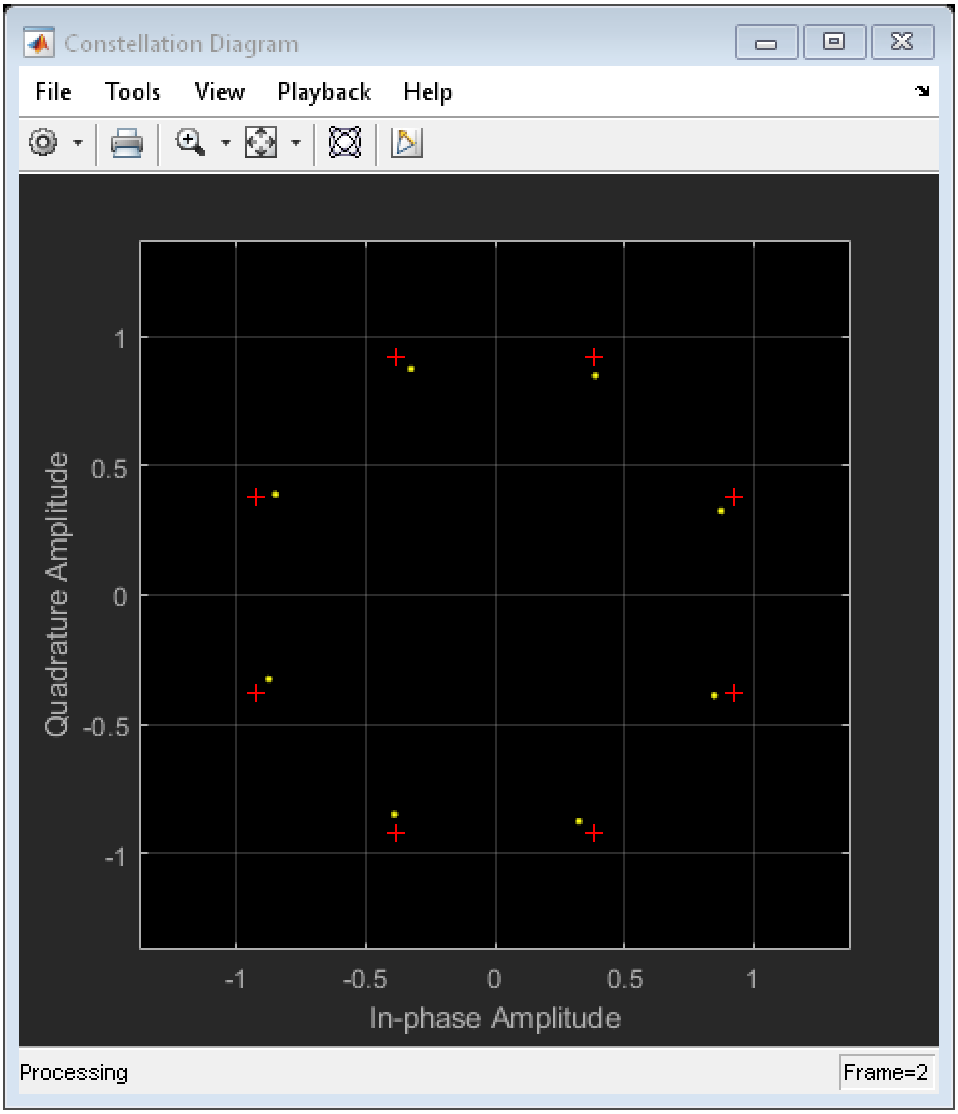


Use the `iqimbal2coef` function to determine the compensation coefficient given the amplitude and phase imbalance.

```
compCoef = iqimbal2coef(ampImb,phImb);
```

Apply the compensation coefficient to the received signal using the `step` function of the `comm.IQImbalanceCompensator` object and view the resultant constellation. You can see that the compensated signal constellation is now nearly aligned with the reference constellation.

```
compSig = step(hIQComp,rxSig,compCoef);
step(hScope,compSig)
```


Constellation Diagram
File
Tools
View
Playback
Help
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=2

Remove I/Q Imbalance from a QAM Signal

Remove an I/Q imbalance from a 64-QAM signal and to make the estimated coefficients externally available while setting the algorithm step size from an input port.

Create a constellation diagram object. Use name-value pairs to ensure that the constellation diagram displays only the last 256 data symbols, set the axes limits, and specify the reference constellation.

```
M = 64;
refC = qammod(0:M-1,M);
constDiagram = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',256, ...
    'XLimits',[-10 10],'YLimits',[-10 10], ...
    'ReferenceConstellation',refC);
```

Create an I/Q imbalance compensator System object in which the step size is specified as an input argument and the estimated coefficients are made available through an output port.

```
iqImbComp = comm.IQImbalanceCompensator('StepSizeSource','Input port', ...
    'CoefficientOutputPort',true);
```

Generate random data symbols and apply 64-QAM modulation.

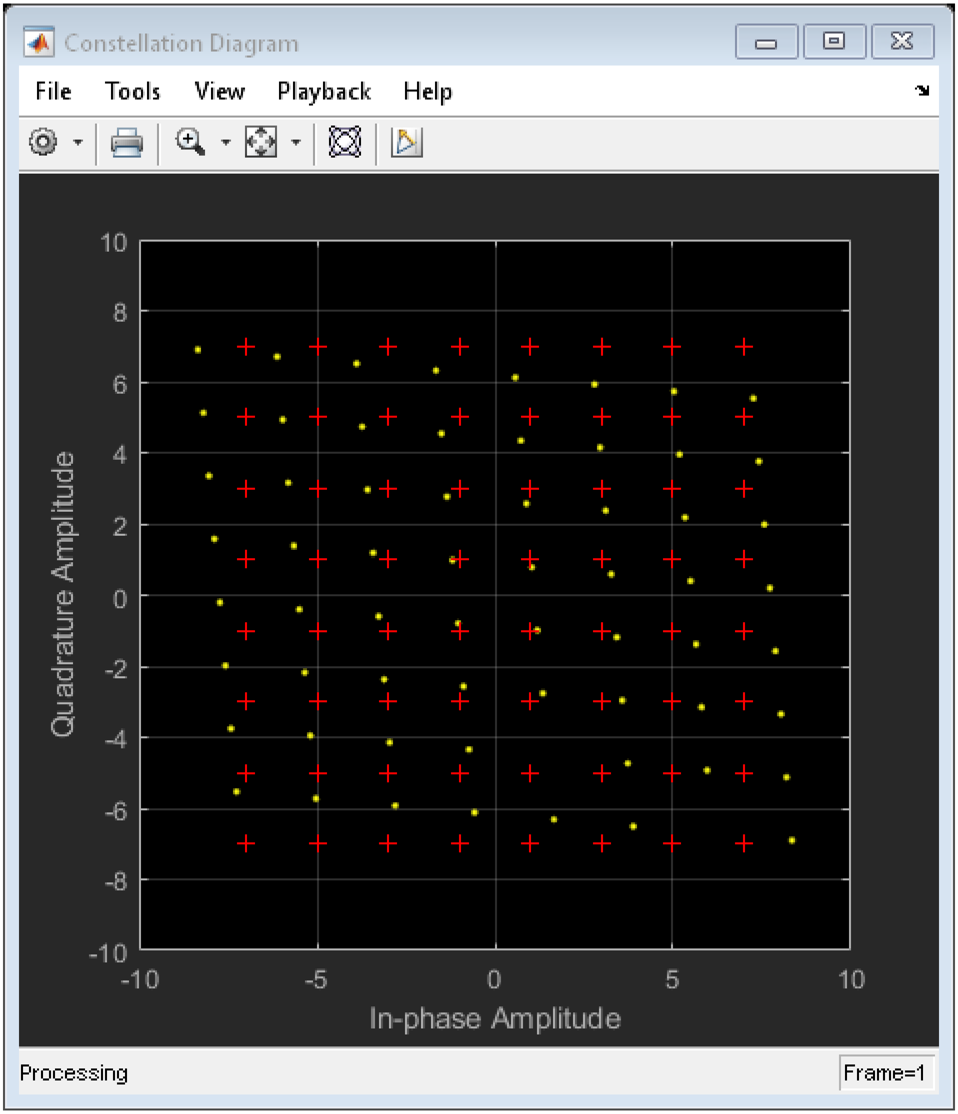
```
nSym = 25000;
data = randi([0 M-1],nSym,1);
txSig = qammod(data,M);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 2; % dB
phImb = 10; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the constellation diagram of the received signal.

```
constDiagram(rxSig);
```

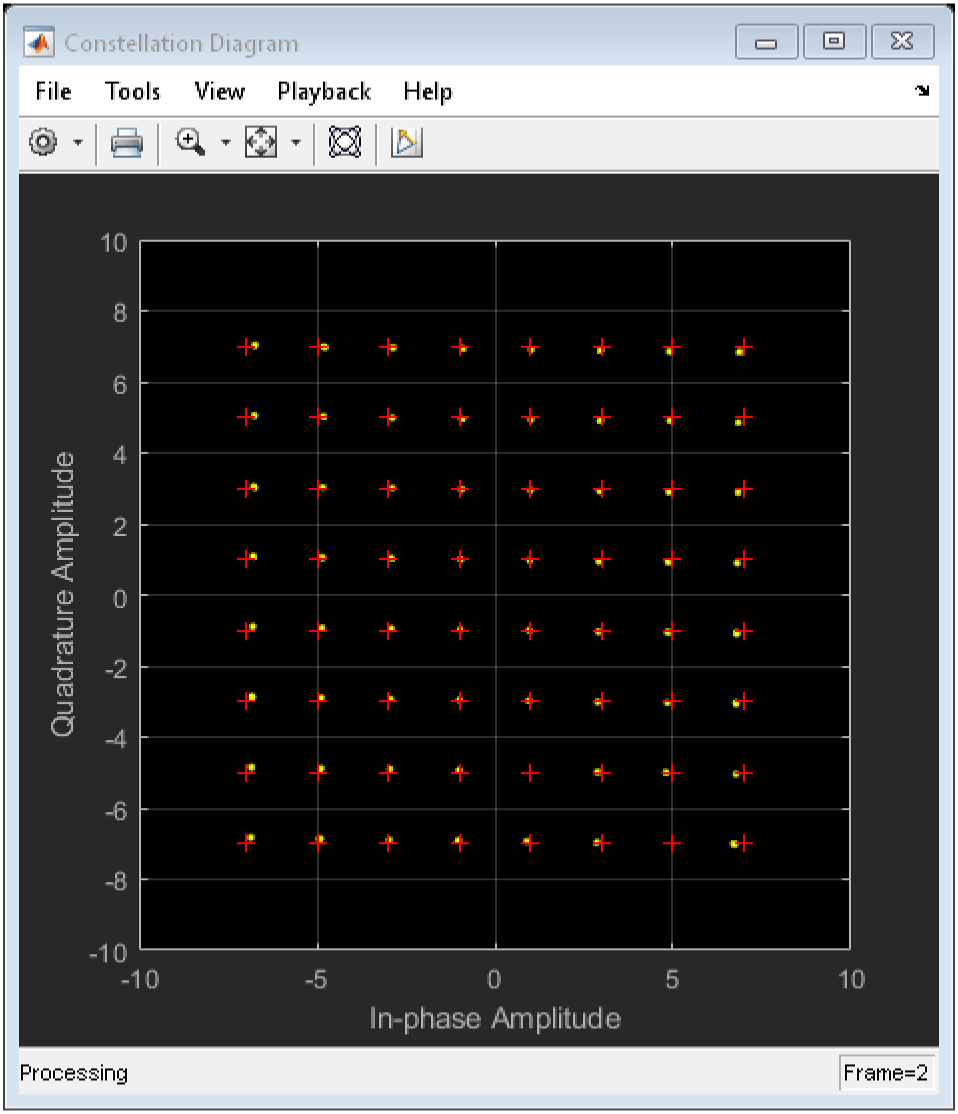


Specify the step size parameter for the I/Q imbalance compensator.

```
stepSize = 1e-5;
```

Compensate for the I/Q imbalance while setting the step size via an input argument. You can see that the compensated signal constellation is now nearly aligned with the reference constellation.

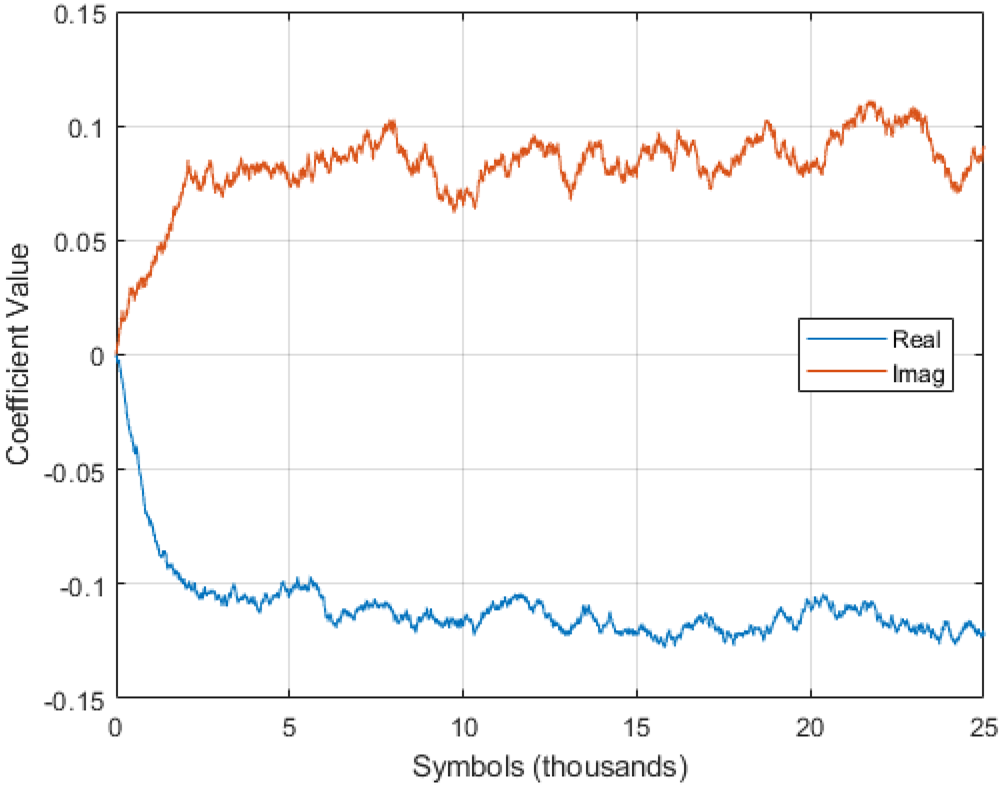
```
[compSig,estCoef] = iqImbComp(rxSig,stepSize);
constDiagram(compSig)
```



Plot the real and imaginary values of the estimated coefficients. You can see that they reach a steady-state solution.

```
plot((1:nSym)'/1000,[real(estCoef),imag(estCoef)])
grid
```

```
xlabel('Symbols (thousands)')
ylabel('Coefficient Value')
legend('Real','Imag','location','best')
```



Control Adaptation Algorithm for I/Q Imbalance Compensator

Control the adaptation algorithm of the I/Q imbalance compensator using an external argument.

Apply QPSK modulation to random data symbols.

```
data = randi([0 3],600,1);
txSig = pskmod(data,4,pi/4,'gray');
```

Create an I/Q imbalance compensator in which the adaptation algorithm is controlled through an input port, the step size is specified through the `StepSize` property, and the estimated coefficients are made available through an output port.

```
iqImbComp = comm.IQImbalanceCompensator('AdaptInputPort',true, ...
    'StepSize',0.001,'CoefficientOutputPort',true);
```

Apply amplitude and phase imbalance to the transmitted signal.

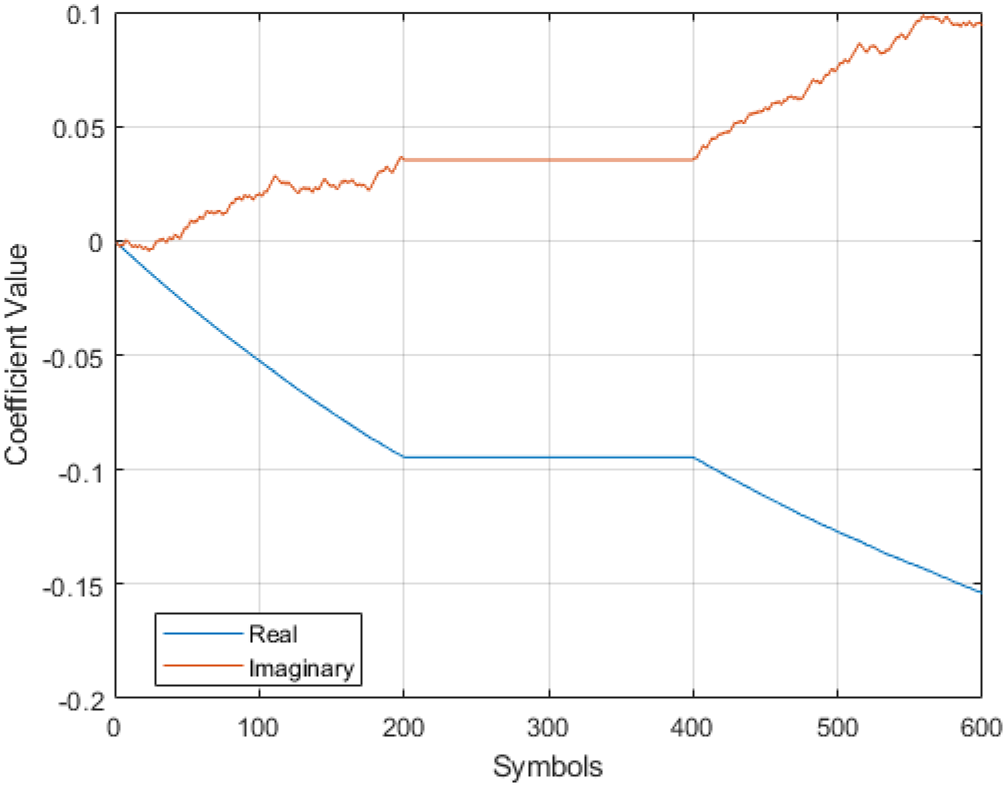
```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Break the compensation operation into three segments in which the compensator is enabled for the first 200 symbols, disabled for the next 200 symbols, and enabled for the last 200 symbols. Save the coefficient data in three vectors.

```
[~,estCoef1] = iqImbComp(rxSig(1:200),true);
[~,estCoef2] = iqImbComp(rxSig(201:400),false);
[~,estCoef3] = iqImbComp(rxSig(401:600),true);
```

Concatenate the complex algorithm coefficients and plot their real and imaginary parts.

```
estCoef = [estCoef1; estCoef2; estCoef3];
plot((1:600)',[real(estCoef) imag(estCoef)])
grid
xlabel('Symbols')
ylabel('Coefficient Value')
legend('Real','Imaginary','location','best')
```

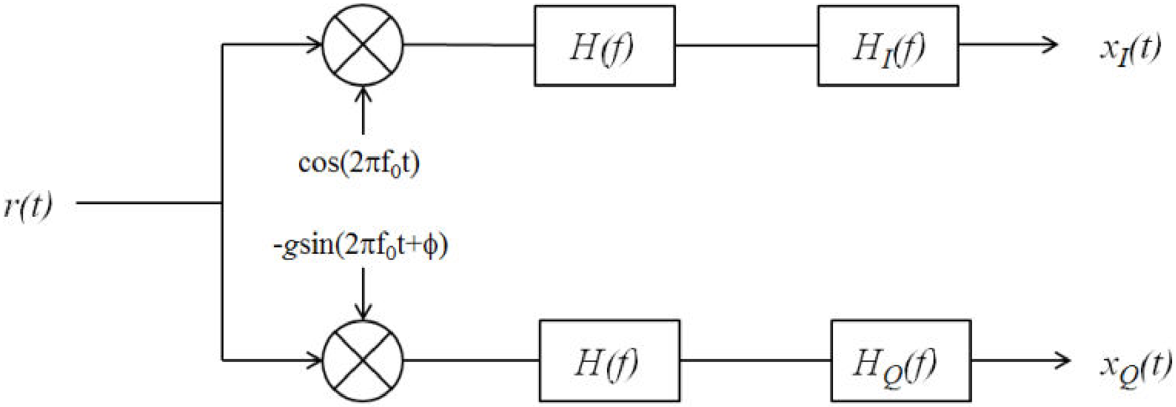


Observe that the coefficients do not adapt during the time in which the compensator is disabled.

Algorithms

One of the major impairments affecting direct conversion receivers is the imbalance between the received signal's in-phase and quadrature components. Rather than improving the front-end, analog hardware, it is more cost effective to tolerate a certain level of I/Q imbalance and then implement compensation methods. A circularity-based blind compensation algorithm is used as the basis for the I/Q Imbalance Compensator.

A generalized I/Q imbalance model is shown, where g is the amplitude imbalance and ϕ is the phase imbalance (ideally, $g = 1$ and $\phi = 0$). In the figure, *H(f)* is the nominal frequency response of the branches due to, for example, lowpass filters. $H_I(f)$ and $H_Q(f)$ represent the portions of the in-phase and quadrature amplitude and phase responses that differ from the nominal response. With perfect matching, $H_I(f) = H_Q(f) = 1$.



Let *z(t)* be the ideal baseband equivalent signal of the received signal, *r(t)*, where its Fourier transform is denoted as *Z(f)*. Given the generalized I/Q imbalance model, the Fourier transform of the imbalanced signal, $x(t) = x_I(t) + x_Q(t)$, is

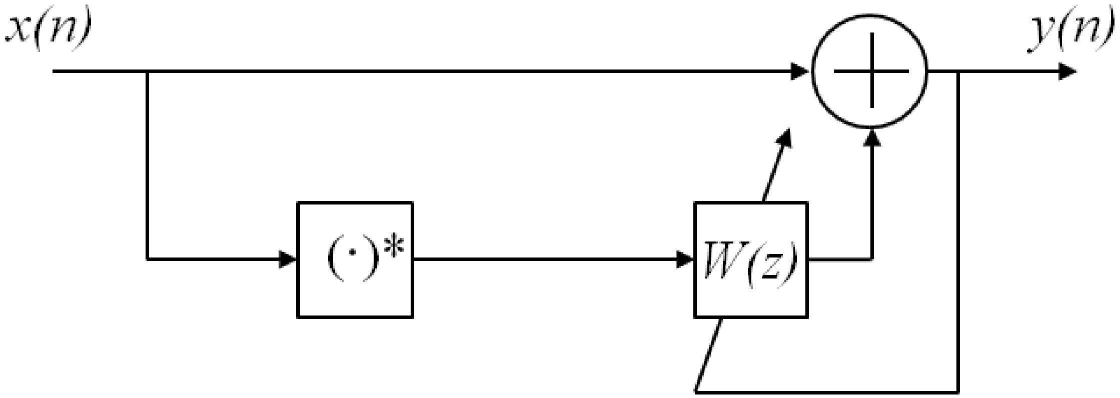
$$X(f) = G_1(f)Z(f) + G_2(f)Z^*(-f)$$

where $G_1(f)$ and $G_2(f)$ are the direct and conjugate components of the I/Q imbalance. These components are defined as

$$G_1(f) = \left[H_I(f) + H_Q(f)g\exp(-j\phi)\right]/2$$
$$G_2(f) = \left[H_I(f) + H_Q(f)g\exp(j\phi)\right]/2$$

Applying the inverse Fourier transform to *X(f)*, the signal model becomes $x(t) = g_1(t) * z(t) + g_2(t) * z^*(t)$.

This suggests the compensator structure as shown in which discrete-time notation is used to express the variables. The compensated signal is expressed as $y(n) = x(n) + wx^*(n)$.



A simple algorithm of the form

$$\begin{cases} y(n) = x(n) + w(n)x^*(n) \\ w(n+1) = w(n) - My^2(n) \end{cases}$$

is used to determine the weights, because it ensures that the output is “proper”, that is, $\mathrm{E}[y^2(n)] = 0$ [1]. The initial value of *w* is determined by the `InitialCoefficient` property, which has a default value of `0 + 0i`. *M* is the step size, as specified in the `StepSize` property.

Selected Bibliography

[1] Anttila, L., M. Valkama, and M. Renfors. “Blind compensation of frequency-selective I/Q imbalances in quadrature radio receivers: Circularity-based approach”, *Proc. IEEE ICASSP*, pp.III-245–248, 2007.

[2] Kiayani, A., L. Anttila, Y. Zou, and M. Valkama, “Advanced Receiver Design for Mitigating Multiple RF Impairments in OFDM Systems: Algorithms and RF Measurements”, *Journal of Electrical and Computer Engineering*, Vol. 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

I/Q Imbalance Compensator | `iqcoef2imbal` | `iqimbal2coef`

Introduced in R2014b

step

System object: comm.IQImbalanceCompensator
Package: comm

Compensate I/Q Imbalance

Syntax

```
Y = step(H,X)
Y = step(H,X,COEF)
Y = step(H,X,STEPSIZE)
Y = step(H,...,ADAPT)
[Y,ESTCOEF] = step(H,X)
[Y,ESTCOEF] = step(H,X,STEPSIZE)
[Y,ESTCOEF] = step(H,X,STEPSIZE,ADAPT)
[Y,ESTCOEF] = step(H,X,ADAPT)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the I/Q imbalance in the input signal, `X`, and returns a compensated signal, `Y`. The input `X` can take real or complex values and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` has the same properties as `X`.

`Y = step(H,X,COEF)` accepts input coefficients, `COEF`, instead of generating them internally. This syntax applies when the CoefficientSource property of `H` is set to `Input port`. The input coefficients, `COEF`, are complex and can be either double or single precision. `COEF` has the same dimensions as `X`.

`Y = step(H,X,STEPSIZE)` accepts a step size input, `STEPSIZE`. This syntax applies when the StepSizeSource property of `H` is set to `Input port`. The step size is a real scalar supporting either double or single precision.

`Y = step(H,...,ADAPT)` accepts a control signal, `ADAPT`, to enable or disable coefficient updates. This syntax applies when the AdaptInputPort property of `H` is `true`. The adaptation control signal is a logical scalar.

`[Y,ESTCOEF] = step(H,X)` outputs the estimated coefficients, `ESTCOEF`, when the CoefficientOutputPort property of `H` is `true`. `ESTCOEF` has the same data properties and dimensionality as the input signal, `X`.

`[Y,ESTCOEF] = step(H,X,STEPSIZE)` outputs the estimated coefficients, `ESTCOEF`, and accepts a step size input, `STEPSIZE`. This syntax applies when the properties of `H` are set so that CoefficientOutputPort is `true` and StepSizeSource is `Input port`.

`[Y,ESTCOEF] = step(H,X,STEPSIZE,ADAPT)` outputs the estimated coefficients, `ESTCOEF`, and accepts a step size input,`STEPSIZE`, and a control signal input, `ADAPT`. This syntax applies when the properties of `H` are set so that CoefficientOutputPort is `true`, StepSizeSource is `Input port`, and AdaptInputPort is `true`.

`[Y,ESTCOEF] = step(H,X,ADAPT)` outputs the estimated coefficients, `ESTCOEF`, and accepts a control signal input, `ADAPT`. This syntax applies when the properties of `H` are set so that CoefficientOutputPort is `true` and AdaptInputPort is `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

reset

System object: comm.IQImbalanceCompensator
Package: comm

Reset states of the `IQImbalanceCompensator` System object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `IQImbalanceCompensator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

comm.BPSKModulator System object

Package: comm

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.

To modulate a binary phase shift signal:

1. Define and set up your BPSK modulator object. See "Construction" on page 3-184.
2. Call `step` to modulate a signal according to the properties of `comm.BPSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using BPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

BPSK Data Scatter Plot

This example creates binary data, modulates it, and then displays the data using a scatter plot.

Create binary data symbols

```
data = randi([0 1],100,1);
```

Create a BPSK modulator System object

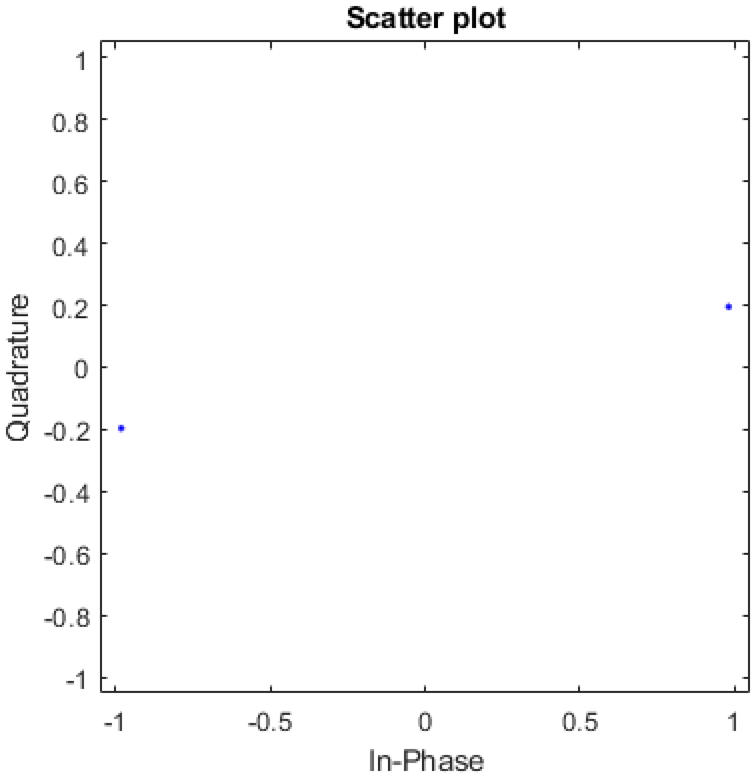
```
bpskModulator = comm.BPSKModulator;
```

Change the phase offset to pi/16

```
bpskModulator.PhaseOffset = pi/16;
```

Modulate and plot the data

```
modData = bpskModulator(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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BPSKDemodulator` | `comm.PSKModulator`

Introduced in R2012a

constellation

System object: comm.BPSKModulator
Package: comm

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 BPSK Modulator Reference Constellation

Create a BPSK Modulator System object™ and calculate the reference constellation values.

Create a `comm.BPSKModulator` System object.

```
h = comm.BPSKModulator;
```

Calculate and display the reference constellation values by calling the `constellation` function.

```
refC = constellation(h)
```

```
refC = 2×1 complex
```

```
   1.0000 + 0.0000i
  -1.0000 + 0.0000i
```

Plot BPSK Modulator Reference Constellation

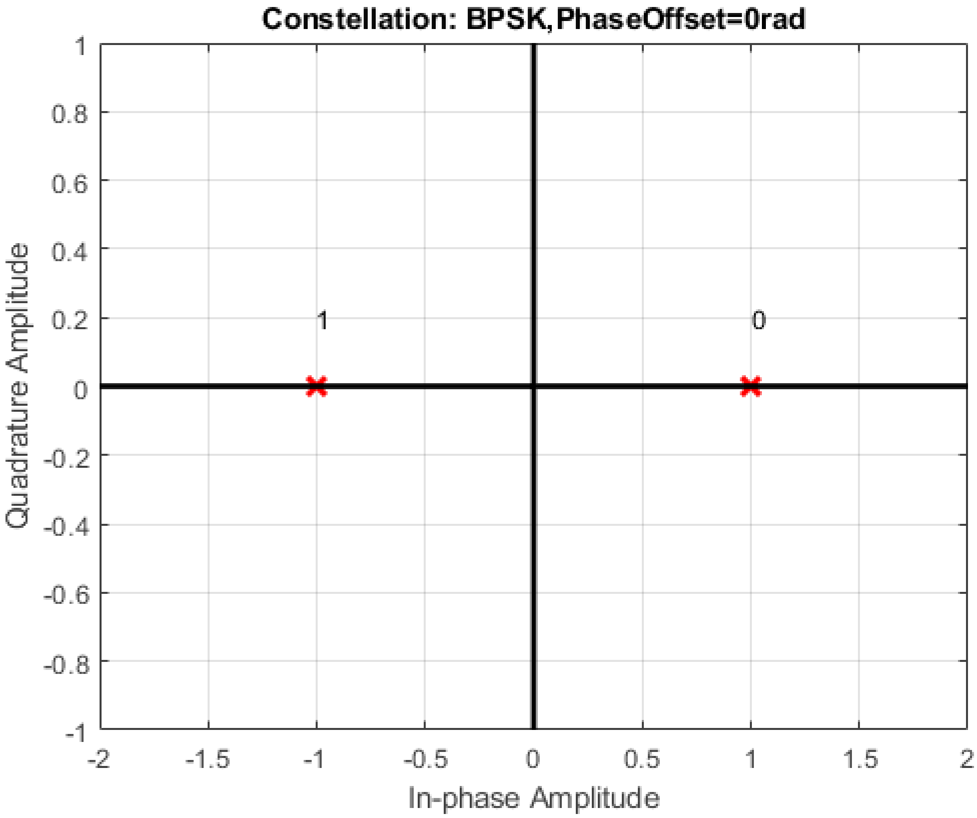
Create a BPSK Modulator System object™ and plot the reference constellation.

Create a `comm.BPSKModulator` System object.

```
bpsk = comm.BPSKModulator;
```

Plot the reference constellation by calling the `constellation` function.

```
constellation(bpsk)
```


Constellation: BPSK,PhaseOffset=0rad
Quadrature Amplitude
In-phase Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
0

step

System object: comm.BPSKModulator
Package: comm

Modulate using BPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.OFDMModulator System object

Package: comm

Modulate using OFDM method

Description

The `OFDMModulator` object modulates using the orthogonal frequency division modulation method. The output is a baseband representation of the modulated signal.

To modulate an OFDM signal:

1. Define and set up the OFDM modulator object. See “Construction” on page 3-193.
2. Call `step` to modulate a signal according to the properties of `comm.OFDMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.OFDMModulator` creates a modulator System object, `H`, that modulates the input signal using the orthogonal frequency division modulation (OFDM) method.

`H = comm.OFDMModulator(Name,Value)` creates a OFDM 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.OFDMModulator(hDemod)` creates an OFDM modulator object, `H`, whose properties are determined by the corresponding OFDM demodulator object, `hDemod`.

Properties

FFTLength

The length of the FFT, N_{FFT}, is equivalent to the number of subcarriers used in the modulation process. FFTLength must be ≥ 8.

Specify the number of subcarriers. The default is 64.

NumGuardBandCarriers

The number of guard band subcarriers allocated to the left and right guard bands.

Specify the number of left and right subcarriers as nonnegative integers from 0 to (floor(FFTLength / 2) − 1) where you specify the left, N_{leftG}, and right, N_{rightG}, guard bands independently in a 2-by-1 column vector. The default values are [6; 5].

InsertDCNull

This is a logical variable that controls whether a DC null is inserted. The default value is false.

The DC subcarrier is the center of the frequency band and has the index value:

- (FFTLength / 2) + 1 when FFTLength is even
- (FFTLength + 1) / 2 when FFTLength is odd

PilotInputPort

This is a logical property that controls whether you can specify the pilot carrier indices. If true, you can assign individual subcarriers for pilot transmission; otherwise, pilot information will be assumed to be embedded in the input data. The default value is false.

PilotCarrierIndices

If the comm.OFDMModulator.PilotInputPort property is set to true, you can specify the indices of the pilot subcarriers. You can assign the indices to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the property vary. Valid pilot indices fall in the range

$$\left[\mathrm{N}_{\mathrm{leftG}}+1,\mathrm{N}_{\mathrm{FFT}}/2\right]\cup\left[\mathrm{N}_{\mathrm{FFT}}/2+2,\mathrm{N}_{\mathrm{FFT}}-\mathrm{N}_{\mathrm{rightG}}\right],$$

where the index value cannot exceed the number of subcarriers. When the pilot indices are the same for every symbol and transmit antenna, the property has dimensions N_{pilot}-by-1, where $\mathrm{N}_{\mathrm{pilot}}$ is the number of pilot subcarriers. When the pilot indices vary across symbols, the property has dimensions of N_{pilot}-by-N_{sym}, where N_{sym} is the number of symbols. If there is only one symbol but multiple transmit antennas, the property has dimensions of N_{pilot}-by-1-by-N_{T}, where N_{T} is the number of transmit antennas. If the indices vary across the number of symbols and transmit antennas, the property has dimensions of N_{pilot}-by-N_{sym}-by-N_{T}. It is desirable that when the number of transmit antennas is greater than one, the indices per symbol should be mutually distinct across antennas to avoid interference. The default value is `[12; 26; 40; 54]`.

CyclicPrefixLength

The `CyclicPrefixLength` property specifies the length of the OFDM cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same length through all antennas. The default value is `16`.

Windowing

This is a `logical` property whose state enables or disables windowing. Windowing is the process in which the OFDM symbol is multiplied by a raised cosine window before transmission to more quickly reduce the power of out-of-band subcarriers. This serves to reduce spectral regrowth. The default value is `false`.

WindowLength

This property specifies the length of the raised cosine window when `comm.OFDMModulator.Windowing` is `true`. Use positive integers with a maximum value no greater than the minimum cyclic prefix length. For example, in a configuration having four symbols with cyclic prefix lengths of `[12 16 14 18]`, the window length cannot exceed `12`. The default value is `1`.

NumSymbols

This property specifies the number of symbols, N_{sym}. `NumSymbols` must be a positive integer. The default value is `1`.

NumTransmitAntennnas

This property determines the number of antennas, N_T, used to transmit the OFDM modulated signal. The property is a positive integer. The default value is `1`.

Methods

info	Provide dimensioning information for the OFDM method
reset	Reset states of the `OFDMModulator` System object
showResourceMapping	Show the subcarrier mapping of the OFDM symbols created by the OFDM modulator System object.
step	Modulate using OFDM method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Construct and Modify OFDM Modulator

An OFDM modulator System object™ can be constructed using default properties. Once constructed, these properties can be modified.

Construct an OFDM modulator.

```
ofdmMod = comm.OFDMModulator;
```

Display the properties of the modulator.

```
disp(ofdmMod)
```

```
  comm.OFDMModulator with properties:

              FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
```

```
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 1
     NumTransmitAntennas: 1
```

Modify the number of subcarriers and symbols.

```
ofdmMod.FFTLength = 128;
ofdmMod.NumSymbols = 2;
```

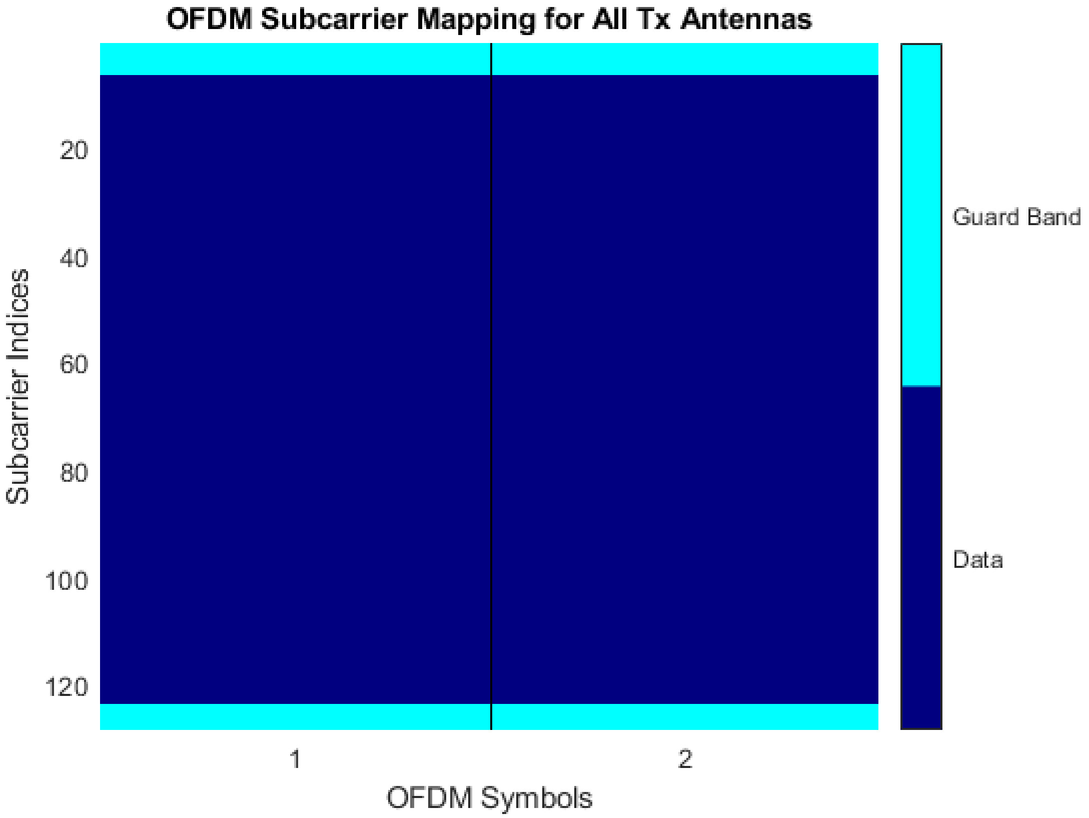
Verify that the number of subcarriers and the number of symbols changed.

```
disp(ofdmMod)
```

```
  comm.OFDMModulator with properties:

               FFTLength: 128
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 2
     NumTransmitAntennas: 1
```

The `showResourceMapping` method shows the mapping of data, pilot, and null subcarriers in the time-frequency space. Apply the `showResourceMapping` method.

```
showResourceMapping(ofdmMod)
```



Construct OFDM Modulator from OFDM Demodulator

An OFDM modulator System object™ can be constructed from an existing OFDM demodulator System object.

Construct an OFDM demodulator, `ofdmDemod` and specify pilot indices for a single symbol and two transmit antennas.

Note: You can set the `PilotCarrierIndices` property in the demodulator object, which then changes the number of transmit antennas in the modulator object. The number of receive antennas in the demodulator is uncorrelated with the number of transmit antennas.

```
ofdmDemod = comm.OFDMDemodulator;
ofdmDemod.PilotOutputPort = true;
ofdmDemod.PilotCarrierIndices = cat(3,[12; 26; 40; 54],...
    [13; 27; 41; 55]);
```

Use the demodulator, `ofdmDemod`, to construct the OFDM modulator.

```
ofdmMod = comm.OFDMModulator(ofdmDemod);
```

Display the properties of the modulator and verify that they match those of the demodulator.

```
disp(ofdmMod)
```

```
  comm.OFDMModulator with properties:

              FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
           InsertDCNull: false
         PilotInputPort: true
     PilotCarrierIndices: [4x1x2 double]
      CyclicPrefixLength: 16
              Windowing: false
             NumSymbols: 1
     NumTransmitAntennas: 2
```

```
disp(ofdmDemod)
```

```
  comm.OFDMDemodulator with properties:

              FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
        RemoveDCCarrier: false
        PilotOutputPort: true
     PilotCarrierIndices: [4x1x2 double]
      CyclicPrefixLength: 16
             NumSymbols: 1
      NumReceiveAntennas: 1
```

Visualize Time-Frequency Resource Assignments for OFDM Modulator

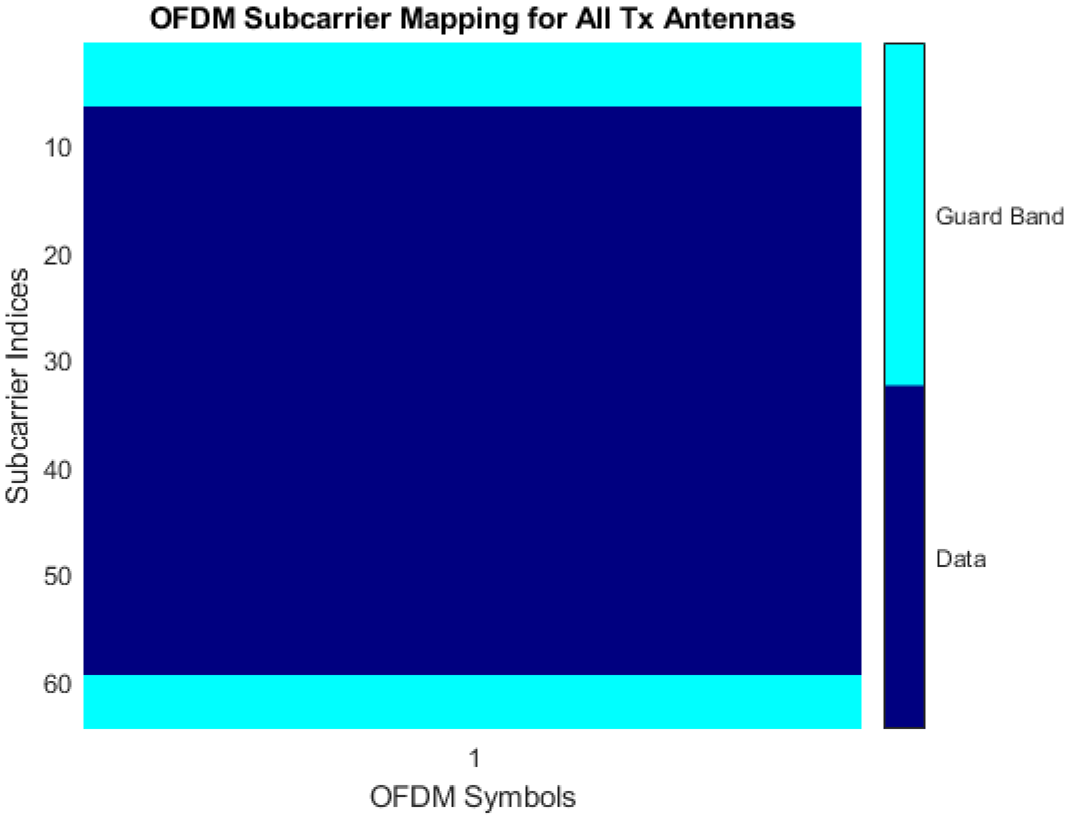
The `showResourceMapping` method displays the time-frequency resource mapping for each transmit antenna.

Construct an OFDM modulator.

```
mod = comm.OFDMModulator;
```

Apply the `showResourceMapping` method.

```
showResourceMapping(mod)
```

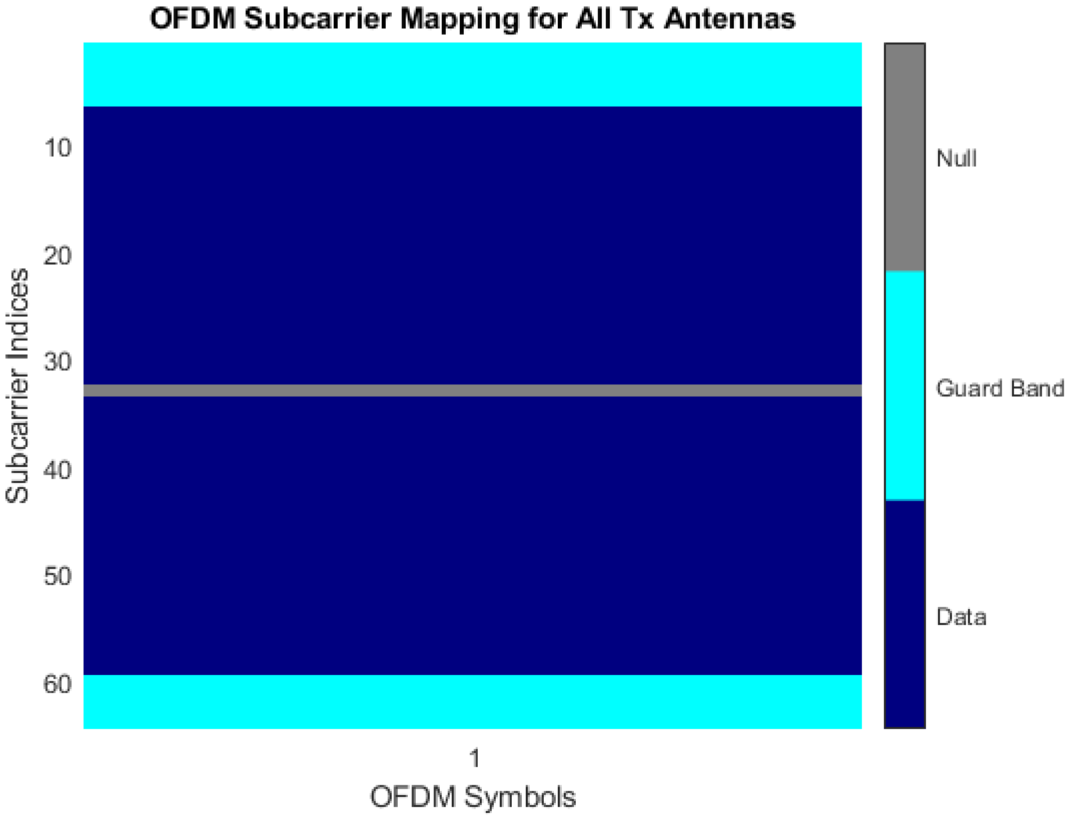


Insert a DC null.

```
mod.InsertDCNull = true;
```

Show the resource mapping after adding the DC null.

```
showResourceMapping(mod)
```



Create Modulator and Specify Pilots

The OFDM modulator enables you to specify the subcarrier indices for the pilot signals. The indices can be specified for each symbol and transmit antenna. When there is more than one transmit antenna, ensure that the pilot indices for each symbol differ between antennas.

Construct an OFDM modulator that has two symbols and insert a DC null.

```
mod = comm.OFDMModulator('FFTLength',128,'NumSymbols',2,...
    'InsertDCNull',true);
```

Turn on the pilot input port so you can specify the pilot indices.

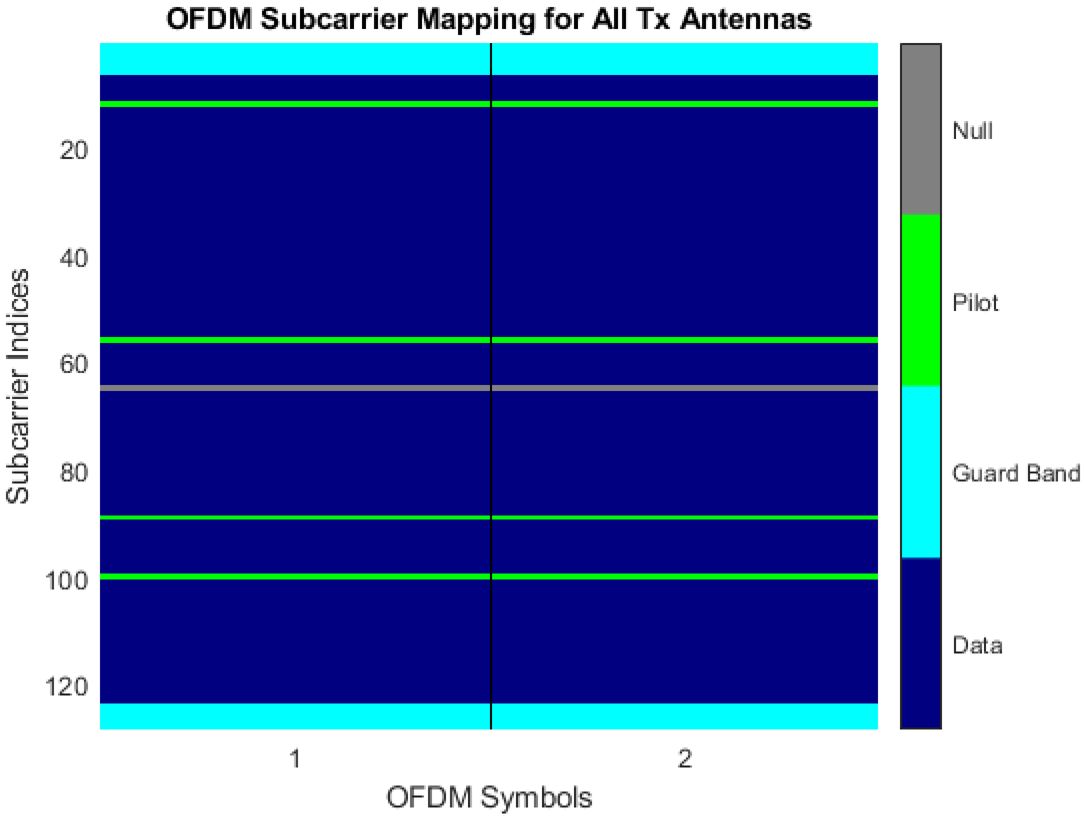
```
mod.PilotInputPort = true;
```

Specify the same pilot indices for both symbols.

```
mod.PilotCarrierIndices = [12; 56; 89; 100];
```

Visualize the placement of the pilot signals and nulls in the OFDM time-frequency grid using the `showResourceMapping` method.

```
showResourceMapping(mod)
```

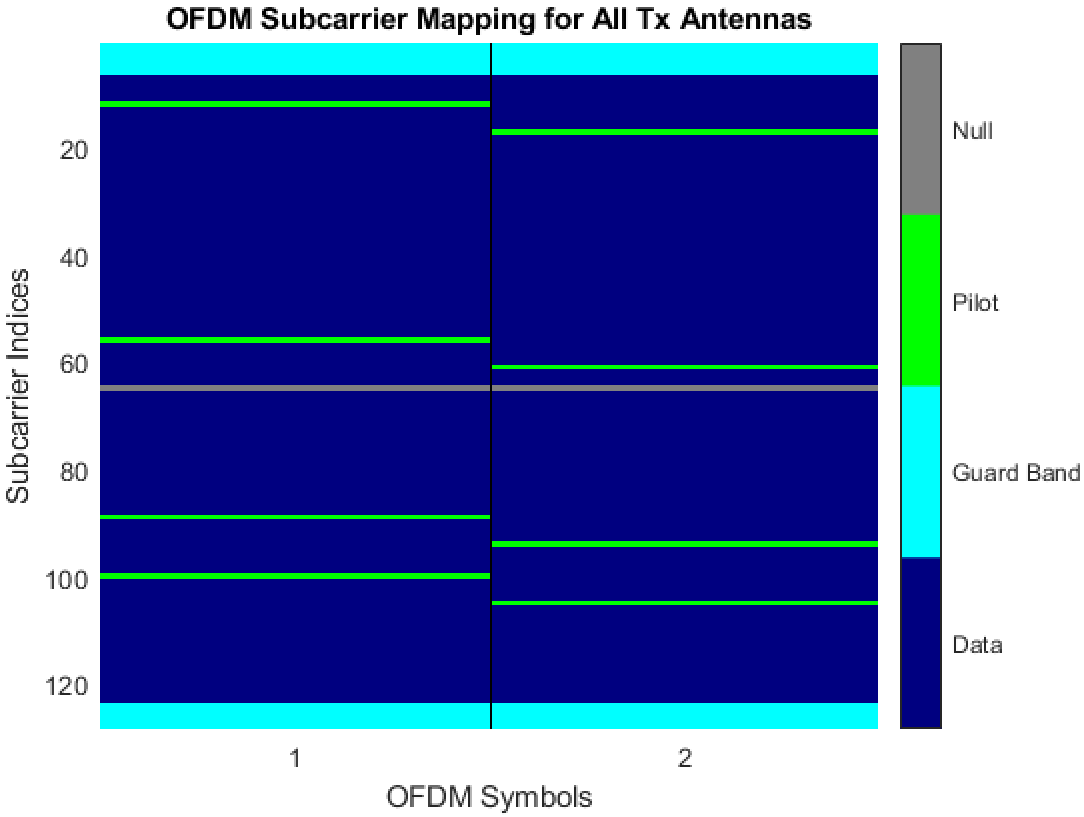


Concatenate a second column of pilot indices to the `PilotCarrierIndices` property to specify different indices for the second symbol.

```
mod.PilotCarrierIndices = cat(2, mod.PilotCarrierIndices, ...
    [17; 61; 94; 105]);
```

Verify that the pilot subcarrier indices differ between symbols.

```
showResourceMapping(mod)
```



Increase the number of transmit antennas to two.

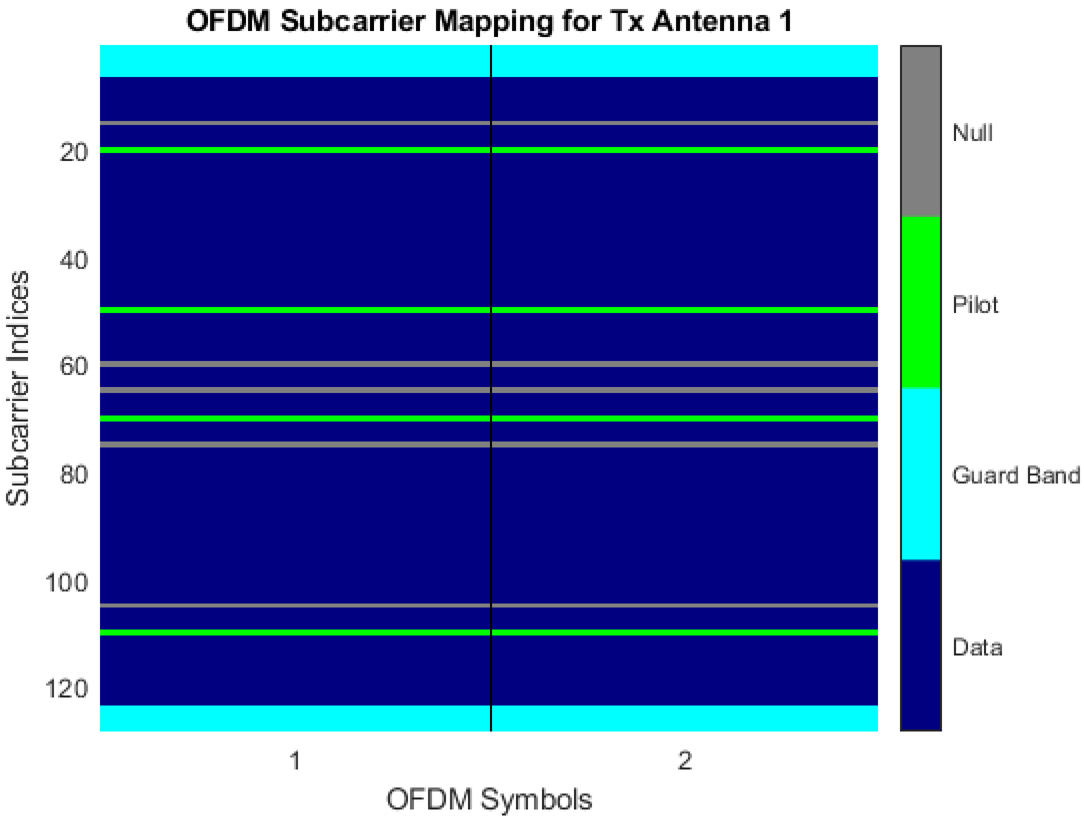
```
mod.NumTransmitAntennas = 2;
```

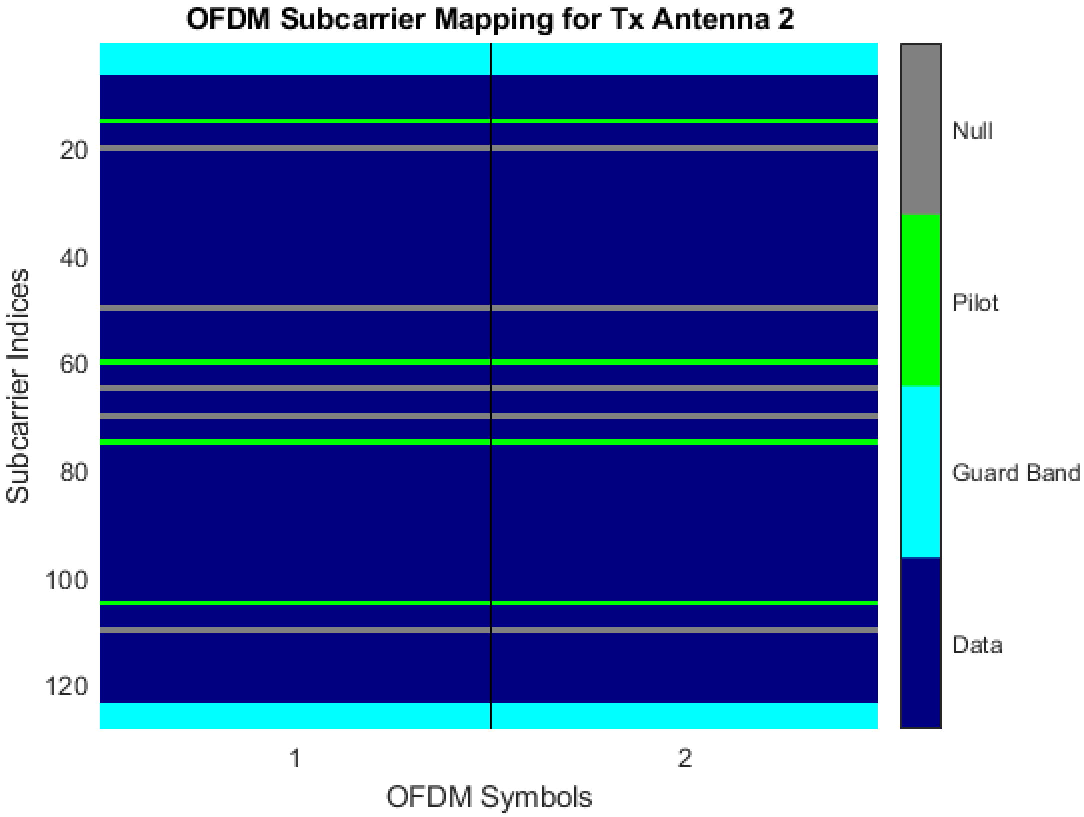
Specify the pilot indices for each of the two transmit antennas. To provide indices for multiple antennas while minimizing interference among the antennas, populate the `PilotCarrierIndices` property as a 3-D array such that the indices for each symbol differ among antennas.

```
mod.PilotCarrierIndices = cat(3,[20; 50; 70; 110], ...
    [15; 60; 75; 105]);
```

Display the resource mapping for the two transmit antennas. The gray lines denote the insertion of custom nulls. The nulls are created by the object to minimize interference among the pilot symbols from different antennas.

```
showResourceMapping(mod)
```





Create Modulator with Varying Cyclic Prefix Lengths

Specify the length of the cyclic prefix for each OFDM symbol.

Construct an OFDM modulator having five symbols, four left guard-band subcarriers, and three right guard-band subcarriers. Specify the cyclic prefix length for each OFDM symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
    'NumSymbols',5,...
    'CyclicPrefixLength',[12 10 14 11 13]);
```

Display the properties of the modulator and verify that the cyclic prefix length changes across symbols.

```
disp(mod)
```

```
  comm.OFDMModulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
      CyclicPrefixLength: [12 10 14 11 13]
               Windowing: false
              NumSymbols: 5
     NumTransmitAntennas: 1
```

Info Method to Determine OFDM Modulator Data Dimensions

Determine the OFDM modulator data dimensions by using the `info` method.

Construct an OFDM modulator System object™ with user-specified pilot indices, insert a DC null, and specify two transmit antennas.

```
hMod = comm.OFDMModulator('NumGuardBandCarriers',[4;3], ...
    'PilotInputPort',true, ...
    'PilotCarrierIndices',cat(3,[12; 26; 40; 54], ...
    [11; 25; 39; 53]), ...
    'InsertDCNull',true, ...
    'NumTransmitAntennas',2);
```

Use the `info` method to find the modulator input data, pilot input data, and output data sizes.

```
info(hMod)
```

```
ans = struct with fields:
     DataInputSize: [48 1 2]
    PilotInputSize: [4 1 2]
        OutputSize: [80 2]
```

Create OFDM Modulated Data

Generate OFDM modulated symbols for use in link-level simulations.

Construct an OFDM modulator with an inserted DC null, seven guard-band subcarriers, and two symbols having different pilot indices for each symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
'PilotInputPort',true, ...
'PilotCarrierIndices',[12 11; 26 27; 40 39; 54 55], ...
'NumSymbols',2, ...
'InsertDCNull',true);
```

Determine input data, pilot, and output data dimensions.

```
modDim = info(mod);
```

Generate random data symbols for the OFDM modulator. The structure variable, `modDim`, determines the number of data symbols.

```
dataIn = complex(randn(modDim.DataInputSize),randn(modDim.DataInputSize));
```

Create a pilot signal that has the correct dimensions.

```
pilotIn = complex(rand(modDim.PilotInputSize),rand(modDim.PilotInputSize));
```

Apply OFDM modulation to the data and pilot signals.

```
modData = step(mod,dataIn,pilotIn);
```

Use the OFDM modulator object to create the corresponding OFDM demodulator.

```
demod = comm.OFDMDemodulator(mod);
```

Demodulate the OFDM signal and output the data and pilot signals.

```
[dataOut, pilotOut] = step(demod,modData);
```

Verify that, within a tight tolerance, the input data and pilot symbols match the output data and pilot symbols.

```
isSame = (max(abs([dataIn(:) - dataOut(:); ...
    pilotIn(:) - pilotOut(:)])) < 1e-10)
```

```
isSame = logical
   1
```

Algorithms

Orthogonal frequency division modulation (OFDM) divides a high-rate transmit data stream into N lower-rate streams, each of which has a symbol duration larger than the channel delay spread. This serves to mitigate intersymbol interference (ISI). The individual substreams are sent over N parallel subchannels which are orthogonal to each other. Through the use of an inverse fast Fourier transform (IFFT), OFDM can be transmitted using a single radio. Specifically, the OFDM Modulator System object modulates an input signal using orthogonal frequency division modulation. The output is a baseband representation of the modulated signal:

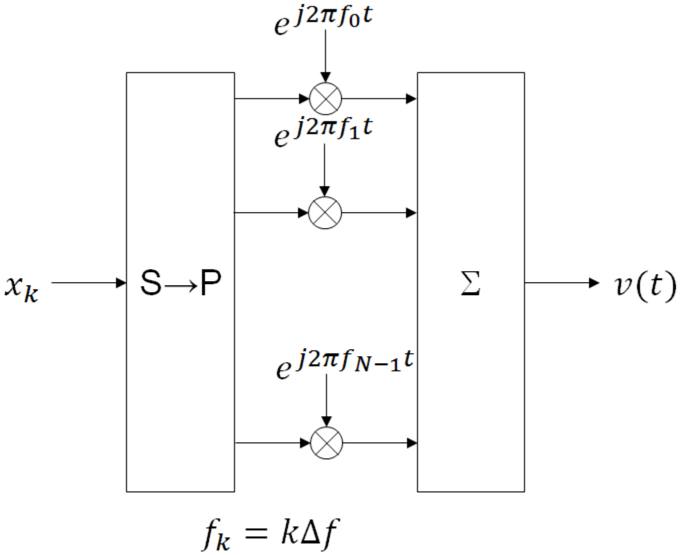
$$v(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi k\Delta ft}, \quad 0 \le t \le T,$$

where $\{X_k\}$ are data symbols, N is the number of subcarriers, and T is the OFDM symbol time. The subcarrier spacing of $\Delta f = 1/T$ makes them orthogonal over each symbol period. This is expressed as:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi m\Delta ft}\right)^* \left(e^{j2\pi n\Delta ft}\right) dt = \frac{1}{T}\int_0^T e^{j2\pi(m-n)\Delta ft}\ dt = 0 \quad \text{for } m \ne n.$$

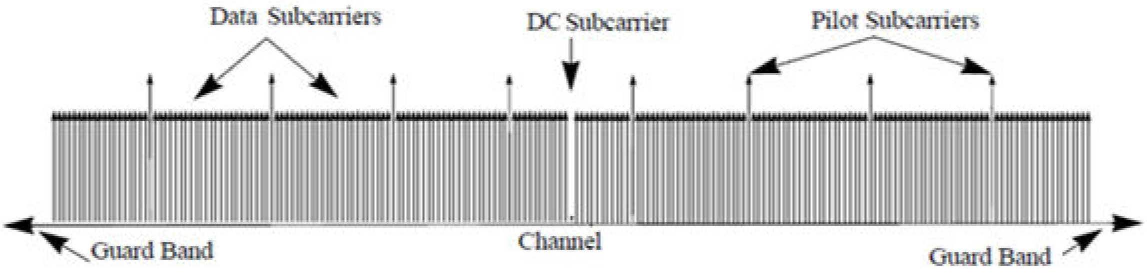
The data symbols, X_k, are usually complex and can be from any modulation alphabet, e.g., QPSK, 16-QAM, or 64-QAM.

The figure shows an OFDM modulator. It consists of a bank of N complex modulators, where each corresponds to one OFDM subcarrier.

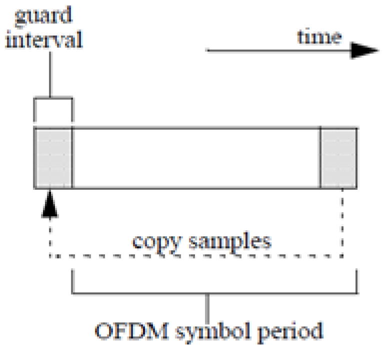


Guard Bands and Intervals

There are three types of OFDM subcarriers: data, pilot, and null. Data subcarriers are used for transmitting data while pilot subcarriers are used for channel estimation. There is no transmission on null subcarriers, which provide a DC null and provide buffers between OFDM resource blocks. These buffers are referred to as guard bands whose purpose is to prevent inter-symbol interference. The allocation of nulls and guard bands vary depending upon the applicable standard, e.g., 802.11n differs from LTE. Consequently, the OFDM modulator object allows the user to assign subcarrier indices.



Analogous to the concept of guard bands, the OFDM modulator object supports guard intervals which are used to provide temporal separation between OFDM symbols so that the signal does not lose orthogonality due to time-dispersive channels. As long as the guard interval is longer than the delay spread, each symbol does not interfere with other symbols. Guard intervals are created by using cyclic prefixes in which the last part of an OFDM symbol is copied and inserted as the first part of the OFDM symbol. The benefit of cyclic prefix insertion is maintained as long as the span of the time dispersion does not exceed the duration of the cyclic prefix. The OFDM modulator object enables the setting of the cyclic prefix length. The drawback in using a cyclic prefix is the penalty from increased overhead.



Raised Cosine Windowing

While the cyclic prefix creates guard period in time domain to preserve orthogonality, an OFDM symbol rarely begins with the same amplitude and phase exhibited at the end of the prior OFDM symbol. This causes spectral regrowth, which is the spreading of signal bandwidth due to intermodulation distortion. To limit this spectral regrowth, it is desired to create a smooth transition between the last sample of a symbol and the first sample of the next symbol. This can be done by using a cyclic suffix and raised cosine windowing.

To create the cyclic suffix, the first N_{WIN} samples of a given symbol are appended to the end of that symbol. However, in order to comply with the 802.11g standard, for example, the length of a symbol cannot be arbitrarily lengthened. Instead, the cyclic suffix must overlap in time and is effectively summed with the cyclic prefix of the following symbol. This overlapped segment is where windowing is applied. Two windows are applied, one of which is the mathematical inverse of the other. The first raised cosine window is applied to the cyclic suffix of symbol *k*, and decreases from 1 to 0 over its duration. The second raised cosine window is applied to the cyclic prefix of symbol *k*+1, and increases from 0 to 1 over its duration. This provides a smooth transition from one symbol to the next.

The raised cosine window, *w(t)*, in the time domain can be expressed as:

$$w(t) = \begin{cases} 1, & 0 \le |t| < \dfrac{T - T_W}{2} \\ \dfrac{1}{2}\left\{1 + \cos\left[\dfrac{\pi}{T_W}\left(|t| - \dfrac{T - T_W}{2}\right)\right]\right\}, & \dfrac{T - T_W}{2} \le |t| \le \dfrac{T + T_W}{2} \\ 0, & \text{otherwise} \end{cases}$$

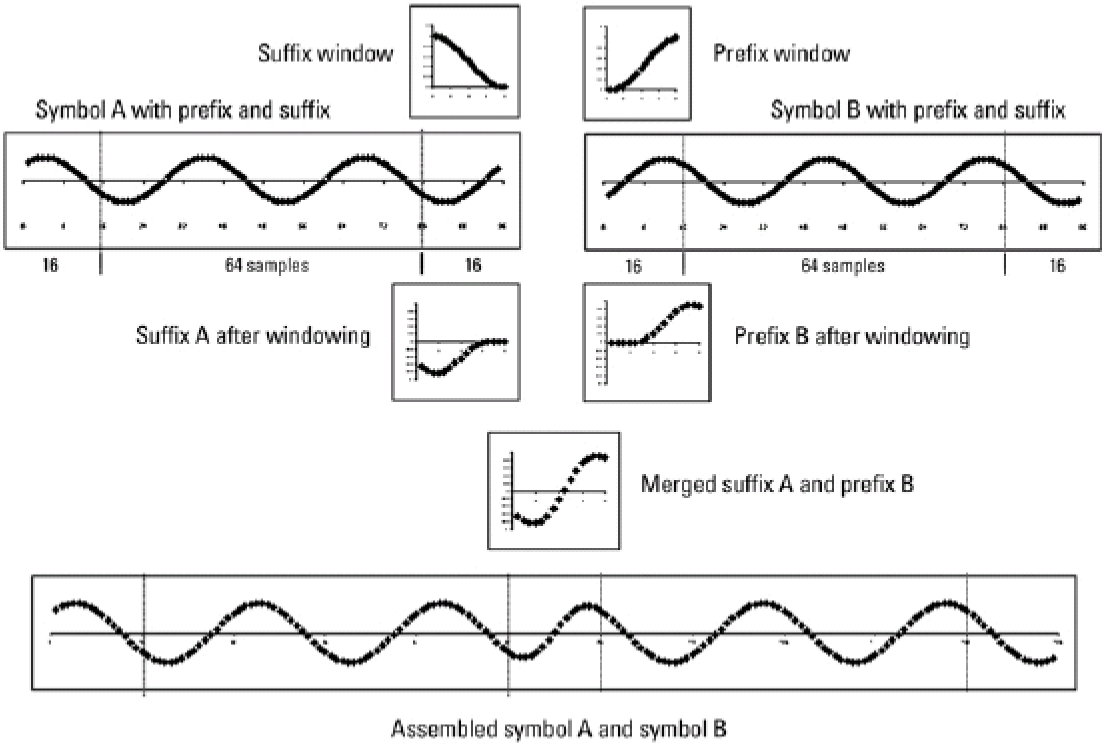
,

where

- T represents the OFDM symbol duration including the guard interval.
- T_{W} represents the duration of the window.

Adjust the length of the cyclic suffix via the window length setting property, with suffix lengths set between 1 and the minimum cyclic prefix length. While windowing improves spectral regrowth, it does so at the expense of multipath fading immunity. This occurs because redundancy in the guard band is reduced because the guard band sample values are compromised by the smoothing.

The following figures display the application of raised cosine windowing.



Selected Bibliography

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX*.Upper Saddle River, NJ: Prentice Hall, 2007.

[3] Agilent Technologies, Inc., “OFDM Raised Cosine Windowing”, http://wireless.agilent.com/rfcomms/n4010a/n4010aWLAN/onlineguide/ofdm_raised_cosine_windowing.htm.

[4] Montreuil, L., R. Prodan, and T. Kolze. “OFDM TX Symbol Shaping 802.3bn”, http://www.ieee802.org/3/bn/public/jan13/montreuil_01a_0113.pdf.Broadcom, 2013.

[5] “IEEE Standard 802.16™-2009,” New York: IEEE, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

OFDM Modulator Baseband | `comm.OFDMDemodulator` | `comm.QPSKModulator` | `comm.RectangularQAMModulator`

Introduced in R2014a

info

System object: comm.OFDMModulator
Package: comm

Provide dimensioning information for the OFDM method

Syntax

```
Y = info(H)
```

Description

`Y = info(H)` provides data dimensioning information for the OFDM modulator System object, `H`. It returns the expected dimensions for the:

- Input data array
- Pilot data array
- Output data array

The output, `Y`, is a structure containing the following three fields.

Y.DataInputSize

Dimensions of the modulator input data, N_{data}-by-N_{sym}-by-N_t, where N_{data} is the number of data subcarriers such that $N_{data} = N_{FFT} - N_{leftG} - N_{rightG} - N_{DCNull} - N_{pilot} - N_{custNull}$.

Variable Definitions

Variable	Description
N_{FFT}	Number of subcarriers
N_{leftG}	Number of subcarriers in the left guard band
N_{rightG}	Number of subcarriers in the right guard band
N_{DCNull}	Number of subcarriers in the DC null (either 0 or 1)
N_{pilot}	Number of pilot subcarriers
$N_{custNull}$	Number of subcarriers used for custom nulls (applies only when the pilot indices property is a 3-D array)
N_t	Number of transmit antennas

Y.PilotInputSize

Dimensions of the pilot input array, N_{pilot}-by-N_{sym}-by-N_t.

Y.OutputSize

Dimensions of the modulator output data, $(N_{FFT} + N_{CP})\times N_{sym}$-by-$N_t$, where N_{CP} is the length of the cyclic prefix.

reset

System object: comm.OFDMModulator
Package: comm

Reset states of the `OFDMModulator` System object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `OFDMModulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

showResourceMapping

System object: comm.OFDMModulator
Package: comm

Show the subcarrier mapping of the OFDM symbols created by the OFDM modulator System object.

Syntax

```
showResourceMapping(H)
showResourceMapping(H,CI)
```

Description

`showResourceMapping(H)` shows a visualization of the subcarrier mapping for the OFDM symbols used by the OFDM modulator System object, `H`. The subcarrier indices are numbered from 1 to N_{FFT}.

`showResourceMapping(H,CI)` shows the resource mapping where the optional argument, `CI`, is used to number the subcarrier indices that will be displayed. `CI` is a 1x2 integer row vector such that `diff(CI)=` N_{FFT} `− 1`.

step

System object: comm.OFDMModulator
Package: comm

Modulate using OFDM method

Syntax

```
Y = step(H,X)
Y = step(H,X,PILOT)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the OFDM modulator System object, `H`, and returns the baseband modulated output, `Y`, which is a double-precision, 2-D array with complex values. The input, `X`, is a numeric, real or complex 3-D array of symbols (typically created with a baseband demodulator, e.g., QPSK). Its dimensions are a function of the number of subcarriers, the number of guard band subcarriers, the number of pilot subcarriers, and whether or not there is a DC null. You can determine the dimensions by using the `info` method.

`Y = step(H,X,PILOT)` maps the `PILOT` signal onto the subcarriers specified by the PilotCarrierIndices property of `H`. The input `PILOT` is a numeric, real or complex 3-D array. This syntax applies when the PilotInputPort property of `H` is true. The `info` method provides the dimensions of the `PILOT` array.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.OFDMDemodulator System object

Package: comm

Demodulate using OFDM method

Description

The `OFDMDemodulator` object demodulates using the orthogonal frequency division demodulation method. The output is a baseband representation of the modulated signal, which was input into the `OFDMModulator` companion object.

To demodulate an OFDM signal:

1. Define and set up the OFDM demodulator object. See "Construction" on page 3-221.
2. Call `step` to demodulate a signal according to the properties of `comm.OFDMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.OFDMDemodulator` creates a demodulator System object, `H`, that demodulates an input signal by using the orthogonal frequency division demodulation method.

`H = comm.OFDMDemodulator(Name,Value)` creates an OFDM 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.OFDMDemodulator(hMod)` creates an OFDM demodulator object, `H`, whose properties are determined by the corresponding OFDM modulator object, `hMod`.

Properties

FFTLength

The length of the FFT, N_{FFT}, is equivalent to the number of subcarriers used in the modulation process. FFTLength must be ≥ 8.

Specify the number of subcarriers. The default is 64.

NumGuardBandCarriers

The number of guard band subcarriers allocated to the left and right guard bands.

Specify the number of left and right subcarriers as nonnegative integers in $[0,N_{FFT}/2 - 1]$ where you specify the left, N_{leftG}, and right, N_{rightG}, guard bands independently in a 2-by-1 column vector. The default values are [6; 5].

RemoveDCCarrier

A logical variable that when true, mandates removal of a DC subcarrier. The default value is false.

PilotOutputPort

A logical property that controls whether to separate the pilot signals and make them available at an additional output port. The location of each pilot output symbol is determined by the pilot subcarrier indices specified in the PilotCarrierIndices property. When false, pilot symbols may be present but embedded in the data. The default value is false.

PilotCarrierIndices

If the PilotOutputPort property is true, output separate pilot signals located at the indices specified by the PilotCarrierIndices property. If the indices are a 2-D array, the pilot carriers across all the transmit antennas per symbol are the same. If there is more than one transmit antenna (this information is not known by the demodulator), the pilots from different transmit antennas may interfere with each other. To avoid this, specify the pilot carrier indices as a 3-D array with different pilot indices for each symbol across the antennas. This avoids interference between pilots from different transmit antennas, since, on a per-symbol basis, each transmit antenna has different pilot carriers and the OFDM modulator creates custom nulls at the appropriate locations. The size of the third dimension of the PilotCarrierIndices property gives the number of transmit antennas.

CyclicPrefixLength

The cyclic prefix length property specifies the length of the OFDM cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same length through all antennas. The default value is 16.

NumSymbols

This property specifies the number of symbols, N_{sym}. Specify N_{sym} as a positive integer. The default value is 1.

NumReceiveAntennnas

This property determines the number of antennas, N_R, used to receive the OFDM modulated signal. Specify N_R as a positive integer. The default value is 1.

Methods

info	Provide dimensioning information for the OFDM method
reset	Reset states of the OFDMDemodulator System object
showResourceMapping	Show the subcarrier mapping of the OFDM symbols created by the OFDM demodulator System object
step	Demodulate using OFDM method

Common to All System Objects	
release	Allow System object property value changes

Examples

Create and Modify OFDM Demodulator

Construct an OFDM demodulator System object™ with default properties. Modify some of the properties.

Construct the OFDM demodulator.

```
demod = comm.OFDMDemodulator
```

```
demod = 
  comm.OFDMDemodulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: false
      CyclicPrefixLength: 16
              NumSymbols: 1
      NumReceiveAntennas: 1
```

Modify the number of subcarriers and symbols.

```
demod.FFTLength = 128;
demod.NumSymbols = 2;
```

Verify that the number of subcarriers and the number of symbols changed.

```
demod
```

```
demod = 
  comm.OFDMDemodulator with properties:

               FFTLength: 128
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: false
      CyclicPrefixLength: 16
              NumSymbols: 2
      NumReceiveAntennas: 1
```

Create OFDM Demodulator from OFDM Modulator

Create an OFDM demodulator System object™ from an existing OFDM modulator System object.

Construct an OFDM modulator using default parameters.

```
mod = comm.OFDMModulator('NumTransmitAntennas',4);
```

Construct the corresponding OFDM demodulator from the modulator, `mod`.

```
demod = comm.OFDMDemodulator(mod);
```

Display the properties of the modulator and verify that they match those of the demodulator.

```
mod
```

```
mod = 
  comm.OFDMModulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 1
     NumTransmitAntennas: 4
```

```
demod
```

```
demod = 
  comm.OFDMDemodulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: false
      CyclicPrefixLength: 16
              NumSymbols: 1
      NumReceiveAntennas: 1
```

Note that the number of transmit antennas is independent of the number of receive antennas.

Visualize Time-Frequency Resource Assignments

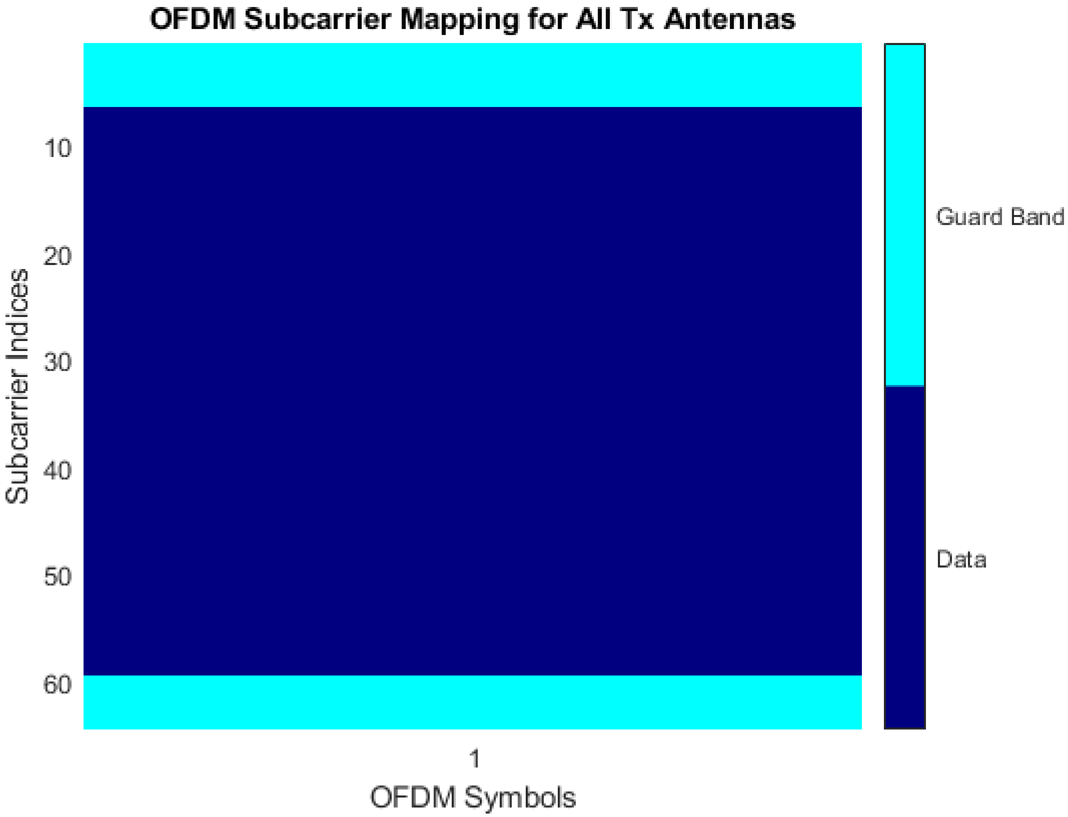
The `showResourceMapping` method shows the time-frequency resource mapping for each transmit antenna.

Construct an OFDM demodulator.

```
demod = comm.OFDMDemodulator;
```

Apply the `showResourceMapping` method.

```
showResourceMapping(demod)
```

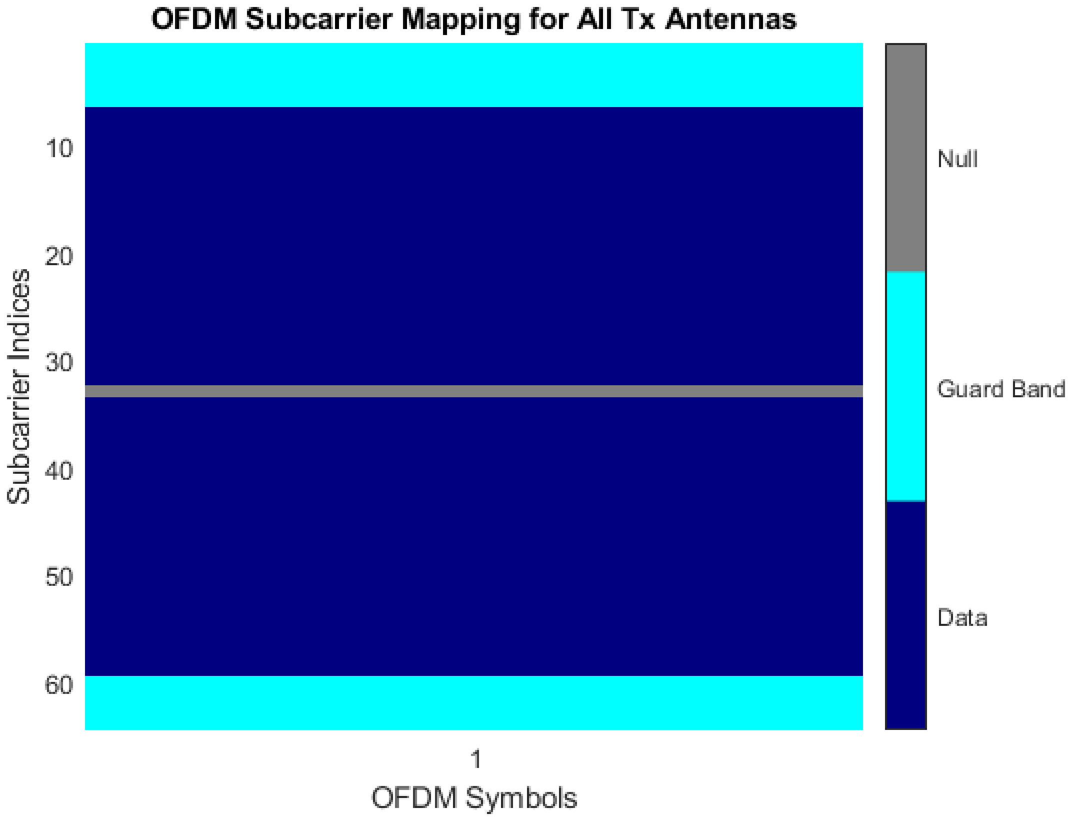


Remove the DC subcarrier.

```
demod.RemoveDCCarrier = true;
```

Show the resource mapping after removing the DC subcarrier.

```
showResourceMapping(demod)
```



Demodulate OFDM Data

Construct an OFDM modulator with an inserted DC null, seven guard-band subcarriers, and two symbols that have different pilot indices for each symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
'PilotInputPort',true,'PilotCarrierIndices',cat(2,[12; 26; 40; 54],...
[11; 27; 39; 55]),'NumSymbols',2,'InsertDCNull',true);
```

Determine input data, pilot, and output data dimensions.

```
modDim = info(mod)
modDim = struct with fields:
     DataInputSize: [52 2]
    PilotInputSize: [4 2]
        OutputSize: [160 1]
```

Generate random data symbols for the OFDM modulator. Determine the number of data symbols by using the structure variable, `modDim`.

```
dataIn = complex(randn(modDim.DataInputSize),randn(modDim.DataInputSize));
```

Create a pilot signal that has the correct dimensions.

```
pilotIn = complex(rand(modDim.PilotInputSize),rand(modDim.PilotInputSize));
```

Apply OFDM modulation to the data and pilot signals.

```
modSig = step(mod,dataIn,pilotIn);
```

Use the OFDM modulator object to create the corresponding OFDM demodulator.

```
demod = comm.OFDMDemodulator(mod);
```

Demodulate the OFDM signal and output the data and pilot signals.

```
[dataOut,pilotOut] = step(demod,modSig);
```

Verify that the input data and pilot symbols match the output data and pilot symbols.

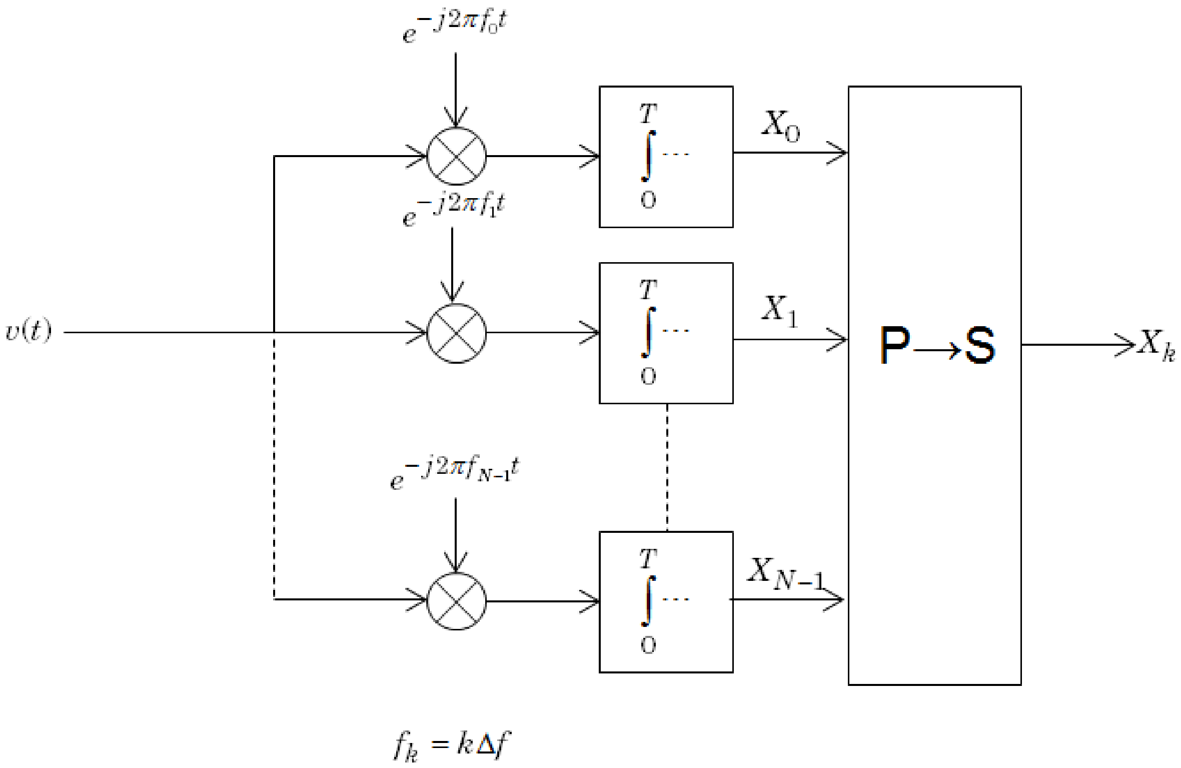
```
isSame = (max(abs([dataIn(:) - dataOut(:); ...
    pilotIn(:) - pilotOut(:)])) < 1e-10)
```

```
isSame = logical
   1
```

Algorithms

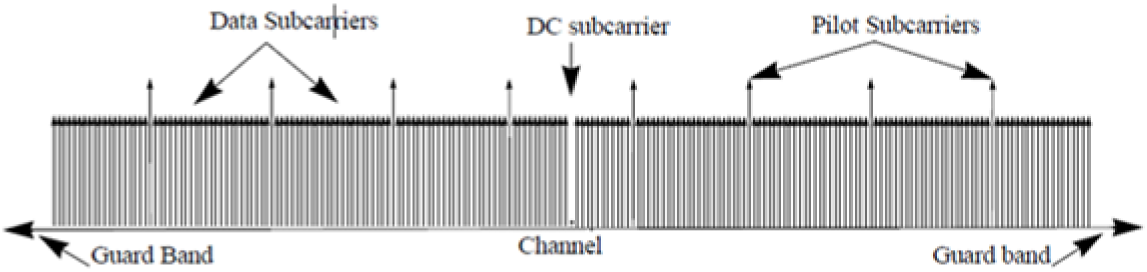
The Orthogonal Frequency Division Modulation (OFDM) Demodulator System object demodulates an OFDM input signal by using an FFT operation that results in *N* parallel data streams.

The figure shows an OFDM demodulator. It consists of a bank of N correlators with one assigned to each OFDM subcarrier followed by a parallel-to-serial conversion.

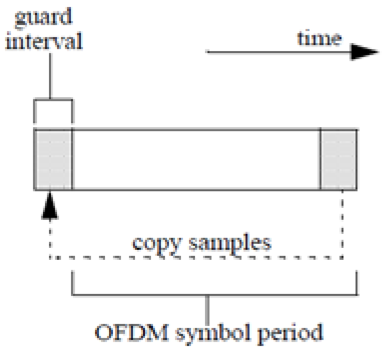


Guard Bands and Intervals

There are three types of OFDM subcarriers: data, pilot, and null. Data subcarriers are used for transmitting data while pilot subcarriers are used for channel estimation. There is no transmission on null subcarriers, which are used to provide a DC null as well as to provide buffers between OFDM resource blocks. These buffers are referred to as guard bands whose purpose is to prevent inter-symbol interference. The allocation of nulls and guard bands varies depending upon the standard, e.g., 802.11n differs from LTE. Consequently, the OFDM modulator object allows the user to assign subcarrier indices as required.



Analogous to the concept of guard bands, the OFDM modulator object supports guard intervals that provide temporal separation between OFDM symbols so that the signal does not lose orthogonality due to time-dispersive channels. As long as the guard interval is longer than the delay spread, each symbol does not interfere with other symbols. Guard intervals are created by using cyclic prefixes in which the last part of an OFDM symbol is copied and inserted as the first part of the OFDM symbol. The benefit of cyclic prefix insertion is maintained as long as the span of the time dispersion does not exceed the duration of the cyclic prefix. The OFDM modulator object enables the cyclic prefix length to be set. The drawback in using a cyclic prefix is increased overhead.



Selected Bibliography

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed, *Fundamentals of WiMAX*, Upper Saddle River, NJ: Prentice Hall, 2007.

[3] I. E. E. E., “IEEE Standard 802.16™-2009.”

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

OFDM Demodulator Baseband | `comm.OFDMModulator` | `comm.QPSKDemodulator` | `comm.RectangularQAMDemodulator` | `ofdmdemod`

Introduced in R2014a

info

System object: comm.OFDMDemodulator
Package: comm

Provide dimensioning information for the OFDM method

Syntax

```
Y = info(H)
```

Description

`Y = info(H)` provides data dimensioning information for the OFDM demodulator System object, `H`. It returns the expected dimensions for data input into the OFDM demodulator, for the pilot output, and for the data output from the demodulator. The output, `Y`, is a structure containing three fields: `InputSize`, `DataOutputSize`, and `PilotOutputSize`.

Y.InputSize

Gives the dimensions of the demodulator input data, $[(N_{FFT} + N_{CP}) \times N_{sym}]$-by-$N_r$, where N_{FFT} is the number of subcarriers, N_{CP} is the length of the cyclic prefix, N_{sym} is the number of symbols, and N_r is the number of receive antennas.

Y.DataOutputSize

Shows the dimensions of the demodulator output data, N_{data}-by-N_{sym}-by-N_r, where N_{data} is the number of data subcarriers such that $N_{data} = N_{FFT} - N_{leftG} - N_{rightG} - N_{DCNull} - N_{pilot} - N_{custNull}$. The variables are defined as follows:

N_{FFT}	Number of subcarriers
N_{leftG}	Number of subcarriers in the left guard band
N_{rightG}	Number of subcarriers in the right guard band

N_{DCNull}	Number of subcarriers in the DC null (either 0 or 1)
N_{pilot}	Number of pilot subcarriers
$N_{custNull}$	Number of subcarriers used for custom nulls

Y.PilotOutputSize

Provides the dimensions of the pilot signal output array, N_{pilot}-by-N_{sym}-by-N_r or N_{pilot}-by-N_{sym}-by-N_t-by-N_r, depending on the number of transmit antennas.

reset

System object: comm.OFDMDemodulator
Package: comm

Reset states of the `OFDMDemodulator` System object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `OFDMDemodulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

showResourceMapping

System object: comm.OFDMDemodulator
Package: comm

Show the subcarrier mapping of the OFDM symbols created by the OFDM demodulator System object

Syntax

```
showResourceMapping(H)
showResourceMapping(H,CI)
```

Description

`showResourceMapping(H)` shows a visualization of the subcarrier mapping for the OFDM symbols used by the OFDM demodulator System object, `H`. The subcarrier indices are numbered from 1 to N_{FFT}.

`showResourceMapping(H,CI)` shows the resource mapping where the optional argument, `CI`, is used to number the subcarrier indices that will be displayed. `CI` is a 1x2 integer row vector such that `diff(CI)`= $N_{FFT} - 1$.

step

System object: comm.OFDMDemodulator
Package: comm

Demodulate using OFDM method

Syntax

```
Y = step(H,X)
[Y,PILOT] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the OFDM demodulator System object, `H`, and returns the baseband demodulated output, `Y`. The input is a double-precision, real or complex, 2-D matrix of symbols whose dimensions are a function of the number of subcarriers, the cyclic prefix length, and the number of receive antennas. You can determine the dimensions by using the `info` method. The output, `Y`, is a double-precision, complex, 3-D array.

`[Y,PILOT] = step(H,X)` separates the `PILOT` signal on the subcarriers specified by the PilotCarrierIndices property value of `H`. This syntax applies when the PilotOutputPort property of `H` is true. `PILOT` is a double-precision, complex, 3-D array.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CarrierSynchronizer System object

Package: comm

Compensate for carrier frequency offset

Description

The `CarrierSynchronizer` System object compensates for carrier frequency and phase offsets for single carrier modulation schemes.

To compensate for frequency and phase offsets:

1. Define and set up the `CarrierSynchronizer` object. See “Construction” on page 3-239.
2. Call `step` to compensate for the carrier frequency and phase offsets according to the properties of `comm.CarrierSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

The algorithm inherent to the carrier synchronizer is compatible with BPSK, QPSK, OQPSK, 8-PSK, QAM, and PAM modulation schemes.

Note This object does not resolve phase ambiguities created by the synchronization algorithm. See “QPSK Transmitter and Receiver” for an example of how ambiguities are addressed.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`S = comm.CarrierSynchronizer` creates a compensator System object, `S`, that compensates for the carrier frequency and phase offsets.

S = comm.CarrierSynchronizer(Name,Value) creates a compensator object 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

Modulation

Modulation type

Specify the modulation type as BPSK, QPSK, OQPSK, 8PSK, QAM, or PAM. The default value is QAM. This property is nontunable.

This object supports CPM. It has been tested for a CPM signal having 1 sample per symbol and a modulation index of 0.5.

ModulationPhaseOffset

Modulation phase offset method

Specify the method used to calculate the modulation phase offset as either Auto or Custom.

- Auto applies the traditional offset for the specified modulation type.

Modulation	Phase Offset
BPSK	0
QPSK or OQPSK	$\pi/4$
8PSK	$\pi/8$
QAM or PAM	0

- Custom enables the CustomPhaseOffset property, which you can use to specify your own phase offset.

The default value is Auto. This property is tunable.

CustomPhaseOffset

Phase offset

Specify the phase offset in radians as a real scalar. This property is available only when the ModulationPhaseOffset property is set to Custom. The default value is 0. This property is tunable.

SamplesPerSymbol

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar. The default value is 2. This property is tunable.

DampingFactor

Damping factor of the loop

Specify the damping factor of the loop as a positive real finite scalar. The default value is 0.707. This property is tunable.

NormalizedLoopBandwidth

Normalized bandwidth of the loop

Specify the normalized loop bandwidth as a real scalar between 0 and 1. The loop bandwidth is normalized by the sample rate of the synchronizer. The default value is 0.01. This property is tunable.

Methods

info	Characteristic information about carrier synchronizer
reset	Reset states of the carrier synchronizer object
step	Compensate for carrier frequency and phase offset

Common to All System Objects	
release	Allow System object property value changes

Examples

Correct Phase and Frequency Offset in QPSK Link

Correct phase and frequency offsets of a QPSK signal passed through an AWGN channel.

Create a phase and frequency offset System object™, where the frequency offset is 1% of the sample rate.

```
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',1e4,...
    'PhaseOffset',45,...
    'SampleRate',1e6);
```

Create a carrier synchronizer object.

```
carrierSync = comm.CarrierSynchronizer( ...
    'SamplesPerSymbol',1,...
    'Modulation','QPSK');
```

Generate random data symbols and apply QPSK modulation.

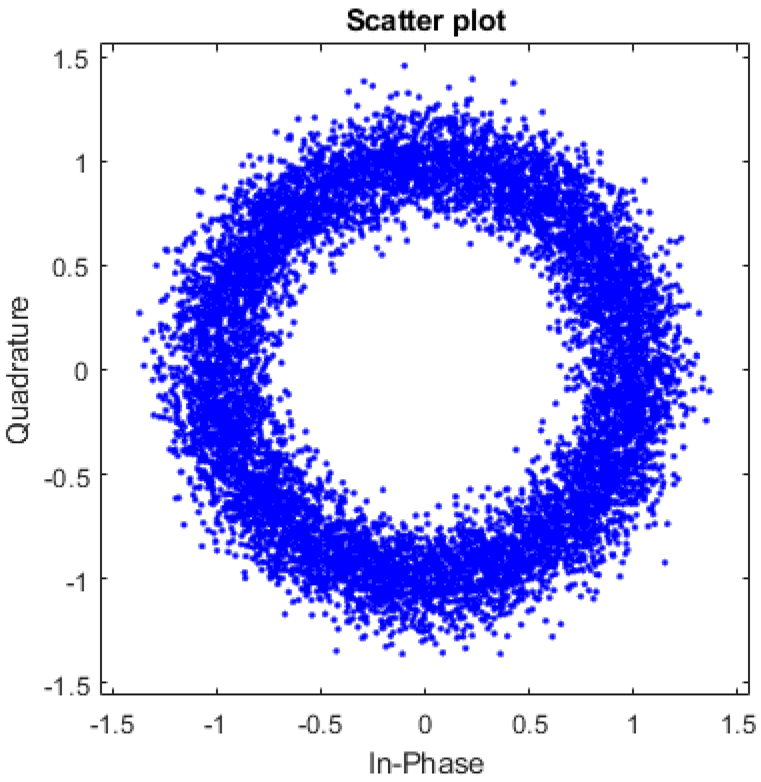
```
data = randi([0 3],10000,1);
modSig = pskmod(data,4,pi/4);
```

Apply phase and frequency offsets using the `pfo` System object. Then, pass the offset signal through an AWGN channel.

```
modSigOffset = pfo(modSig);
rxSig = awgn(modSigOffset,15);
```

Display the scatter plot of the received signal. The data appear in a circle instead of being grouped around the reference constellation points due to the frequency offset.

```
scatterplot(rxSig)
```

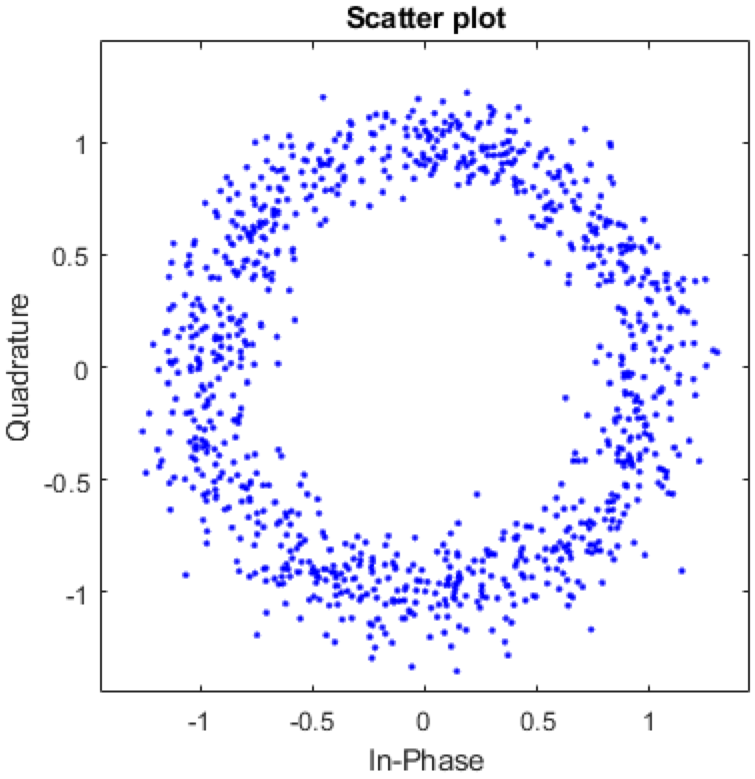


Correct for the phase and frequency offset by using the carrier synchronizer object.

```
syncSignal = carrierSync(rxSig);
```

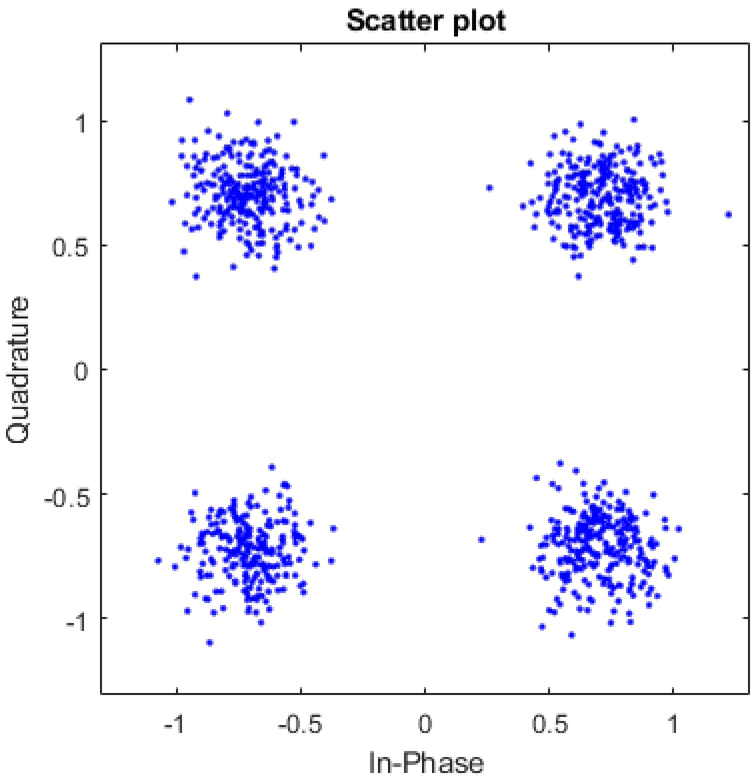
Display the first 1000 symbols of corrected signal. The synchronizer has not yet converged so the data is not grouped around the reference constellation points.

```
scatterplot(syncSignal(1:1000))
```



Display the last 1000 symbols of the corrected signal. The data is now aligned with the reference constellation because the synchronizer has converged to a solution.

```
scatterplot(syncSignal(9001:10000))
```



Estimate Frequency Offset in an 8-PSK Link

Estimate the frequency offset introduced into a noisy 8-PSK signal using the carrier synchronizer System object™.

Set the example parameters.

```
M = 8;                      % Modulation order
fs = 1e6;                   % Sample rate (Hz)
foffset = 1000;             % Frequency offset (Hz)
phaseoffset = 15;           % Phase offset (deg)
snrdb = 20;                 % Signal-to-noise ratio (dB)
```

Create a phase frequency offset object to introduce offsets to a modulated signal.

```
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',foffset,...
    'PhaseOffset',phaseoffset,...
    'SampleRate',fs);
```

Create a carrier synchronizer object to correct for the phase and frequency offsets. Set the `Modulation` property to `8PSK`.

```
carrierSync = comm.CarrierSynchronizer('Modulation','8PSK');
```

Generate random data and apply 8-PSK modulation.

```
data = randi([0 M-1],5000,1);
modData = pskmod(data,M,pi/M);
```

Introduce offsets to the signal and add white noise.

```
rxSig = awgn(pfo(modData),snrdb);
```

Use the carrier synchronizer to estimate the phase offset of the received signal.

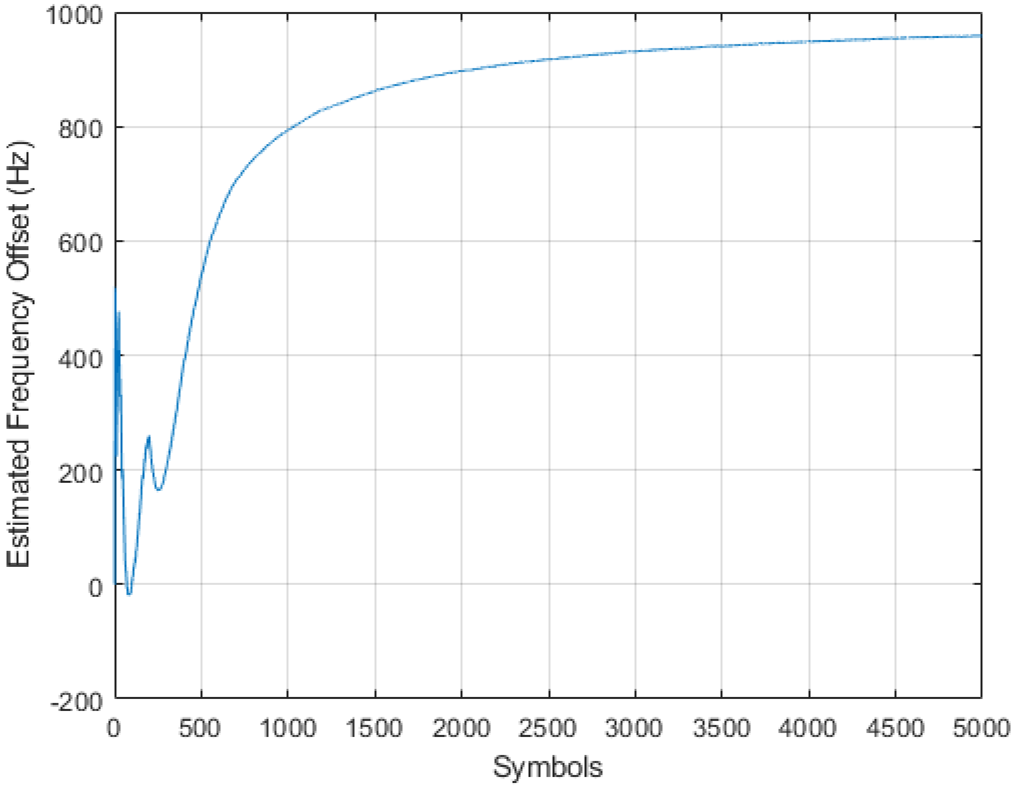
```
[~,phError] = carrierSync(rxSig);
```

Determine the frequency offset by using the `diff` function to compute an approximate derivative of the phase error. The derivative must be scaled by 2π because the phase error is measured in radians.

```
estFreqOffset = diff(phError)*fs/(2*pi);
```

Plot the running mean of the estimated frequency offset. After the synchronizer converges to a solution, the mean value of the estimate is approximately equal to the input value of 1000 Hz.

```
rmean = cumsum(estFreqOffset)./(1:length(estFreqOffset))';
plot(rmean)
xlabel('Symbols')
ylabel('Estimated Frequency Offset (Hz)')
grid
```



Correct Frequency Offset QAM Using Coarse and Fine Synchronization

Correct phase and frequency offsets for a QAM signal in an AWGN channel. Coarse frequency estimator and carrier synchronizer System objects™ are used to compensate for a significant offset.

Set the example parameters.

```
fs = 10000;           % Sample rate (Hz)
sps = 4;              % Samples per symbol
M = 16;               % Modulation order
k = log2(M);          % Bits per symbol
```

Create an AWGN channel System object™.

```
awgnChannel = comm.AWGNChannel('EbNo',15,'BitsPerSymbol',k,'SamplesPerSymbol',sps);
```

Create a pulse shaping filter

```
txFilter = comm.RaisedCosineTransmitFilter(...
    'OutputSamplesPerSymbol',sps);
rxFilter = comm.RaisedCosineReceiveFilter(...
    'InputSamplesPerSymbol',sps, ...
    'DecimationFactor', sps);
```

Create a constellation diagram object to visualize the effects of the carrier synchronization.

```
constDiagram = comm.ConstellationDiagram(...
    'ReferenceConstellation',qammod(0:M-1,M), ...
    'XLimits',[-5 5],'YLimits',[-5 5]);
```

Create a QAM coarse frequency estimator to roughly estimate the frequency offset. This is used to reduce the frequency offset of the signal passed to the carrier synchronizer. In this case, a frequency estimate to within 10 Hz is sufficient.

```
coarse = comm.QAMCoarseFrequencyEstimator('SampleRate',fs, ...
    'FrequencyResolution',10);
```

Create a carrier synchronizer System object. Because of the coarse frequency correction, the carrier synchronizer will converge quickly even though the normalized bandwidth is set to a low value. Lower normalized bandwidth values enable better correction.

```
fine = comm.CarrierSynchronizer( ...
    'DampingFactor',0.7, ...
    'NormalizedLoopBandwidth',0.005, ...
    'SamplesPerSymbol',sps,...
    'Modulation','QAM');
```

Create phase and frequency offset objects. `pfo` is used to introduce a phase and frequency offset of 30 degrees and 250 Hz, respectively. `pfc` is used to correct the offset in the received signal by using the output of the coarse frequency estimator.

```
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',250,...
    'PhaseOffset',30,...
    'SampleRate',fs);
```

```
pfc = comm.PhaseFrequencyOffset('FrequencyOffsetSource','Input port', ...
    'SampleRate',fs);
```

Generate random data symbols and apply 16-QAM modulation.

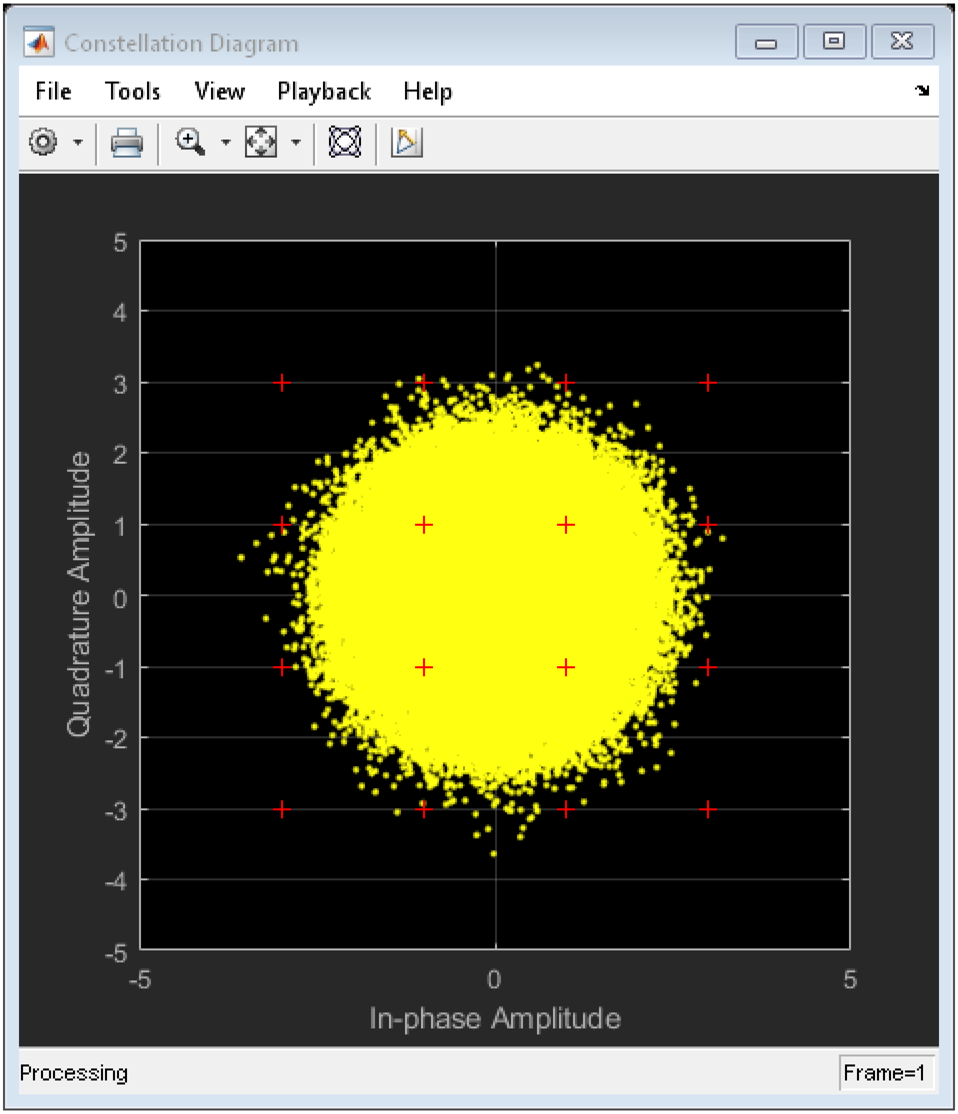
```
data = randi([0 M-1],10000,1);
txSig = txFilter(qammod(data,M));
```

Pass the signal through an AWGN channel and apply a phase and frequency offset.

```
rxSig = awgnChannel(pfo(txSig));
```

Estimate the frequency offset and compensate for it using `PFC`. Plot the constellation diagram of the output, `syncCoarse`. From the spiral nature of the diagram, you can see that the phase and frequency offsets are not corrected.

```
freqEst = coarse(rxSig);
syncCoarse = pfc(rxSig,-freqEst);
constDiagram(syncCoarse)
```

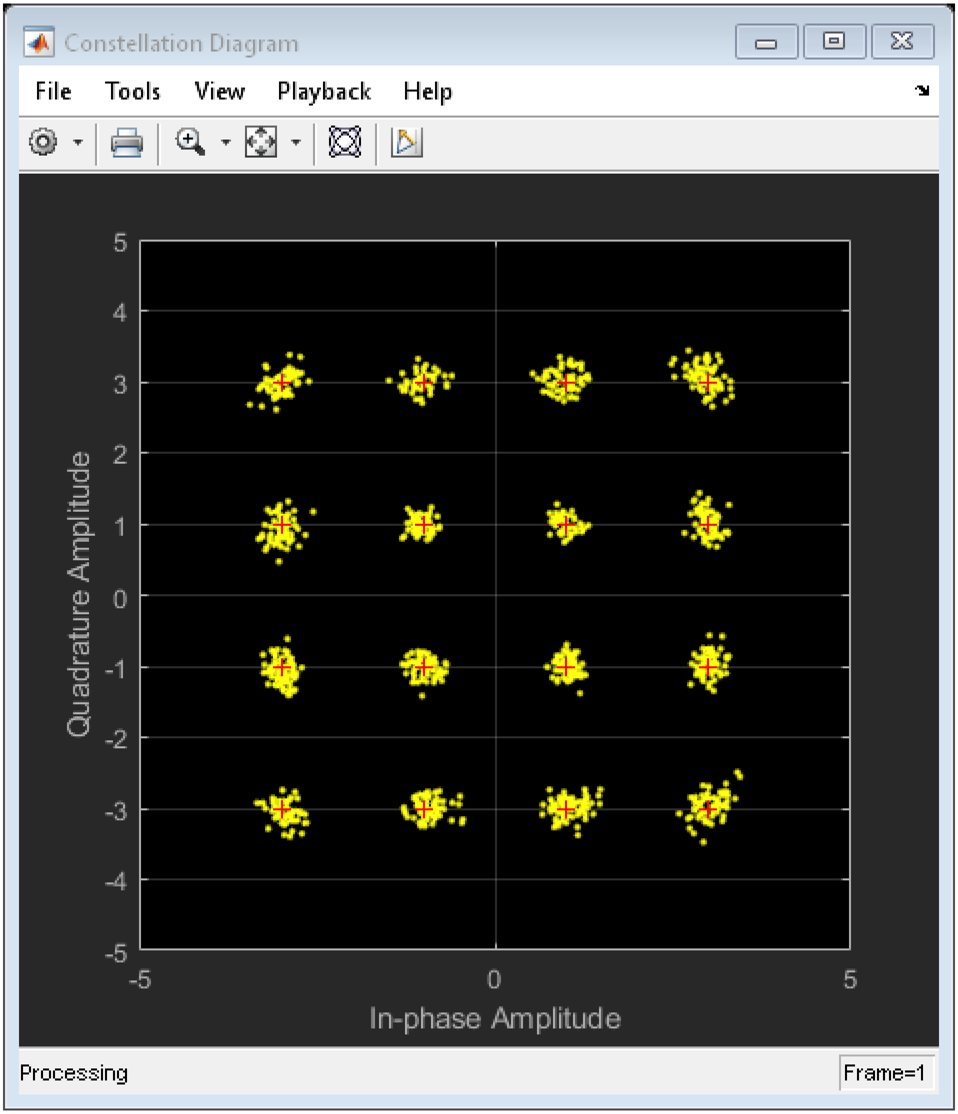


Apply fine frequency correction to the signal by using the carrier synchronizer object.

```
rxData = rxFilter(fine(syncCoarse));
```

Display the constellation diagram of the last 1000 symbols. You can see that these symbols are aligned with the reference constellation because the carrier synchronizer has converged to a solution.

```
release(constDiagram)
constDiagram(rxData(9001:10000))
```


Constellation Diagram
File
Tools
View
Playback
Help
5
4
3
2
1
0
-1
-2
-3
-4
-5
Quadrature Amplitude
-5
0
5
In-phase Amplitude
Processing
Frame=1

MSK Signal Recovery

This example shows how to model channel impairments such as timing phase offset, carrier frequency offset, and carrier phase offset for a minimum shift keying (MSK) signal. The example also shows the use of System objects™ to synchronize such signals at the receiver.

Introduction

This example models an MSK transmitted signal undergoing channel impairments such as timing, frequency, and phase offset as well as AWGN noise. An MSK timing synchronizer recovers the timing offset, while a carrier synchronizer recovers the carrier frequency and phase offsets.

Initialize system variables by using the MATLAB script `configureMSKSignalRecoveryEx`. Define logical control variables to enable timing phase and carrier frequency and phase recovery.

```
configureMSKSignalRecoveryEx;
recoverTimingPhase = true;
recoverCarrier = true;
```

Modeling Channel Impairments

Specify the sample delay, `timingOffset` , that the channel model applies, and create a variable fractional delay object to introduce the timing delay to the transmitted signal.

```
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
```

Introduce carrier phase and frequency offsets by creating a phase and frequency offset object, `PFO`. Because the MSK modulator upsamples the transmitted symbols, set the `SampleRate` property appropriately.

```
freqOffset = 50;
phaseOffset = 30;
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset', freqOffset, ...
    'PhaseOffset', phaseOffset, ...
    'SampleRate', samplesPerSymbol/Ts);
```

Create an AWGN channel to add additive white Gaussian noise to the modulated signal. The noise power is determined by the bit energy to noise power spectral density ratio

EbNo property. Because the MSK modulator generates symbols with 1 Watt of power, the signal power property of the AWGN channel is also set to 1.

```
EbNo = 20 + 10*log10(samplesPerSymbol);
chAWGN = comm.AWGNChannel(...
    'NoiseMethod', 'Signal to noise ratio (Eb/No)', ...
    'EbNo', EbNo,...
    'SignalPower', 1, ...
    'SamplesPerSymbol', samplesPerSymbol);
```

Timing Phase, Carrier Frequency, and Carrier Phase Synchronization

Construct an MSK timing synchronizer to recover symbol timing phase using a fourth-order nonlinearity method.

```
timeSync = comm.MSKTimingSynchronizer(...
    'SamplesPerSymbol', samplesPerSymbol, ...
    'ErrorUpdateGain', 0.02);
```

Construct a carrier synchronizer to recover both carrier frequency and phase. Set the modulation to QPSK, because the MSK constellation is QPSK with a 0 degree phase offset.

```
phaseSync = comm.CarrierSynchronizer(...
    'Modulation', 'QPSK', ...
    'ModulationPhaseOffset', 'Custom', ...
    'CustomPhaseOffset', 0, ...
    'SamplesPerSymbol', 1);
```

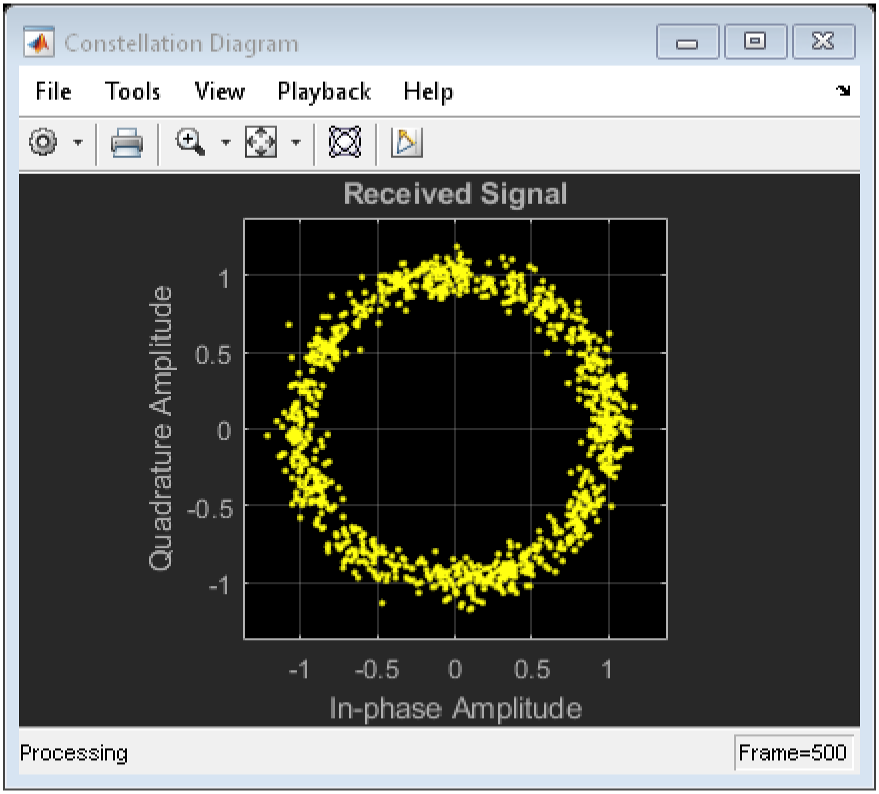
Stream Processing Loop

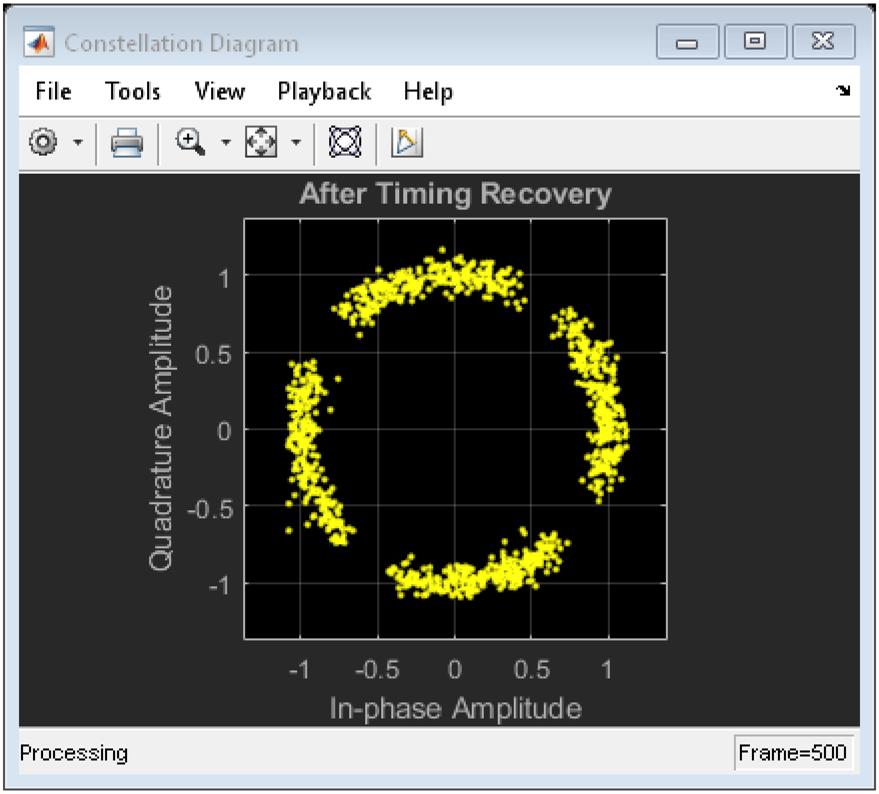
The system modulates data using MSK modulation. The modulated symbols pass through the channel model, which applies timing delay, carrier frequency and phase shift, and additive white Gaussian noise. In this system, the receiver performs timing phase, and carrier frequency and phase recovery. Finally, the system demodulates the symbols and calculates the bit error rate using an error rate calculator object. The plotResultsMSKSignalRecoveryEx script generates scatter plots to show these effects:

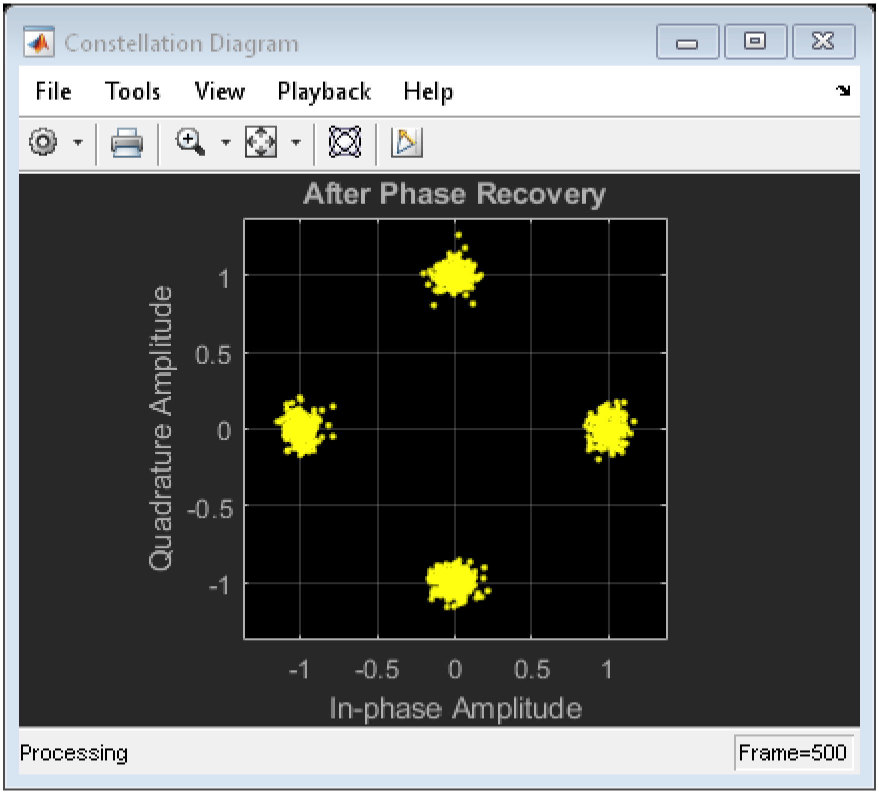
1. Channel impairments
2. Timing synchronization
3. Carrier synchronization

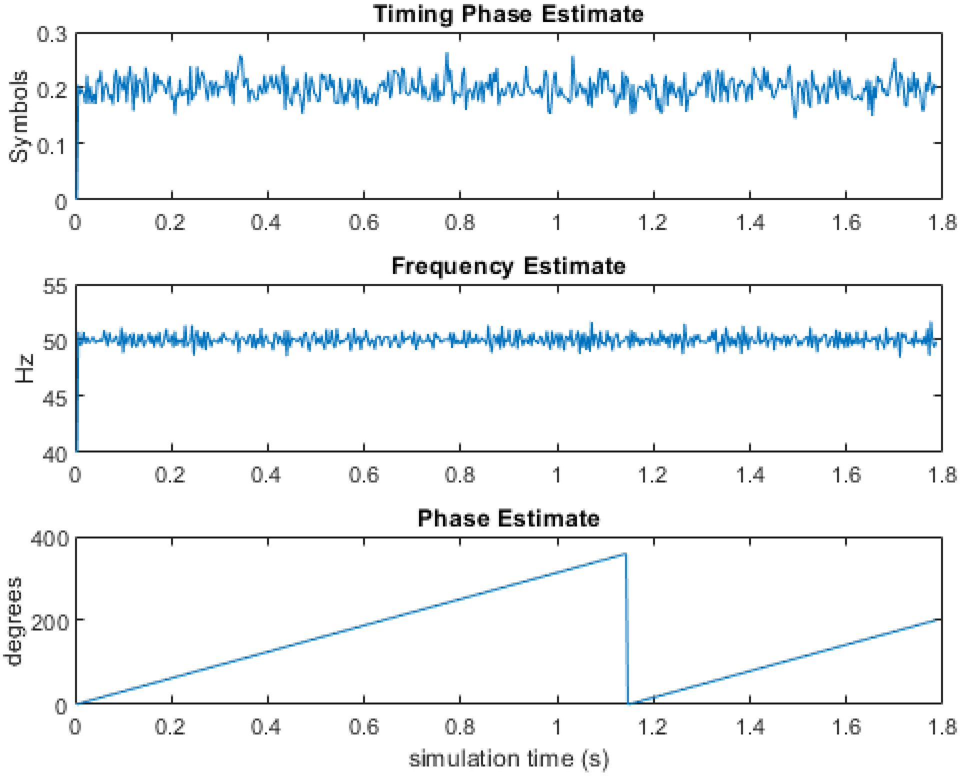
At the end of the simulation, the example displays the timing phase, frequency, and phase estimates as a function of simulation time.

```
for p = 1:numFrames
  %------------------------------------------------------------------------
  % Generate and modulate data
  %------------------------------------------------------------------------
  txBits = randi([0 1],samplesPerFrame,1);
  txSym = modem(txBits);
  %------------------------------------------------------------------------
  % Transmit through channel
  %------------------------------------------------------------------------
  %
  % Add timing offset
  rxSigTimingOff = varDelay(txSym,timingOffset*samplesPerSymbol);
  %
  % Add carrier frequency and phase offset
  rxSigCFO = pfo(rxSigTimingOff);
  %
  % Pass the signal through an AWGN channel
  rxSig = chAWGN(rxSigCFO);
  %
  % Save the transmitted signal for plotting
  plot_rx = rxSig;
  %
  %------------------------------------------------------------------------
  % Timing recovery
  %------------------------------------------------------------------------
  if recoverTimingPhase
    % Recover symbol timing phase using fourth-order nonlinearity method
    [rxSym,timEst] = timeSync(rxSig);
    % Calculate the timing delay estimate for each sample
    timEst = timEst(1)/samplesPerSymbol;
  else
    % Do not apply timing recovery and simply downsample the received signal
    rxSym = downsample(rxSig,samplesPerSymbol);
    timEst = 0;
  end

  % Save the timing synchronized received signal for plotting
  plot_rxTimeSync = rxSym;

  %------------------------------------------------------------------------
  % Carrier frequency and phase recovery
  %------------------------------------------------------------------------
  if recoverCarrier
    % The following script applies carrier frequency and phase recovery
    % using a second order PLL, and removes phase ambiguity
    [rxSym,phEst] = phaseSync(rxSym);
    removePhaseAmbiguityMSKSignalRecoveryEx;
    freqShiftEst = mean(diff(phEst)/(Ts*2*pi));
    phEst = mod(mean(phEst),360); % in degrees
  else
    freqShiftEst = 0;
    phEst = 0;
  end

  % Save the phase synchronized received signal for plotting
  plot_rxPhSync = rxSym;
  %------------------------------------------------------------------------
  % Demodulate the received symbols
  %------------------------------------------------------------------------
  rxBits = demod(rxSym);
  %------------------------------------------------------------------------
  % Calculate the bit error rate
  %------------------------------------------------------------------------
  errorStats = BERCalc(txBits,rxBits);
  %------------------------------------------------------------------------
  % Plot results
  %------------------------------------------------------------------------
  plotResultsMSKSignalRecoveryEx;
end
```


Constellation Diagram
File
Tools
View
Playback
Help
Received Signal
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=500


Constellation Diagram
File
Tools
View
Playback
Help
After Timing Recovery
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=500


Constellation Diagram
File
Tools
View
Playback
Help
After Phase Recovery
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=500



Display the bit error rate and the total number of symbols processed by the error rate calculator.

```
BitErrorRate = errorStats(1)
```

```
BitErrorRate = 4.0001e-06
```

```
TotalNumberOfSymbols = errorStats(3)
```

```
TotalNumberOfSymbols = 499982
```

Conclusion and Further Experimentation

The recovery algorithms are demonstrated by using constellation plots taken after timing, carrier frequency, and carrier phase synchronization.

Click on the *Open This Example* button to create a writable copy of this example and its supporting files. Then, to show the effects of the recovery algorithms, you can enable and disable the control variables `recoverTimingPhase` and `recoverCarrier` and rerun the simulation.

Appendix

This example uses these scripts:

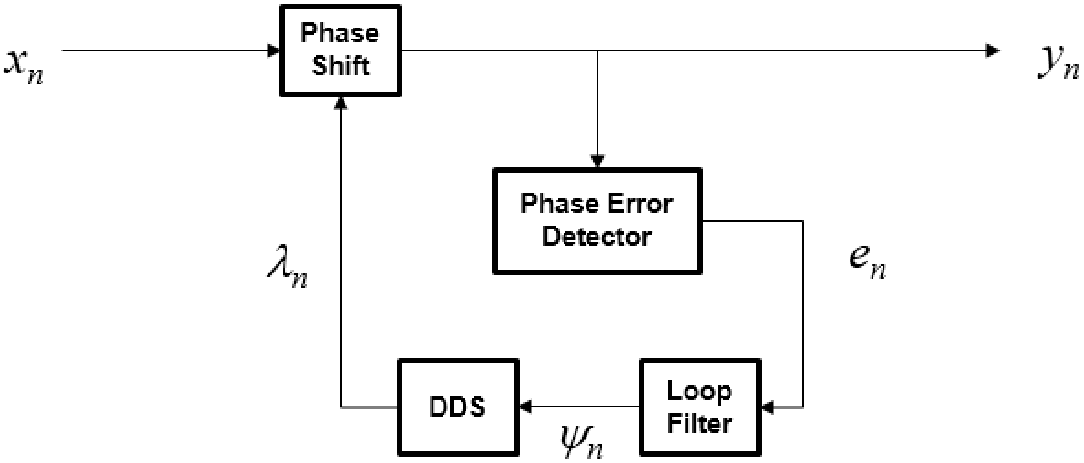
- `configureMSKSignalRecoveryEx`
- `plotResultsMSKSignalRecoveryEx`
- `removePhaseAmbiguityMSKSignalRecoveryEx`

Algorithms

The `CarrierSynchronizer` is a closed-loop compensator that uses the PLL-based algorithm described in [1]. The output of the synchronizer, y_n, is a frequency shifted version of the complex input signal, x_n. The synchronizer output is

$$y_n = x_n e^{i\lambda_n} ,$$

where λ_n is the output of the direct digital synthesizer, DDS. The DDS is the discrete-time version of a voltage-controlled oscillator and is a core component of discrete-time phase locked loops. In the context of this System object, the DDS can be thought of as an integration filter.



To correct for the frequency offset, first the algorithm determines the phase error, e_n, for the nth symbol. The value of the phase error depends on the modulation scheme.

Modulation	Phase Error
QAM or QPSK [1]	$e_n = \text{sgn}(\text{Re}(x_n)) \times \text{Im}(x_n) - \text{sgn}(\text{Im}(x_n)) \times \text{Re}(x_n)$
BPSK or PAM [1]	$e_n = \text{sgn}(\text{Re}(x_n)) \times \text{Im}(x_n)$
8-PSK [2]	$e_n = \begin{cases} \left(\sqrt{2}-1\right)\text{sgn}\left(\text{Re}\{x_n\}\right)\times\text{Im}\{x_n\} - \text{sgn}\left(\text{Im}\{x_n\}\right)\times\text{Re}\{ \\ \text{sgn}\left(\text{Re}\{x_n\}\right)\times\text{Im}\{x_n\} - \left(\sqrt{2}-1\right)\text{sgn}\left(\text{Im}\{x_n\}\right)\times\text{Re}\{ \end{cases}$
OQPSK	$e_n = \text{sgn}(\text{Re}(x_{n\text{-SamplesPerSymbol/2}})) \times \text{Im}(x_{n\text{-SamplesPerSymbol/2}}) - \text{sgn}(\text{Im}(x_n)) \times \text{Re}(x_n)$

To ensure system stability, the phase error passes through a biquadratic loop filter governed by

$$\psi_n = g_I e_n + \psi_{n-1} \text{ ,}$$

where ψ_n is the output of the loop filter at sample n, and g_I is the integrator gain. The integrator gain is determined from the expression

$$g_I = \frac{4\left(\theta^2/d\right)}{K_p K_0} \quad ,$$

where θ, d, K_0, and K_P are determined from the System object properties. Specifically,

$$\theta = \frac{B_n}{\left(\zeta + \frac{1}{4\zeta}\right)} \quad \text{and}$$

$$d = 1 + 2\zeta\theta + \theta^2 \quad ,$$

where B_n is the normalized loop bandwidth and ζ is the damping factor. The phase recovery gain, K_0, is equal to the number of samples per symbol. The phase error detector gain, K_P, is determined by the modulation type.

Modulation	K_P
QAM, QPSK, or OQPSK	2
BPSK or PAM	2
8-PSK	1

The output of the loop filter then passes to the DDS. The DDS is implemented as another biquadratic filter whose expression is based on the forward Euler integration rule such that

$$\lambda_n = \left(g_P e_{n-1} + \psi_{n-1}\right) + \lambda_{n-1} \ ,$$

where g_P is the proportional gain that is expressed as

$$g_P = \frac{4\zeta\left(\theta/d\right)}{K_p K_0} \ .$$

The `info` method of the System object returns estimates of the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay. The normalized pull-in range, $(\Delta f)_{\text{pull-in}}$, expressed in radians, is calculated as

$$(\Delta f)_{\text{pull-in}} \approx \min\left(1, 2\pi\sqrt{2}\zeta B_n\right).$$

The expression for $(\Delta f)_{\text{pull-in}}$ becomes less accurate as $2\pi\sqrt{2}\zeta B_n$ approaches 1.

The maximum frequency and phase lock delays, T_{FL} and T_{PL}, expressed in samples, are given by

$$T_{FL} \approx 4\frac{(\Delta f)^2_{\text{pull-in}}}{B_n^3},$$

$$T_{PL} \approx \frac{1.3}{B_n}.$$

Selected Bibliography

[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 359–393.

[2] Huang, Zhijie, Zhiqiang Yi, Ming Zhang, and Kuang Wang. “8PSK Demodulation for New Generation DVB-S2.” *International Conference on Communications, Circuits and Systems, 2004. ICCCAS 2004.* Vol. 2, 2004, pp. 1447–1450.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.SymbolSynchronizer`

Introduced in R2015a

info

System object: comm.CarrierSynchronizer
Package: comm

Characteristic information about carrier synchronizer

Syntax

```
S = info(OBJ)
```

Description

`S = info(OBJ)` returns a structure, `S`, containing characteristic information for the `CarrierSynchronizer` System object, `OBJ`. `S` has these fields:

- `NormalizedPullInRange` is the largest frequency offset (rad), normalized by the loop bandwidth, for which the synchronizer can acquire lock.
- `MaxFrequencyLockDelay` is the number of samples required for the synchronizer to acquire frequency lock.
- `MaxPhaseLockDelay` is the number of samples required for the synchronizer to acquire phase lock.

Examples

Determine Carrier Synchronizer Loop Parameters

Create a carrier synchronizer object.

```
csync = comm.CarrierSynchronizer;
```

Determine the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay by using the `info` method.

```
syncInfo = info(csync)
syncInfo = struct with fields:
    NormalizedPullInRange: 0.0628
    MaxFrequencyLockDelay: 1.5787e+04
        MaxPhaseLockDelay: 130
```

The normalized pull-in range is 0.0628 rad/sec. Convert the pull-in range to Hz. This represents the maximum normalized frequency offset that can be corrected by the carrier synchronizer.

```
foffsetmax = syncInfo.NormalizedPullInRange/(2*pi)
```

```
foffsetmax = 0.0100
```

The time to acquire a frequency lock is 15787 s, and the time to acquire a phase lock is 130 s.

The overall acquisition time, `Tlock`, is well approximated by the sum of the frequency and phase lock terms.

```
Tlock = syncInfo.MaxFrequencyLockDelay + syncInfo.MaxPhaseLockDelay
```

```
Tlock = 1.5917e+04
```

reset

System object: comm.CarrierSynchronizer
Package: comm

Reset states of the carrier synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CarrierSynchronizer` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: comm.CarrierSynchronizer
Package: comm

Compensate for carrier frequency and phase offset

Syntax

```
[Y,P] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,P] = step(H,X)` compensates for frequency and phase offsets in the input signal, `X`, and returns a compensated signal, `Y`, and an estimate of the phase error, `P`, in radians. The input `X` is complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` has the same properties as `X`, while `P` is real scalar or column vector having the same dimensions as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CCDF System object

Package: comm

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.

To measure complementary cumulative distribution of a signal:

1. Define and set up your CCDF object. See “Construction” on page 3-270 .
2. Call `step` to measure complementary cumulative distribution according to the properties of `comm.CCDF`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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

getPercentileRelativePower	Get relative power value for a given probability
getProbability	Get the probability for a given relative power value
plot	Plot CCDF curves
reset	Reset states of CCDF measurement object
step	Measure complementary cumulative distribution function

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Obtain CCDF curves for 16-QAM and QPSK signals

Create a CCDF System object and specify that it output average power and peak power measurements.

```
ccdf = comm.CCDF('AveragePowerOutputPort',true, ...
    'PeakPowerOutputPort',true);
```

Generate 16-QAM and QPSK modulated signals.

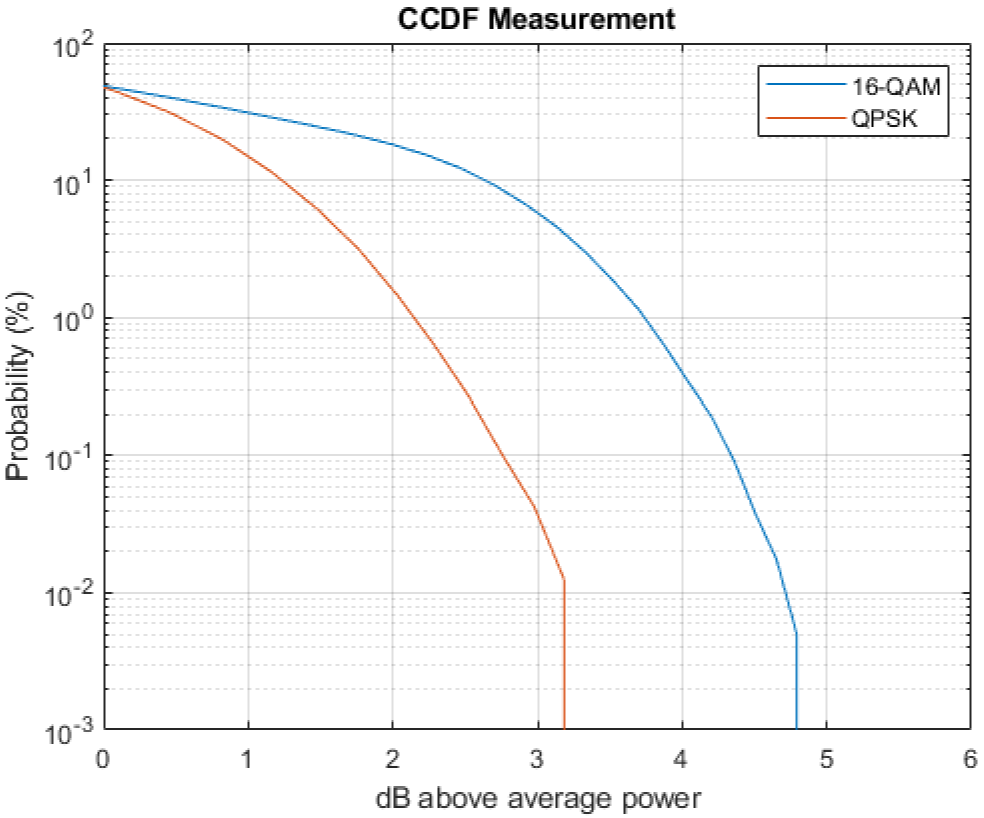
```
qamTxSig = qammod(randi([0 15],20e3,1),16,'UnitAveragePower',true);
qpskTxSig = pskmod(randi([0 3],20e3,1),4,pi/4);
```

Pass the signals through an AWGN channel.

```
qamRxSig = awgn(qamTxSig,15);
qpskRxSig = awgn(qpskTxSig,15);
```

Measure the CCDF of the two waveforms. Plot the CCDF using the `plot` method of `comm.CCDF`.

```
[CCDFy,CCDFx,AvgPwr,PeakPwr] = ccdf([qamRxSig qpskRxSig]);

plot(ccdf)
legend('16-QAM','QPSK')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.EVM` | `comm.MER`

Introduced in R2012a

getPercentileRelativePower

System object: comm.CCDF
Package: comm

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)
```

getProbability

System object: comm.CCDF
Package: comm

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
```

plot

System object: comm.CCDF
Package: comm

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.

reset

System object: comm.CCDF
Package: comm

Reset states of CCDF measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CCDF` object, `H`.

step

System object: comm.CCDF
Package: comm

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

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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.CoarseFrequencyCompensator System object

Package: comm

Compensate for frequency offset for PAM, PSK, or QAM

Description

The `CoarseFrequencyCompensator` System object compensates for the frequency offset of received signals.

To compensate for the frequency offset of a PAM, PSK, or QAM signal:

1. Define and set up your coarse frequency compensator object. See "Construction" on page 3-284.
2. Call `step` to compensate for the frequency offset of a PAM, PSK, or QAM signal according to the properties of `comm.CoarseFrequencyCompensator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`CFC = comm.CoarseFrequencyCompensator` creates a coarse frequency offset compensator object, `CFC`. This object uses an open-loop technique to estimate and compensate for the carrier frequency offset in a received signal.

`CFC = comm.CoarseFrequencyCompensator(Name,Value)` creates a coarse frequency offset compensator object, `CFC`, 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

Modulation

Modulation type

Specify the signal modulation type as BPSK, QPSK, OQPSK, 8PSK, PAM, or QAM. The default is QAM. This property is nontunable.

Algorithm

Algorithm used to estimate frequency offset

Specify the estimation algorithm as one of FFT-based or Correlation-based. The default is FFT-based. This property is nontunable.

The table shows the allowable combinations of the modulation type and the estimation algorithm.

Modulation	FFT-Based Algorithm	Correlation-Based Algorithm
BPSK, QPSK, 8PSK, PAM	✓	✓
OQPSK, QAM	✓	

Use the correlation-based algorithm for HDL implementations and for other situations in which you want to avoid using an FFT.

This property appears when Modulation is 'BPSK', 'QPSK', '8PSK', or 'PAM'.

FrequencyResolution

Frequency resolution (Hz)

Specify the frequency resolution for the offset frequency estimation as a positive, real scalar of data type double. This property establishes the FFT length used to perform spectral analysis and must be less than the sample rate. The default is 0.001. This property is nontunable.

MaximumFrequencyOffset

Maximum measurable frequency offset (Hz)

Specify the maximum measurable frequency offset as a positive, real scalar of data type `double`.

The value of this property must be less than f_{samp} / M, where f_{samp} is the sample rate and M is the modulation order. As a best practice, set `MaximumOffset` to less than $r / (4M)$. This property applies only if `Algorithm` is `Correlation-based`. The default is `0.05`. This property is nontunable.

`SampleRate`

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type `double`. The default is `1`. This property is nontunable.

`SamplesPerSymbol`

Samples per symbol

Specify the number of samples per symbol, s, as a real positive finite integer scalar, such that $s \geq 2$. The default value is `4`. This property is nontunable.

This property appears when `Modulation` is `'OQPSK'`.

Methods

info	Characteristic information about coarse frequency compensator
reset	Reset states of the `CoarseFrequencyCompensator` object
step	Compensate for frequency offset

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Compensate for Frequency Offset in a QPSK Signal

Compensate for a 4 kHz frequency offset imposed on a noisy QPSK signal.

Set the example parameters.

```
nSym = 2048;       % Number of input symbols
sps = 4;           % Samples per symbol
nSamp = nSym*sps;  % Number of samples
fs = 80000;        % Sampling frequency (Hz)
```

Create a square root raised cosine transmit filter.

```
txfilter = comm.RaisedCosineTransmitFilter(...
    'RolloffFactor',0.2, ...
    'FilterSpanInSymbols',8,   ...
    'OutputSamplesPerSymbol',sps);
```

Create a phase frequency offset object to introduce the 4 kHz frequency offset.

```
freqOffset = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',-4000, ...
    'SampleRate',fs);
```

Create a coarse frequency compensator object to compensate for the offset.

```
freqComp = comm.CoarseFrequencyCompensator(...
    'Modulation','QPSK', ...
    'SampleRate',fs, ...
    'FrequencyResolution',1);
```

Generate QPSK symbols, filter the modulated data, pass the signal through an AWGN channel, and apply the frequency offset.

```
data = randi([0 3],nSym,1);
modData = pskmod(data,4,pi/4);
txSig = txfilter(modData);
rxSig = awgn(txSig,20,'measured');
offsetData = freqOffset(rxSig);
```

Compensate for the frequency offset using `freqComp`. When the frequency offset is high, it is beneficial to do coarse frequency compensation prior to receive filtering because filtering suppresses energy in the useful spectrum.

```
[compensatedData,estFreqOffset] = freqComp(offsetData);
```

Display the estimate of the frequency offset.

```
estFreqOffset
```

```
estFreqOffset = -3.9999e+03
```

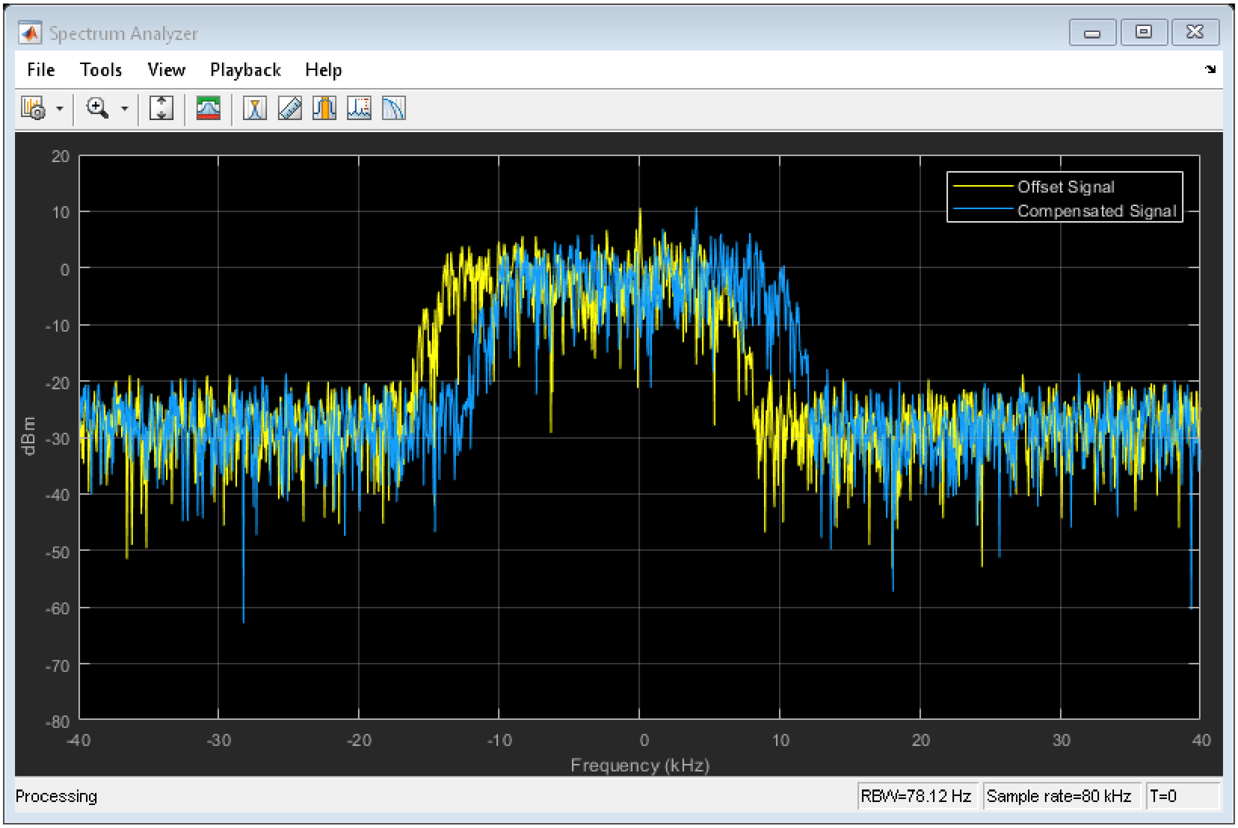
Return information about the `freqComp` object. To obtain the FFT length, you must call `freqComp` prior to calling the `info` method.

```
freqCompInfo = info(freqComp)
```

```
freqCompInfo = struct with fields:
    FFTLength: 131072
    Algorithm: 'FFT-based'
```

Create a spectrum analyzer object and plot the offset and compensated spectra. Verify that the compensated signal has a center frequency at 0 Hz and that the offset signal has a center frequency at -4 kHz.

```
specAnal = dsp.SpectrumAnalyzer('SampleRate',fs,'ShowLegend',true, ...
    'ChannelNames',{'Offset Signal' 'Compensated Signal'});
specAnal([offsetData compensatedData])
```



Compensate for Frequency Offset Using Coarse and Fine Compensation

Correct for a phase and frequency offset in a noisy QAM signal using a carrier synchronizer. Then correct for the offsets using both a carrier synchronizer and a coarse frequency compensator.

Set the example parameters.

```
fs = 10000;             % Symbol rate (Hz)
sps = 4;                % Samples per symbol
M = 16;                 % Modulation order
k = log2(M);            % Bits per symbol
```

Create a QAM modulator and an AWGN channel.

```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',k,'SamplesPerSymbol',sps);
```

Create a constellation diagram object to visualize the effects of the offset compensation techniques. Specify the constellation diagram to display only the last 4000 samples.

```
constdiagram = comm.ConstellationDiagram(...
    'ReferenceConstellation',qammod(0:M-1,M), ...
    'SamplesPerSymbol',sps, ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',4000, ...
    'XLimits',[-5 5],'YLimits',[-5 5]);
```

Introduce a frequency offset of 400 Hz and a phase offset of 30 degrees.

```
phaseFreqOffset = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',400,...
    'PhaseOffset',30,...
    'SampleRate',fs);
```

Generate random data symbols and apply 16-QAM modulation.

```
data = randi([0 M-1],10000,1);
modSig = qammod(data,M);
```

Create a raised cosine filter object and filter the modulated signal.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps, ...
    'Gain',sqrt(sps));
txSig = txfilter(modSig);
```

Apply the phase and frequency offset, and then pass the signal through the AWGN channel.

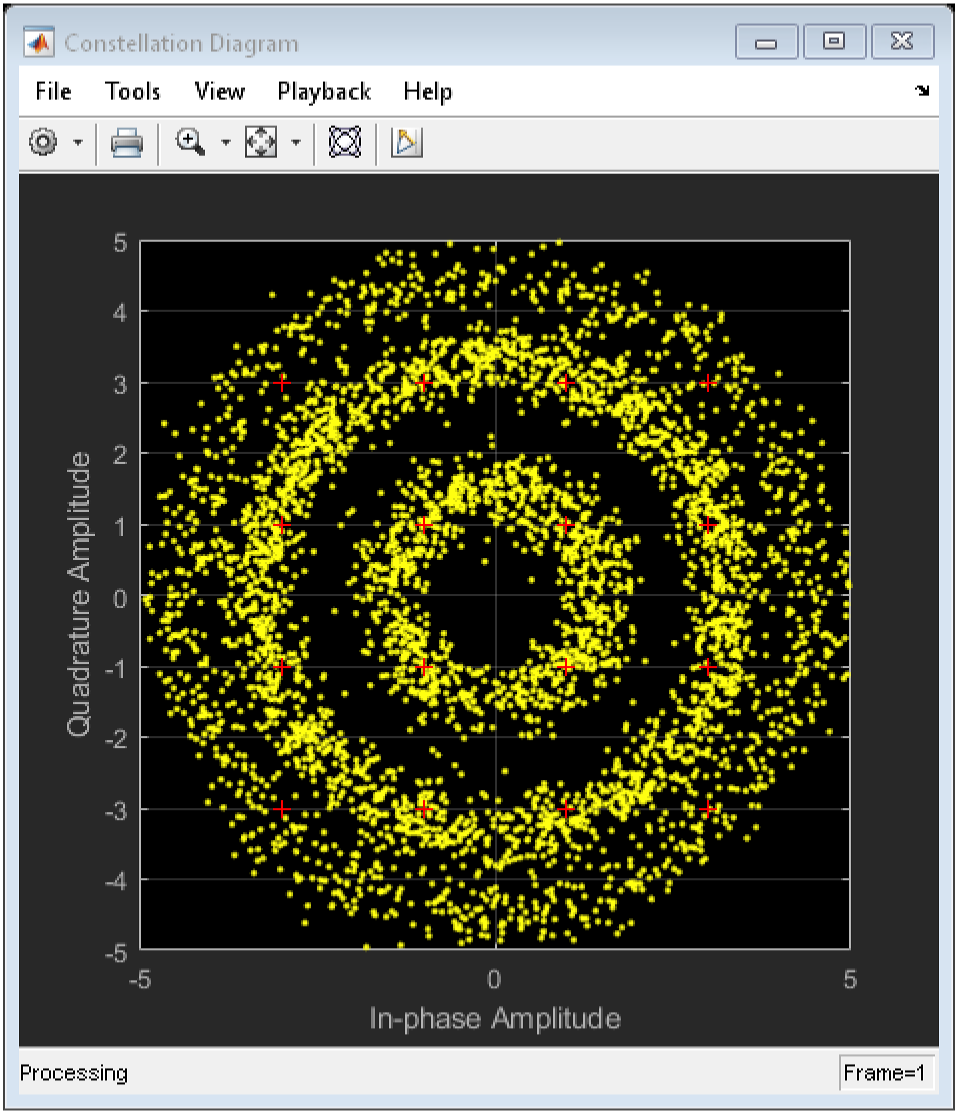
```
freqOffsetSig = phaseFreqOffset(txSig);
rxSig = channel(freqOffsetSig);
```

Apply fine frequency correction to the signal by using the carrier synchronizer.

```
fineSync = comm.CarrierSynchronizer('DampingFactor',0.7, ...
    'NormalizedLoopBandwidth',0.005, ...
    'SamplesPerSymbol',sps, ...
    'Modulation','QAM');
rxData = fineSync(rxSig);
```

Display the constellation diagram of the last 4000 symbols.

```
constdiagram(rxData)
```



Even with time to converge, the spiral nature of the plot shows that the carrier synchronizer has not yet compensated for the large frequency offset. The 400 Hz offset is 1% of the sample rate.

Repeat the process with a coarse frequency compensator inserted before the carrier synchronizer.

Create a coarse frequency compensator to reduce the frequency offset to a manageable level.

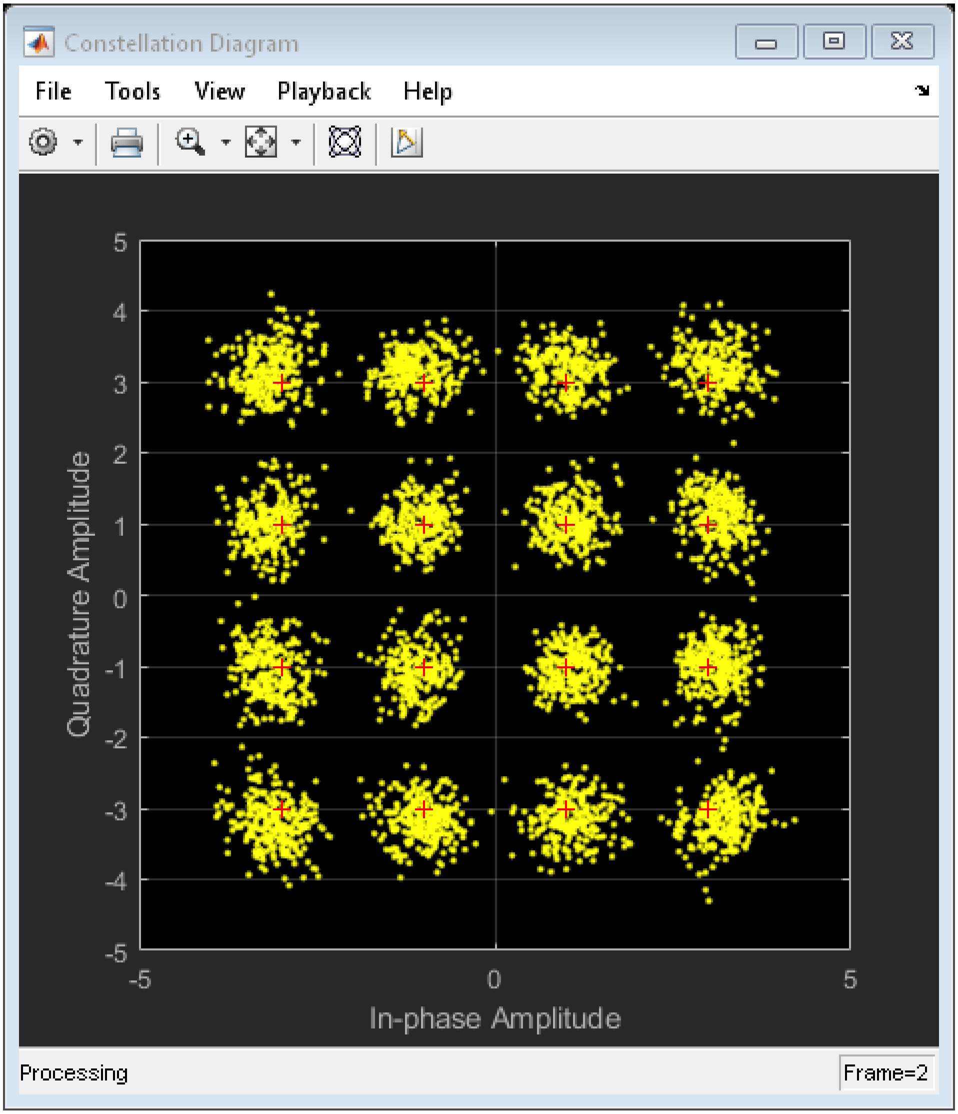
```
coarseSync = comm.CoarseFrequencyCompensator('Modulation','QAM','SampleRate',fs*sps);
```

Pass the received signal to the coarse frequency compensator and then to the carrier synchronizer.

```
syncCoarse = coarseSync(rxSig);
rxData = fineSync(syncCoarse);
```

Plot the constellation diagram of the signal after coarse and fine frequency compensation.

```
constdiagram(rxData)
```



The received data now aligns with the reference constellation.

Algorithms

Correlation-Based

The correlation-based estimation algorithm, which can be used to estimate the frequency offset for PSK and PAM signals, is described in [1]. To determine the frequency offset, *Δf*, the algorithm performs a maximum likelihood (ML) estimation of the complex-valued oscillation exp(*j2πΔft*). The observed signal, r_k, is represented as

$$r_k = e^{j(2\pi\Delta f k \mathrm{T}_s + \theta)}, 1 \le k \le N \ ,$$

where T_s is the sampling interval, *θ* is an unknown random phase, and *N* is the number of samples. The maximum likelihood estimation of the frequency offset is equivalent to seeking the maximum of the likelihood function, *Λ(Δf)*,

$$\Lambda(\Delta f) \approx \left| \sum_{i=1}^{N} r_i e^{-j2\pi\Delta f i T_s} \right|^2 = \sum_{k=1}^{N} \sum_{m=1}^{N} r_k r_m^* e^{-j2\pi\Delta f T_s (k-m)} \ .$$

After simplifying, the problem is expressed as a discrete Fourier transform, weighted by a parabolic windowing function. It is expressed as

$$\mathrm{Im}\left\{ \sum_{k=1}^{N-1} k(N-k)R(k)e^{j2\pi\Delta\breve{f}T_s} \right\} = 0 \, ,$$

where *R(k)* denotes the estimated autocorrelation of the sequence r_k and is represented as

$$R(k) \triangleq \frac{1}{N-k} \sum_{i=k+1}^{N} r_i \, r_{i-k}^*, \ 0 \le k \le N-1 \, .$$

The term *k(N–k)* is the parabolic windowing function. In [1], it is shown that *R(k)* is a poor estimate of the autocorrelation of r_k when *k* = 0 or when *k* is close to *N*. Consequently, the windowing function can be expressed as a rectangular sequence of 1s for *k* = 1, 2, ..., *L*, where *L* ≤ *N* – 1. The results is a modified ML estimation strategy in which

$$\operatorname{Im}\left\{\sum_{k=1}^{L} R(k)e^{-j2\pi\Delta\breve{f}kT_s}\right\}=0\,.$$

This results in an estimate of $\Delta\hat{f}$ in which

$$\Delta\breve{f}\cong\frac{f_{samp}}{\pi(L+1)}\arg\left\{\sum_{k=1}^{L} R(k)\right\}.$$

The sampling frequency, f_{samp}, is the reciprocal of T_s. The number of elements used to compute the autocorrelation sequence, L, are determined as

$$L=\operatorname{round}\left(\frac{f_{samp}}{f_{max}}\right)-1,$$

where f_{max} is the maximum expected frequency offset and `round` is the nearest integer function. The frequency offset estimate improves when $L \geq 7$ and leads to the recommendation that $f_{max} \leq f_{samp} / (4M)$.

FFT-Based

FFT-based algorithms can be used to estimate the frequency offset for all modulation types. Two variations are used in comm.CoarseFrequencyCompensator.

- For `BPSK`, `QPSK`, `8PSK`, `PAM`, or `QAM` modulations the FFT-based algorithm used is described in [2]. The algorithm estimates $\Delta\hat{f}$ by using a periodogram of the m^{th} power of the received signal and is given as

  $$\Delta\breve{f}=\frac{f_{samp}}{N\cdot m}\arg\max_{f}\left|\sum_{k=0}^{N-1} r^{m}(k)e^{-j2\pi kt/N}\right|,\quad\left(-\frac{R_{sym}}{2}\leq f\leq\frac{R_{sym}}{2}\right),$$

  where m is the modulation order, $r(k)$ is the received sequence, R_{sym} is the symbol rate, and N is the number of samples. The algorithm searches for a frequency that maximizes the time average of the m^{th} power of the received signal multiplied by various frequencies in the range of $[-R_{sym}/2, R_{sym}/2]$. As the form of the algorithm is the definition of the discrete Fourier transform of *$r^m(t)$*, searching for a frequency that maximizes the time average is equivalent to searching for a peak line in the spectrum of *$r^m(t)$*. The number of points required by the FFT is

$$N = 2^{\left\lceil \log_2\left(\frac{f_{samp}}{f_r}\right)\right\rceil},$$

where f_r is the desired frequency resolution.

- For `OQPSK` modulation the FFT-based algorithm used is described in [4]. The algorithm searches for spectral peaks at +/- 200 kHz around the symbol rate. This technique locates desired peaks in the presence of interference from spectral content around baseband frequencies due to filtering.

References

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions.” *IEEE Transactions on Communications*. Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

[2] Wang, Y., K. Shi, and E. Serpedi. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach.” *EURASIP Journal on Applied Signal Processing*. 2004:13, pp. 1993–2001.

[3] Nakagawa, T., M. Matsui, T. Kobayashi, K. Ishihara, R. Kudo, M. Mizoguchi, and Y. Miyamoto. “Non-Data-Aided Wide-Range Frequency Offset Estimator for QAM Optical Coherent Receivers.” *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*. March 2011, pp. 1–3.

[4] Olds, Jonathan. "*Designing an OQPSK demodulator*", http://jontio.zapto.org/hda1/oqpsk.html.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer` | `comm.PhaseFrequencyOffset` | `dsp.FFT`

Introduced in R2015b

info

System object: comm.CoarseFrequencyCompensator
Package: comm

Characteristic information about coarse frequency compensator

Syntax

```
S = info(CFC)
```

Description

`S = info(CFC)` returns a structure, `S`, containing characteristic information for the `CoarseFrequencyCompensator` System object, `CFC`. `S` has fields `FFTLength`, `Algorithm`, and `MaxLag`. `Algorithm` is the type of algorithm used in estimating the frequency offset. `FFTLength` is the number of samples used in the FFT and is provided when the algorithm is `FFT-based`. `MaxLag` is the number of samples used in estimating the autocorrelation and is provided when the algorithm is `Correlation-based`.

Note The `step` method must be run once to determine the `FFTLength`.

reset

System object: comm.CoarseFrequencyCompensator
Package: comm

Reset states of the `CoarseFrequencyCompensator` object

Syntax

```
reset(CFC)
```

Description

`reset(CFC)` resets the internal states of the `CoarseFrequencyCompensator` object, `CFC`.

step

System object: comm.CoarseFrequencyCompensator
Package: comm

Compensate for frequency offset

Syntax

```
Y = step(CFC,X)
[Y,EST] = step(CFC,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(CFC,X)` compensates for the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a column vector. The step method outputs the compensated signal `Y` as a complex column vector having the same dimensions and data type as `X`.

`[Y,EST] = step(CFC,X)` returns a scalar estimate of the frequency offset, `EST`.

Note `CFC` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ConstellationDiagram System object

Package: comm

Display a constellation diagram for input signals

Description

The `ConstellationDiagram` System object plots constellation diagrams, plots signal trajectory, and provides the ability to perform EVM and MER measurements.

To plot constellation diagrams:

1. Define and set up your constellation diagram object. See “Construction” on page 3-301.
2. Call `step` to display a constellation diagram figure according to the properties of `comm.ConstellationDiagram`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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. 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.

SymbolsToDisplaySource

Specify the source of symbols to display as one of `Input frame length` | `Property`]. When you set the `SymbolsToDisplaySource` to `Input frame length`, the object calculates the number of symbols to display as the input frame length divided by the value of the SamplesPerSymbol property. When you set this property to `Property`, the maximum number of symbols to display is the value of the SymbolsToDisplay property. The default is `Input frame length`. This property is tunable.

SymbolsToDisplay

The maximum number of symbols that can be displayed when input signal is long.

This property is applicable when you set the SymbolsToDisplaySource property to `Property`. 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.

ShowTrajectory

Option to turn on the signal trajectory plot.

Set this property to `true` to display a plot of the signal trajectory. The signal trajectory is a plot of the in-phase component versus the quadrature component of a modulated signal. The default value of this property is `false`. 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. The default value of this property is an empty character vector. 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 character vector. 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 character vector. The default value of this property is `'Quadrature Amplitude'`. This property is tunable.

EnableMeasurements

Option to turn on measurements

Set this property to `true` to activate the measurements pane, compute and display EVM or MER measurements. The default value of this property is `false`. This property is tunable.

MeasurementInterval

The measurement interval

Specify `MeasurementInterval` as `'Current Display'`, `'All displays'`, or as a positive integer.

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 from `2` to SymbolsToDisplay. The measurement is computed after the number of input data samples is greater than `MeasurementInterval`. The default value of this property is set to `'Current Display'`, indicating the measurement interval is equal to the length of the input. 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

hide	Hide scope window
reset	Reset internal states of the scope object
show	Make scope window visible
step	Display constellation diagram of signal in scope figure

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Plot 16-QAM Constellation

This example shows how to create a 16-QAM modulator, transmit data using an AWGN channel, and plot the signal constellation.

Create a Rectangular QAM Modulator System object, `qamModulator`, and set the modulation order to 16. Find the constellation reference points using the `constellation` function.

```
qamModulator = comm.RectangularQAMModulator('ModulationOrder',16);
refC = constellation(qamModulator);
```

Create a constellation diagram System object and specify the constellation reference points and axes limits using name-value pairs.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refC, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Generate random, 16-ary data symbols.

```
d = randi([0 15],1000,1);
```

Apply 16-QAM modulation.

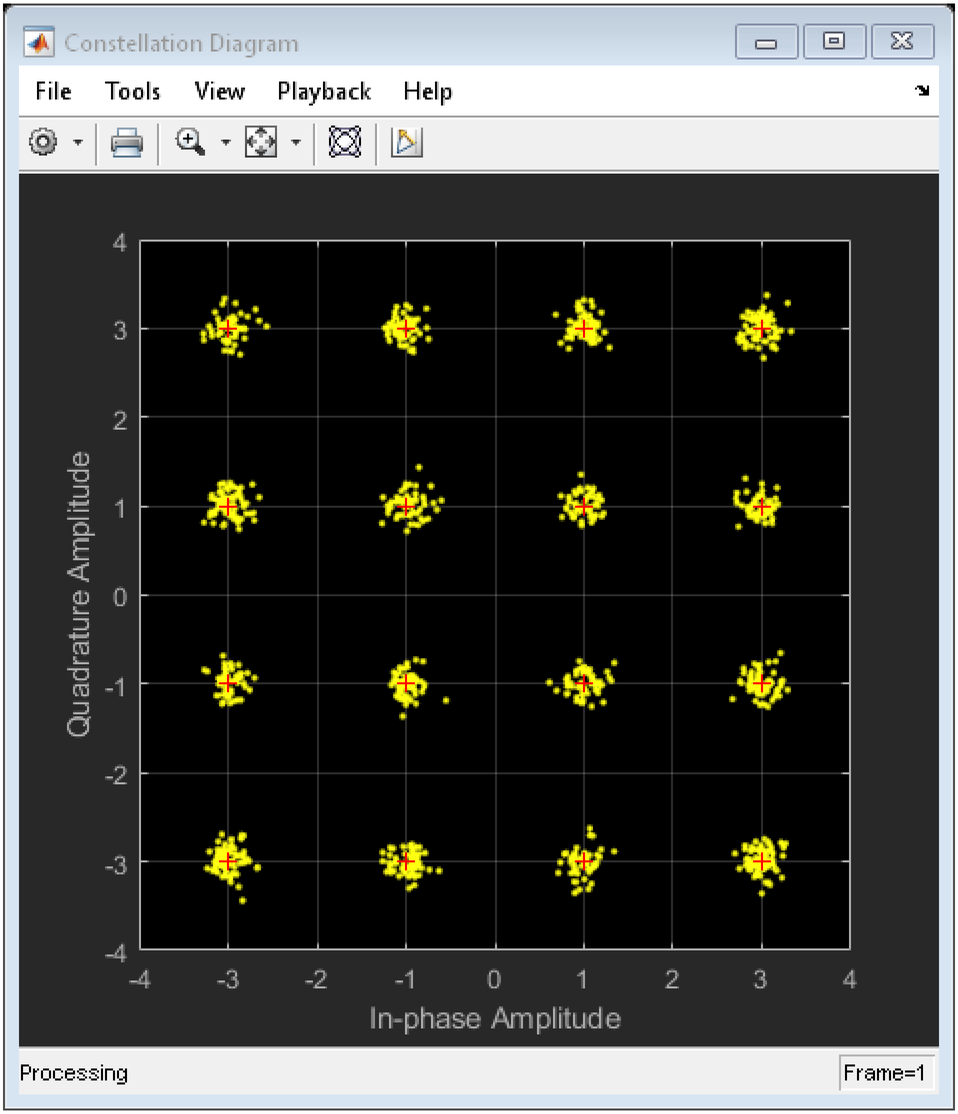
```
sym = step(qamModulator,d);
```

Pass the modulated signal through an AWGN channel.

```
rcv = awgn(sym,15);
```

Display the constellation diagram.

```
constDiagram(rcv)
```


Constellation Diagram
File
Tools
View
Playback
Help
Quadrature Amplitude
In-phase Amplitude
4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
Processing
Frame=1

Plot Amplitude Imbalanced QPSK Constellation

This example shows how to modulate random data symbols, apply an amplitude imbalance, pass the signal through a noisy channel, and plot the resultant constellation.

Create a constellation diagram object. Because the default reference constellation for the `comm.ConstellationDiagram` System object is QPSK, it is not necessary to set additional properties.

```
constDiagram = comm.ConstellationDiagram;
```

Create an AWGN channel.

```
channel = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Es/No)', ...
    'EsNo',20);
```

Generate random data symbols and apply QPSK modulation.

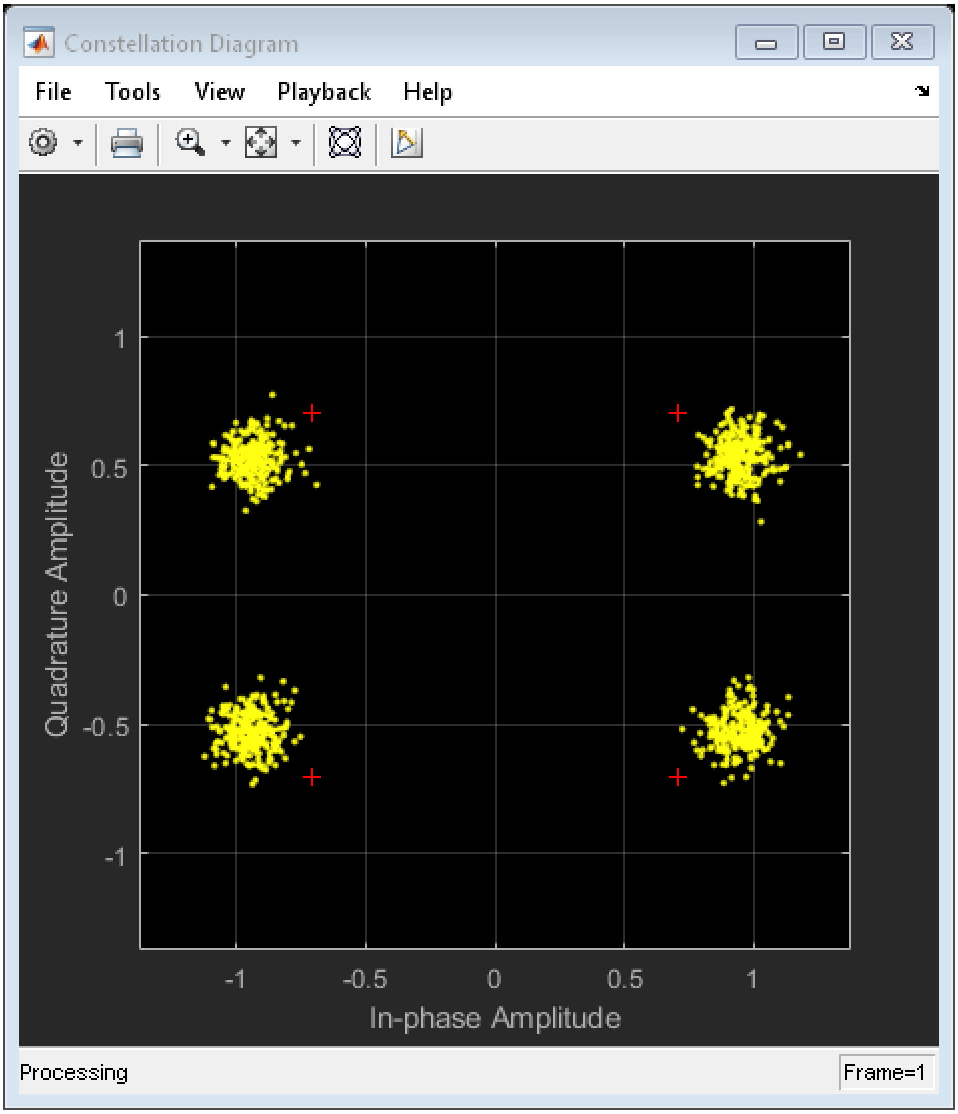
```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Apply an amplitude imbalance to the modulated signal.

```
txSig = iqimbal(modData,5);
```

Pass the transmitted signal through the AWGN channel and display the constellation diagram. Observe that the data points have shifted from their ideal locations.

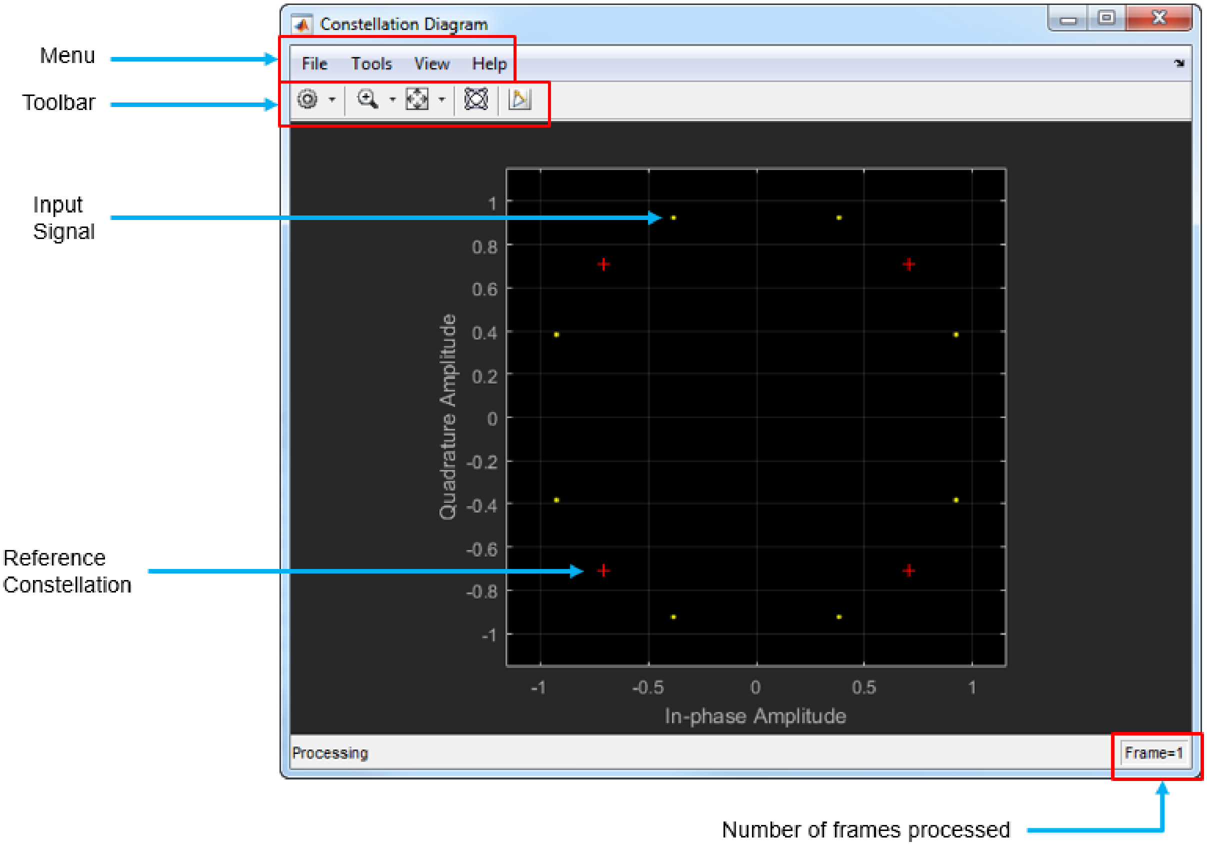
```
rxSig = channel(txSig);
constDiagram(rxSig)
```


Constellation Diagram
File
Tools
View
Playback
Help
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=1

Signal Display

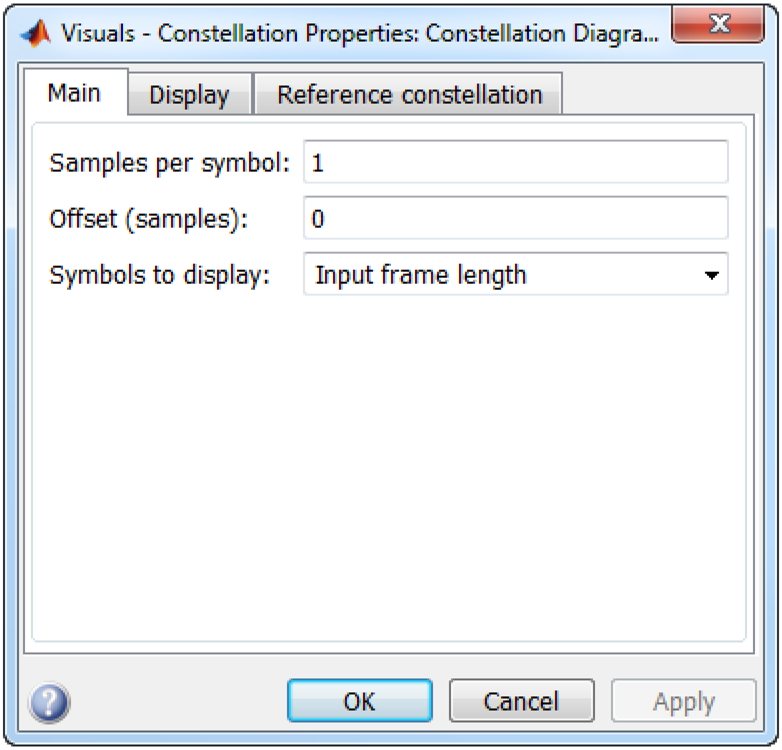
To change the signal display settings, select **View** > **Configuration Properties** to bring up the Visuals—Constellation Properties dialog box. Then, modify the values for the **Samples per symbol**, **Offset** and **Symbols to display** parameters on the **Main** tab. You can modify the reference constellation parameters on the **Reference constellation** 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.



Visuals — Constellation Properties

Main Pane



Samples per symbol: 1

Offset (samples): 0

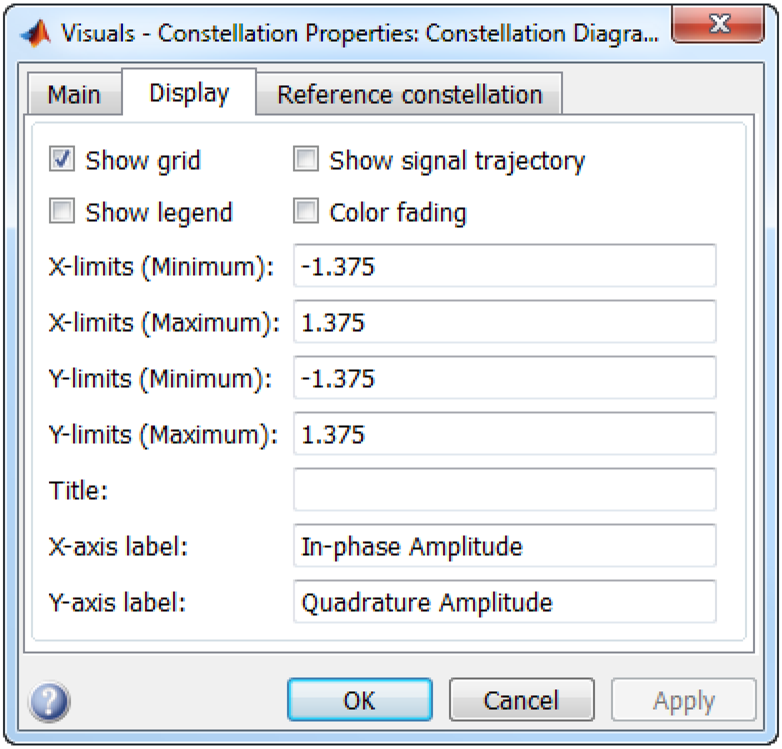
Symbols to display: Input frame length

Number of samples used to represent a symbol. This value must be a positive number.

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.

The maximum number of symbols that can be displayed. Must be a positive integer value.

Display Pane



☑ Show grid
□ Show signal trajectory
□ Show legend
□ Color fading

X-limits (Minimum): -1.375

X-limits (Maximum): 1.375

Y-limits (Minimum): -1.375

Y-limits (Maximum): 1.375

Title:

X-axis label: In-phase Amplitude

Y-axis label: Quadrature Amplitude

Select this check box to turn on the grid.

Select this check box to display a legend for the graph.

Select this check box to display the trajectory of a modulated signal by plotting its in-phase component versus its quadrature component.

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.

Specify the minimum value of the x-axis.

Specify the maximum value of the x-axis.

Specify the minimum value of the y-axis.

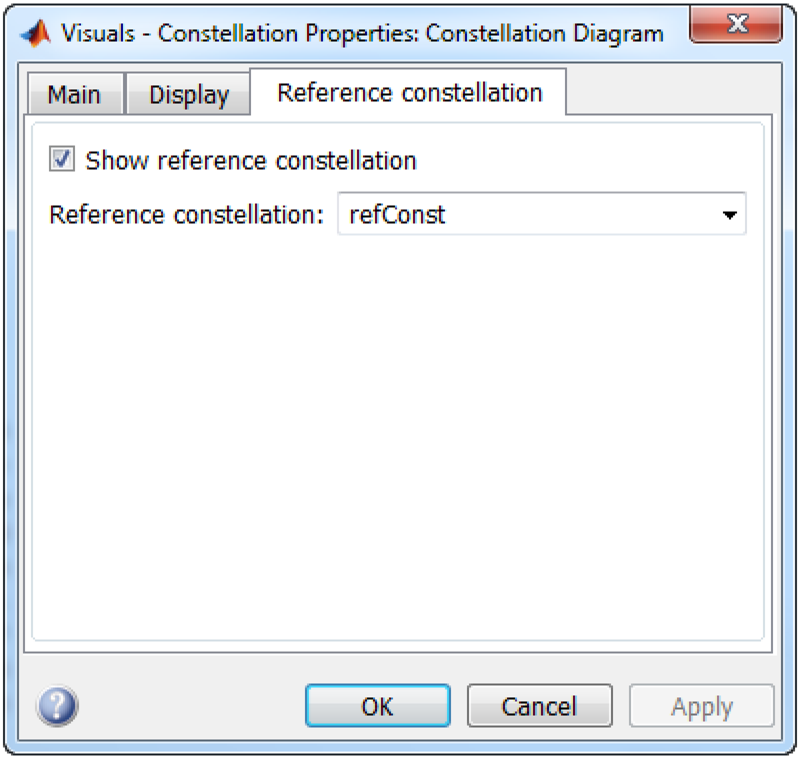
Specify the maximum value of the y-axis.

Specify a label that appears above the constellation diagram plot. By default, there is no title.

Specify the text the scope displays along the x-axis

Specify the text the scope displays along the y-axis

Reference Constellation Pane



☑ Show reference constellation

Reference constellation: refConst

Select the check box to display the reference constellation.

Select the reference constellation from `BPSK` | `QPSK` | `8-PSK` | `16-QAM` | `64-QAM` | `256-QAM` | `<user-defined>`. If not selected, the reference constellation is specified in the variable `refConst`.

Select the type of constellation normalization as `Minimum distance`, `Average power`, or `Peak power`.

Specify the minimum distance between symbols in the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Minimum distance`.

Specify the average power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Average power`.

Specify the peak power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to `Peak power`.

Specify the phase offset of the reference constellation in radians as a real scalar.

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.

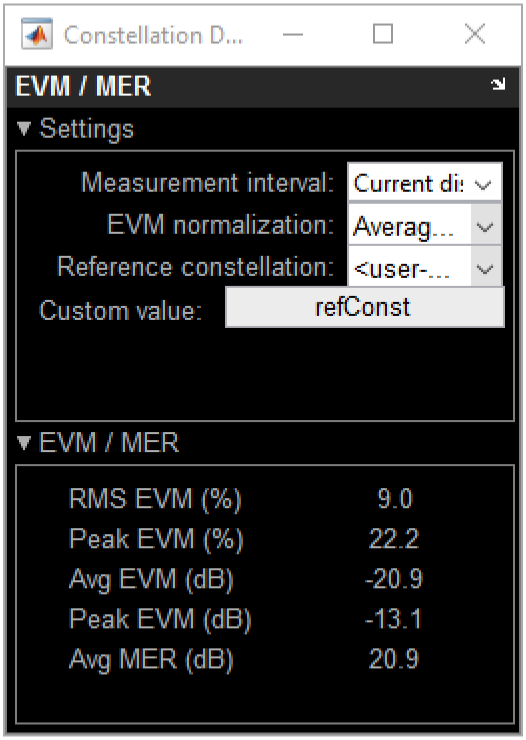
Button	Description
⤒	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
▼	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide 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 controls the Settings and Signal Quality panes. Both panels can be independently expanded or collapsed.



Measurement interval: Current di:
EVM normalization: Averag...
Reference constellation: <user-...
Custom value: refConst

RMS EVM (%)	9.0
Peak EVM (%)	22.2
Avg EVM (dB)	-20.9
Peak EVM (dB)	-13.1
Avg MER (dB)	20.9

You can choose to hide or display the **Signal Quality** panel. In the Scope menu, select **Tools** > **Measurements** > **Signal Quality**.

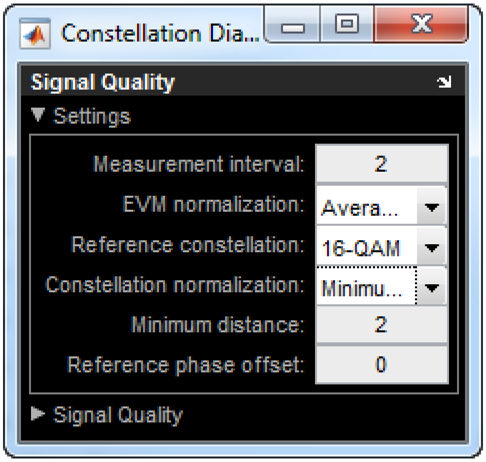
Settings Pane

The **Settings** pane enables you to define the measurement interval and normalization method the scope uses when obtaining signal measurements.

- **Measurement interval** — The duration of the EVM or MER measurement, specified as `'Current Display'`, `'All displays'`, or as a positive integer. The value of this property must be greater than one and less than or equal to the setting for the number of symbols to display. The measurement is computed after the number of input data samples exceeds the measurement interval.
- **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.

- **Reference constellation** — Select the reference constellation as BPSK | QPSK | 8-PSK | 16-QAM | 64-QAM | 256-QAM | <user-defined>.
- **Constellation normalization** — Select the type of constellation normalization as Minimum distance, Average power, or Peak power.
- **Minimum distance** — Specify the minimum distance between symbols in the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to Minimum distance.
- **Average reference power** — Specify the average power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to Average power.
- **Peak reference power** — Specify the peak power of the reference constellation as a positive real scalar. This parameter is available when **Constellation normalization** is set to Peak power.
- **Reference phase offset** — Specify the phase offset of the reference constellation in radians as a real scalar.



Measurement interval: 2
EVM normalization: Avera...
Reference constellation: 16-QAM
Constellation normalization: Minimu...
Minimum distance: 2
Reference phase offset: 0

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. The root mean square error vector magnitude, EVM_{RMS}, is measured for the average and peak constellation power.

  On the constellation diagram, you can display the EVM_{RMS} measurements normalized by the `Average constellation power` or `Peak constellation power` as computed using these algorithms.

EVM Normalization Method	Algorithm
`Average constellation power`	EVM_{RMS} in percent for average constellation power normalization $$EVM_{RMS}(percent) = \sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{avg}}} * 100$$
`Peak constellation power`	EVM_{RMS} in percent for peak constellation power normalization $$EVM_{RMS}(percent) = \sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\max}}} *100$$

  The display shows the average and peak EVM_{RMS} in percent and in decibels. The EVM reported in decibels is computed as EVM (dB) = 10*log10(EVM_{MS}) = 20*log10(EVM_{RMS})

  Where:

  - $$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$
  - I_k = In-phase measurement of the k$^{\mathrm{th}}$ symbol in the burst
  - Q_{k} = Quadrature phase measurement of the k$^{\mathrm{th}}$ 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.
- EVM_{RMS} = sqrt(EVM_{MS})

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.

EVM Normalization	Algorithm
`Average constellation power`	Average constellation power normalization $EVM_k = \sqrt{\frac{e_k}{P_{avg}}} * 100$
`Peak constellation power`	Peak constellation power normalization $EVM_k = \sqrt{\frac{e_k}{P_{\max}}} * 100$

For more information, see `comm.EVM`.

- **MER** — MER is the ratio of the average power of the transmitted signal to the average power of the error vector. The scope indicates the measurement result in decibels.

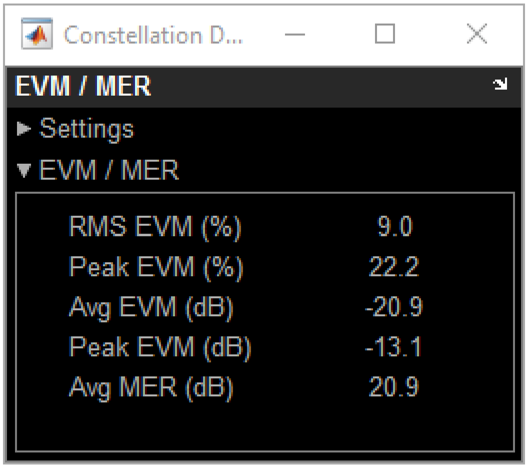
  MER is a measure of the SNR in a modulated signal calculated in dB. The MER over N symbols is

$$MER = 10 * \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}\left(e_k\right)}\right) \text{dB.}$$

  where:

- $$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$
- I_k = In-phase measurement of the *k*th symbol in the burst
- Q_k = Quadrature phase measurement of the *k*th symbol in the burst
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

For more information, see `comm.MER`.



EVM / MER

▶ Settings

▼ EVM / MER

RMS EVM (%)	9.0
Peak EVM (%)	22.2
Avg EVM (dB)	-20.9
Peak EVM (dB)	-13.1
Avg MER (dB)	20.9

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.

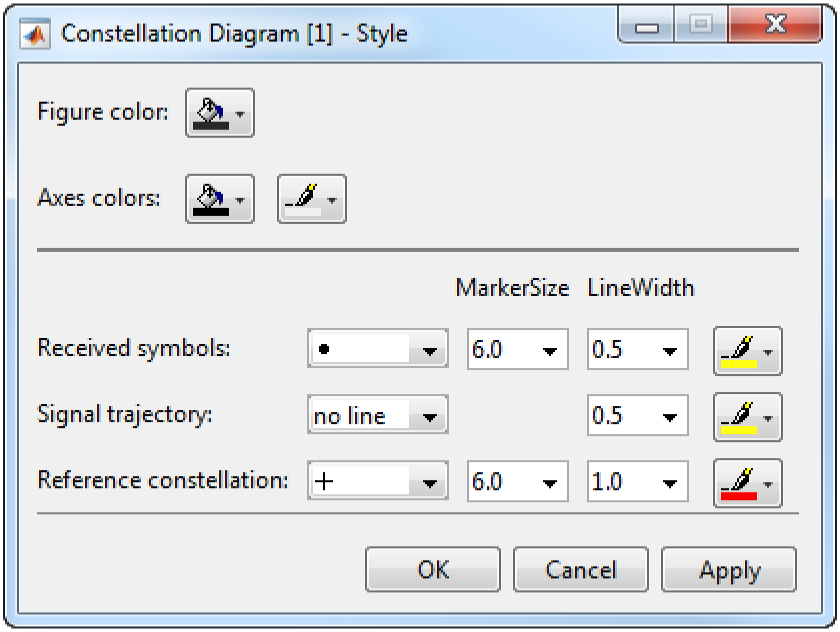


Figure color:

Axes colors:

		MarkerSize	LineWidth
Received symbols:	●	6.0	0.5
Signal trajectory:	no line		0.5
Reference constellation:	+	6.0	1.0

Properties

The **Style** dialog box allows you to modify the following elements of the scope figure:

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Specify the color that you want to apply to the background of the axes for the active display. Using a second drop down, you can also specify the color of the ticks, labels, and grid lines.

Specify the marker shape, marker size, marker line width, and color for the signal on the active display. The marker shape cannot be set to `none` unless the `ShowTrajectory` property is `true`.

Specify the line type, width, and color for the signal trajectory plot. The line type can only be set to something other than `no line` when the `ShowTrajectory` property is `true`.

Conversely, the line type must be `no line` when the `ShowTrajectory` property is `false`.

Specify the marker shape, marker size, marker line width, and color for the reference constellation shown on the active display. These settings are only applicable when the `ShowReferenceConstellation` property is `true`.

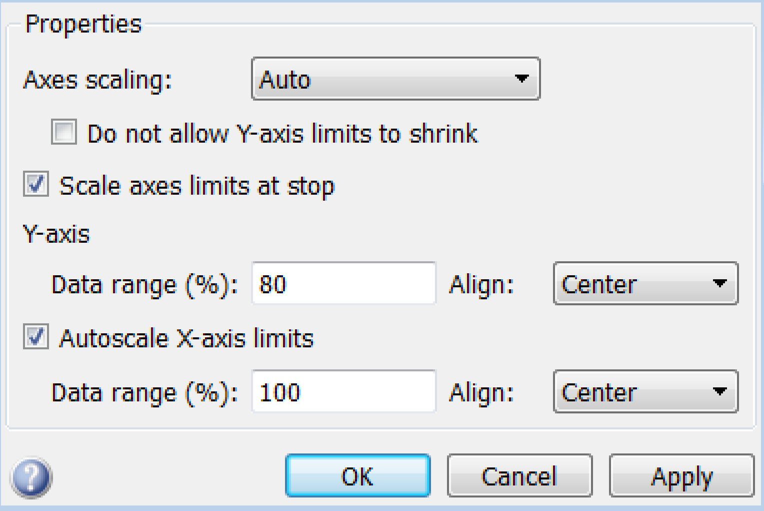
Specify the markers for the selected signal and the reference constellation on the active display. This parameter is similar to the `Marker` property for the MATLAB Handle Graphics plot objects.

Specifier	Marker Type
none	No marker
○	Circle
□	Square
×	Cross
•	Point (default)
+	Plus sign
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Tools: Plot Navigation Properties

Properties

The Tools—Axes Scaling Properties dialog box appears as follows.



Properties

Axes scaling: Auto

□ Do not allow Y-axis limits to shrink

☑ Scale axes limits at stop

Y-axis

Data range (%): 80 Align: Center

☑ Autoscale X-axis limits

Data range (%): 100 Align: Center

Specify when the scope automatically scales 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** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - 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. This option is useful and more efficient when your scope

display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire *x*-axis range. If you then set this property 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** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Programmatic Configuration

You can programmatically configure the scope properties with callbacks or within scripts by using a scope configuration object as describe in “Control Scopes Programmatically” (Simulink).

Tips

Use `comm.ConstellationDiagram` when these are required:

- Measurements
- Basic reference constellations
- Signal trajectory plots
- Maintaining state between calls

Use `scatterplot` when:

- A simple snapshot of the signal constellation is needed.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Blocks
Constellation Diagram

System Objects
`comm.EyeDiagram`

Functions
`scatterplot`

Introduced in R2013a

hide

System object: comm.ConstellationDiagram
Package: comm

Hide scope window

Syntax

```
hide(H)
```

Description

`hide(H)` hides the scope window associated with System object, `H`.

See Also

`comm.ConstellationDiagram.show`

reset

System object: comm.ConstellationDiagram
Package: comm

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`

show

System object: comm.ConstellationDiagram
Package: comm

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`

step

System object: comm.ConstellationDiagram
Package: comm

Display constellation diagram of signal in scope figure

Syntax

```
step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(H,X)` displays the constellation diagram of the signal, `X`, in the scope figure.

Examples

Constellation Diagram of QPSK Signal

Create a constellation diagram System object.

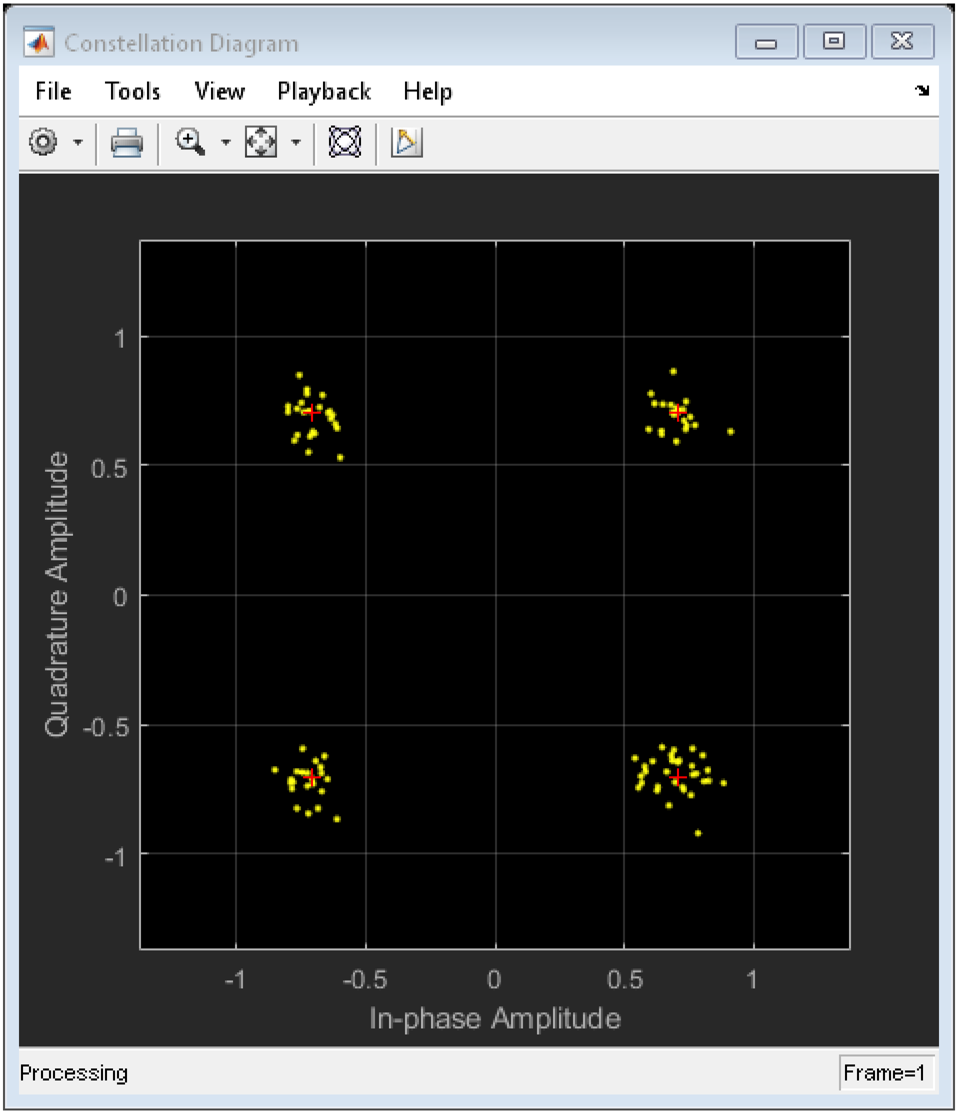
```
cd = comm.ConstellationDiagram;
```

Generate random symbols, apply QPSK modulation, and pass the modulated signal through a noisy channel.

```
d = randi([0 3],100,1);
x = pskmod(d,4,pi/4);
y = awgn(x,20);
```

Plot the constellation diagram by using the `step` method.

```
step(cd,y)
```



comm.ConvolutionalDeinterleaver System object

Package: comm

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.

To recover convolutionally interleaved binary data:

1. Define and set up your convolutional deinterleaver object. See “Construction” on page 3-334.
2. Call `step` to convolutionally deinterleave according to the properties of `comm.ConvolutionalDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 last shift register, which has zero delay. If you set this property to a scalar, then all shift registers, except the last 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 *(N–i+1)*-th element of the specified vector. The value of the first element of this property is unimportant because the last shift register has zero delay. The default is `0`.

Methods

reset	Reset states of the convolutional deinterleaver object
step	Restore ordering of symbols using shift registers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans = 21×3

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

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
```

```
intrlvDelay = 6
```

```
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
numSymErrors = 0
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.ConvolutionalInterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: comm.ConvolutionalDeinterleaver
Package: comm

Reset states of the convolutional deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalDeinterleaver` object, `H`.

step

System object: comm.ConvolutionalDeinterleaver
Package: comm

Restore ordering of symbols using shift registers

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ConvolutionalEncoder System object

Package: comm

Convolutionally encode binary data

Description

The `ConvolutionalEncoder` object encodes a sequence of binary input vectors to produce a sequence of binary output vectors.

To convolutionally encode a binary signal:

1. Define and set up your convolutional encoder object. See “Construction” on page 3-341.
2. Call `step` to encode a sequence of binary input vectors to produce a sequence of binary output vectors according to the properties of `comm.ConvolutionalEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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` on page 3-0 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 on page 3-0 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 on page 3-0 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 on page 3-0 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 on page 3-0 property. This property applies when you set the TerminationMethod on page 3-0 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 1s and 0s, where the 0 indicates the punctured, or excluded, bits. This property applies when you set the TerminationMethod on page 3-0 property to Continuous or Truncated and the PuncturePatternSource on page 3-0 property to Property.

Methods

reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Common to All System Objects	
release	Allow System object property value changes

Examples

Encode and Decode 8-DPSK Modulated Data

Transmit a convolutionally encoded 8-DPSK modulated bit stream through an AWGN channel. Then, demodulate and decode using a Viterbi decoder.

Create the necessary System objects.

```
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);
```

Process the data using the following steps:

1. Generate random bits
2. Convolutionally encode the data
3. Apply DPSK modulation
4. Pass the modulated signal through AWGN
5. Demodulate the noisy signal
6. Decode the data using a Viterbi algorithm
7. Collect error statistics

```
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
```

Display the number of errors.

```
errors(2)
```

```
ans = 3
```

Convolutional Encoding and Viterbi Decoding with a Puncture Pattern Matrix

Encode and decode a sequence of bits using a convolutional encoder and a Viterbi decoder with a defined puncture pattern. Verify that the input and output bits are identical

Define a puncture pattern matrix and reshape it into vector form for use with the Encoder and Decoder objects.

```
pPatternMat = [1 0 1;1 1 0];
pPatternVec = reshape(pPatternMat,6,1);
```

Create convolutional encoder and a Viterbi decoder in which the puncture pattern is defined by `pPatternVec`.

```
ENC = comm.ConvolutionalEncoder(...
    'PuncturePatternSource','Property', ...
    'PuncturePattern',pPatternVec);

DEC = comm.ViterbiDecoder('InputFormat','Hard', ...
    'PuncturePatternSource','Property',...
    'PuncturePattern',pPatternVec);
```

Create an error rate counter with the appropriate receive delay.

```
ERR = comm.ErrorRate('ReceiveDelay',DEC.TracebackDepth);
```

Encode and decode a sequence of random bits.

```
dataIn = randi([0 1],600,1);
```

```
dataEncoded = step(ENC,dataIn);
```

```
dataOut = step(DEC,dataEncoded);
```

Verify that there are no errors in the output data.

```
errStats = step(ERR,dataIn,dataOut);
errStats(2)
```

```
ans = 0
```

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` on page 3-0 property.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.APPDecoder` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: comm.ConvolutionalEncoder
Package: comm

Reset states of the convolutional encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalEncoder` object, `H`.

step

System object: comm.ConvolutionalEncoder
Package: comm

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

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ConvolutionalInterleaver System object

Package: comm

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.

To convolutionally interleave binary data:

1. Define and set up your convolutional interleaver object. See "Construction" on page 3-351.
2. Call `step` to convolutionally interleave according to the properties of `comm.ConvolutionalInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans = 21×3

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

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
```

```
intrlvDelay = 6
```

```
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
numSymErrors = 0
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: comm.ConvolutionalInterleaver
Package: comm

Reset states of the convolutional interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalInterleaver` object, `H`.

step

System object: comm.ConvolutionalInterleaver
Package: comm

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CPFSKDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using the continuous phase frequency shift keying method:

1. Define and set up your CPFSK demodulator object. See “Construction” on page 3-358 .
2. Call `step` to demodulate the signal according to the properties of `comm.CPFSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 and with elements that are integers between –(ModulationOrder on page 3-0 –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*× (*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 on page 3-0 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` on page 3-0 –1) to a *P*-length bit word, where *P* = `ModulationOrder` on page 3-0 (`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` on page 3-0 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, …. $h_{H\text{-}1}$]

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 on page 3-0 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

reset	Reset states of CPFSK demodulator object
step	Demodulate using CPFSK method and Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate a signal using CPFSK modulation with Gray mapping

```
% Create a CPFSK modulator, an AWGN channel, and a CPFSK demodulator
    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 Viterbi algorit
    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))
```

```
Error rate = 0.004006
Number of errors = 120
```

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. For CPFSK the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.CPMDemodulator` | `comm.CPMModulator`

Introduced in R2012a

reset

System object: comm.CPFSKDemodulator
Package: comm

Reset states of CPFSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPFSKDemodulator` object, `H`.

step

System object: comm.CPFSKDemodulator
Package: comm

Demodulate using CPFSK method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CPFSKModulator System object

Package: comm

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.

To modulate a signal using the continuous phase frequency shift keying method:

1. Define and set up your CPFSK modulator object. See “Construction” on page 3-365.
2. Call `step` to modulate the signal according to the properties of `comm.CPFSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 –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 on page 3-0 property. The object then maps the integer *K* to the intermediate value 2*K*–(ModulationOrder–1) and proceeds as in the case when you set the BitInput on page 3-0 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 on page 3-0), 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 on page 3-0 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. The phase shift over a symbol is π × *h*.

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

reset	Reset states of CPFSK modulator object
step	Modulate using CPFSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate a signal using CPFSK modulation with Gray mapping

```
% Create a CPFSK modulator, an AWGN channel, and a CPFSK demodulator
    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 Viterbi algorit
    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))
```

```
Error rate = 0.004006
Number of errors = 120
```

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. For CPFSK the phase shift per symbol is $\pi \times h$, where h is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPMDemodulator` | `comm.CPMModulator`

Introduced in R2012a

reset

System object: comm.CPFSKModulator
Package: comm

Reset states of CPFSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPFSKModulator` object, `H`.

step

System object: comm.CPFSKModulator
Package: comm

Modulate using CPFSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CPMCarrierPhaseSynchronizer System object

Package: comm

(To be removed) Recover carrier phase of baseband CPM signal

Note `comm.CPMCarrierPhaseSynchronizer` will be removed in a future release. Use `comm.CarrierSynchronizer` instead.

Description

The `CPMCarrierPhaseSynchronizer` object recovers the carrier phase of the input signal using the 2*P*-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).

To recover the carrier phase of the input signal:

1. Define and set up your CPM carrier phase synchronizer object. See “Construction” on page 3-373.
2. Call `step` to recover the carrier phase of the input signal using the 2*P*-Power method according to the properties of `comm.CPMCarrierPhaseSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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

reset	Reset states of the CPM carrier phase synchronizer object
step	Recover carrier phase of baseband CPM signal

Common to All System Objects	
release	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMModulator` | `comm.CarrierSynchronizer`

Introduced in R2012a

reset

System object: comm.CPMCarrierPhaseSynchronizer
Package: comm

Reset states of the CPM carrier phase synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMCarrierPhaseSynchronizer` object, `H`.

step

System object: comm.CPMCarrierPhaseSynchronizer
Package: comm

Recover carrier phase of baseband CPM signal

Syntax

```
[Y,PH] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To demodulate a signal that was modulated using continuous phase modulation:

1. Define and set up your CPM demodulator object. See "Construction" on page 3-378.
2. Call `step` to demodulate a signal according to the properties of `comm.CPMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 and with elements that are integers between -(`ModulationOrder` on page 3-0 -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*× (*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` on page 3-0 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` on page 3-0 –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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 –1) and (`ModulationOrder`–1). If the value is a vector, then its length must be one less than the value in the `PulseLength` on page 3-0 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 on page 3-0 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

reset	Reset states of CPM demodulator object
step	Demodulate using CPM method and Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate a CPM signal with Gray mapping and bit inputs

```
% Create a CPM modulator, an AWGN channel, and a CPM demodulator.
    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 Viterbi algorith
    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))

Error rate = 0.004006
Number of errors = 120
```

Apply GFSK Modulation and Demodulation

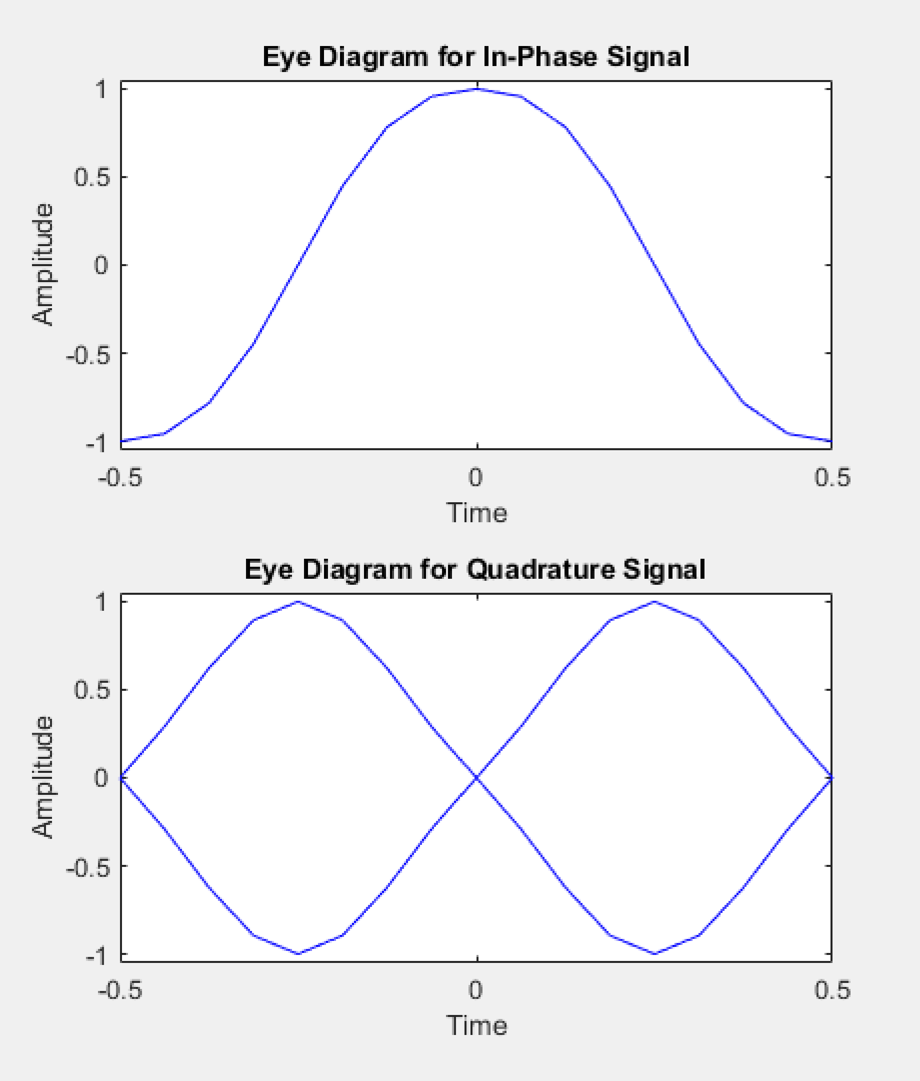
Using the `comm.CPMModulator` and `comm.CPMDemodulator` System objects apply GFSK modulation and demodulation to random bit data.

Create a GFSK modulator and demodulator object pair.

```
gfskMod = comm.CPMModulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitInput', true);
gfskDemod = comm.CPMDemodulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ..
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitOutput', true);
```

Generate random bit data and apply GFSK modulation. Use a scatter plot to view the constellation.

```
numSym = 100;
x = randi([0 1],numSym*gfskMod.SamplesPerSymbol,1);
y = gfskMod(x);
eyediagram(y,16)
```



Demodulate the GFSK modulated data. To verify that the demodulated signal data is equal to the original data, you must account for the delay introduced by the Gaussian filtering in the GFSK modulation and demodulation processing.

```
z = gfskDemod(y);
delay = finddelay(x,z);
isequal(x(1:end-delay),z(delay+1:end))

ans = logical
   1
```

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. For CPM the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPMModulator` | `comm.GMSKDemodulator` | `comm.MSKDemodulator`

Introduced in R2012a

reset

System object: comm.CPMDemodulator
Package: comm

Reset states of CPM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMDemodulator` object, `H`.

step

System object: comm.CPMDemodulator
Package: comm

Demodulate using CPM method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CPMModulator System object

Package: comm

Modulate using CPM method

Description

The `CPMModulator` object modulates using continuous phase modulation. The output is a baseband representation of the modulated signal.

To modulate a signal using continuous phase modulation:

1. Define and set up your CPM modulator object. See “Construction” on page 3-388.
2. Call `step` to modulate a signal according to the properties of `comm.CPMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 –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` on page 3-0 property. The object then maps the integer *K* to the intermediate value 2*K*-(`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` on page 3-0), 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` on page 3-0 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, …. $h_{\text{H-1}}$]

where H-1 represents the length of the column vector. The phase shift over a symbol is π × *h*.

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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 –1) and (`ModulationOrder`–1). If the value is a vector, then its length must be one less than the value in the `PulseLength` on page 3-0 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

reset	Reset states of CPM modulator object
step	Modulate using CPM method

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate a CPM signal with Gray mapping and bit inputs

```
% Create a CPM modulator, an AWGN channel, and a CPM demodulator.
    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 Viterbi algorith
    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))
```

```
Error rate = 0.004006
Number of errors = 120
```

Apply GFSK Modulation and Demodulation

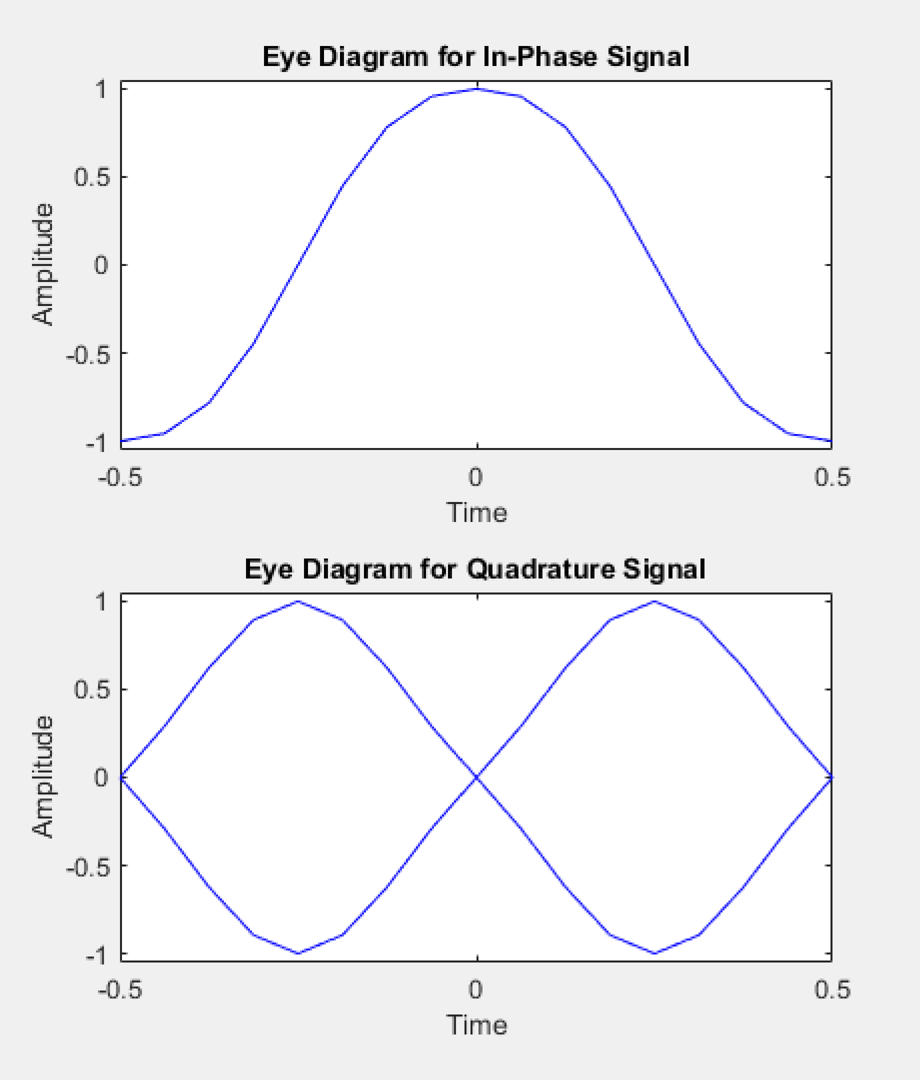
Using the `comm.CPMModulator` and `comm.CPMDemodulator` System objects apply GFSK modulation and demodulation to random bit data.

Create a GFSK modulator and demodulator object pair.

```
gfskMod = comm.CPMModulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitInput', true);
gfskDemod = comm.CPMDemodulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ..
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitOutput', true);
```

Generate random bit data and apply GFSK modulation. Use a scatter plot to view the constellation.

```
numSym = 100;
x = randi([0 1],numSym*gfskMod.SamplesPerSymbol,1);
y = gfskMod(x);
eyediagram(y,16)
```



Demodulate the GFSK modulated data. To verify that the demodulated signal data is equal to the original data, you must account for the delay introduced by the Gaussian filtering in the GFSK modulation and demodulation processing.

```
z = gfskDemod(y);
delay = finddelay(x,z);
isequal(x(1:end-delay),z(delay+1:end))
```

```
ans = logical
   1
```

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. For CPM the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.CPMDemodulator` | `comm.GMSKModulator` | `comm.MSKModulator`

Introduced in R2012a

reset

System object: comm.CPMModulator
Package: comm

Reset states of CPM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMModulator` object, `H`.

step

System object: comm.CPMModulator
Package: comm

Modulate using CPM method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Detect errors in input data using CRC

Description

The `CRCDetector` object computes checksums for its entire input frame.

To detect errors in the input data using cyclic redundancy code:

1. Define and set up your CRC detector object. See “Construction” on page 3-398.
2. Call `step` to detect errors according to the properties of `comm.CRCDetector`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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, or as a polynomial character vector. The default is `'z^16 + z^12 + z^5 + 1'`. 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	`'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'`
CRC-24	`'z^24 + z^23 + z^14 + z^12 + z^8 + 1'`
CRC-16	`'z^16 + z^15 + z^2 + 1'`
Reversed CRC-16	`'z^16 + z^14 + z + 1'`
CRC-8	`'z^8 + z^7 + z^6 + z^4 + z^2 + 1'`
CRC-4	`'z^4 + z^3 + z^2 + z + 1'`

`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` on page 3-0 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` on page 3-0 property value minus the degree of the generator polynomial, which you specify in the `Polynomial` on page 3-0 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` on page 3-0 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 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` on page 3-0 property to `0` and 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

reset	Reset states of CRC detector object
step	Detect errors in input data using CRC

Common to All System Objects	
`release`	Allow System object property value changes

Examples

CRC Detection of Errors in a Random Message

Pass binary data through a CRC generator, introduce a bit error, and detect the error using a CRC detector.

Create a random binary vector.

```
msg = randi([0 1],12,1);
```

Encode the message words using a CRC generator with the `ChecksumsPerFrame` property set to `2`. This subdivides the incoming message into two equal length subframes.

```
gen = comm.CRCGenerator([1 0 0 1],'ChecksumsPerFrame',2);
codeword = step(gen,msg);
```

Decode the message and verify that there are no errors in either subframe.

```
detect = comm.CRCDetector([1 0 0 1],'ChecksumsPerFrame',2);
[~, err] = step(detect,codeword)
```

```
err = 2×1

     0
     0
```

Introduce an error in the second subframe by inverting the last element of subframe 2. Pass the corrupted codeword through the CRC detector and verify that the error is detected in the second subframe.

```
codeword(end) = not(codeword(end));
[~,err] = step(detect,codeword)
```

```
err = 2×1

     0
     1
```

Cyclic Redundancy Check of Noisy BPSK Data Frames

Use a CRC code to detect frame errors in a noisy BPSK signal.

Create a CRC generator and detector pair using a standard CRC-4 polynomial, $z^4+z^3+z^2+z+1$.

```
crcGen = comm.CRCGenerator('z4+z3+z2+z+1');
crcDet = comm.CRCDetector('z4+z3+z2+z+1');
```

Generate 12-bit frames of binary data and append CRC bits. Based on the degree of the polynomial, 4 bits are appended to each frame. Apply BPSK modulation and pass the signal through an AWGN channel. Demodulate and use the CRC detector to determine if the frame is in error.

```
numFrames = 20;
frmError = zeros(numFrames,1);

for k = 1:numFrames
    data = randi([0 1],12,1);                    % Generate binary data
    encData = step(crcGen,data);                 % Append CRC bits
    modData = pskmod(encData,2);                 % BPSK modulate
    rxSig = awgn(modData,5);                     % AWGN channel, SNR = 5 dB
    demodData = pskdemod(rxSig,2);               % BPSK demodulate
    [~,frmError(k)] = step(crcDet,demodData);   % Detect CRC errors
end
```

Identify the frames in which bit errors are detected.

```
find(frmError)
```

```
ans = 6
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CRCGenerator`

reset

System object: comm.CRCDetector
Package: comm

Reset states of CRC detector object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CRCDetector` object, `H`.

step

System object: comm.CRCDetector
Package: comm

Detect errors in input data using CRC

Syntax

`[Y,ERR] = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 $\text{length}(\mathtt{X}) - P \times \mathtt{CheckSumsPerFrame}$, where P is the order of the polynomial that you specify in the `Polynomial` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.CRCGenerator System object

Package: comm

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.

To generate cyclic redundancy code bits and append them to the input data:

1. Define and set up your CRC generator object. See “Construction” on page 3-408.
2. Call `step` to generate cyclic redundancy code (CRC) bits for each input data frame according to the properties of `comm.CRCDetector`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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, or as a polynomial character vector. The default is `'z^16 + z^12 + z^5 + 1'`. 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	`'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'`
CRC-24	`'z^24 + z^23 + z^14 + z^12 + z^8 + 1'`
CRC-16	`'z^16 + z^15 + z^2 + 1'`
Reversed CRC-16	`'z^16 + z^14 + z + 1'`
CRC-8	`'z^8 + z^7 + z^6 + z^4 + z^2 + 1'`
CRC-4	`'z^4 + z^3 + z^2 + z + 1'`

`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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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

reset	Reset states of CRC generator object
step	Generate CRC code bits and append to input data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

CRC Detection of Errors in a Random Message

Pass binary data through a CRC generator, introduce a bit error, and detect the error using a CRC detector.

Create a random binary vector.

```
msg = randi([0 1],12,1);
```

Encode the message words using a CRC generator with the `ChecksumsPerFrame` property set to `2`. This subdivides the incoming message into two equal length subframes.

```
gen = comm.CRCGenerator([1 0 0 1],'ChecksumsPerFrame',2);
codeword = step(gen,msg);
```

Decode the message and verify that there are no errors in either subframe.

```
detect = comm.CRCDetector([1 0 0 1],'ChecksumsPerFrame',2);
[~, err] = step(detect,codeword)
```

```
err = 2×1

     0
     0
```

Introduce an error in the second subframe by inverting the last element of subframe 2. Pass the corrupted codeword through the CRC detector and verify that the error is detected in the second subframe.

```
codeword(end) = not(codeword(end));
[~,err] = step(detect,codeword)
```

```
err = 2×1

     0
     1
```

Cyclic Redundancy Check of Noisy BPSK Data Frames

Use a CRC code to detect frame errors in a noisy BPSK signal.

Create a CRC generator and detector pair using a standard CRC-4 polynomial, $z^4 + z^3 + z^2 + z + 1$.

```
crcGen = comm.CRCGenerator('z4+z3+z2+z+1');
crcDet = comm.CRCDetector('z4+z3+z2+z+1');
```

Generate 12-bit frames of binary data and append CRC bits. Based on the degree of the polynomial, 4 bits are appended to each frame. Apply BPSK modulation and pass the signal through an AWGN channel. Demodulate and use the CRC detector to determine if the frame is in error.

```
numFrames = 20;
frmError = zeros(numFrames,1);

for k = 1:numFrames
    data = randi([0 1],12,1);                   % Generate binary data
    encData = step(crcGen,data);                % Append CRC bits
    modData = pskmod(encData,2);                % BPSK modulate
    rxSig = awgn(modData,5);                    % AWGN channel, SNR = 5 dB
    demodData = pskdemod(rxSig,2);              % BPSK demodulate
    [~,frmError(k)] = step(crcDet,demodData);  % Detect CRC errors
end
```

Identify the frames in which bit errors are detected.

```
find(frmError)
```

```
ans = 6
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CRCDetector`

reset

System object: comm.CRCGenerator
Package: comm

Reset states of CRC generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CRCGenerator` object, `H`.

step

System object: comm.CRCGenerator
Package: comm

Generate CRC code bits and append to input data

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DBPSKDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using differential binary phase shift keying:

1. Define and set up your DBPSK demodulator object. See "Construction" on page 3-417.
2. Call `step` to demodulate a signal according to the properties of `comm.DBPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of DBPSK demodulator object
step	Demodulate using DBPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

DBPSK Signal in AWGN

Create a DBPSK modulator and demodulator pair.

```
dbpskmod = comm.DBPSKModulator(pi/4);
dpbpskdemod = comm.DBPSKDemodulator(pi/4);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50-bit frames
- DBPSK modulate
- Pass through AWGN channel
- DBPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],50,1);
    modSig = dbpskmod(txData);
    rxSig = awgn(modSig,7);
    rxData = dpbpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0040
```

```
numErrors = errorStats(2)
```

```
numErrors = 20
```

```
numBits = errorStats(3)
```

```
numBits = 4999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: comm.DBPSKDemodulator
Package: comm

Reset states of DBPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DBPSKDemodulator` object, `H`.

step

System object: comm.DBPSKDemodulator
Package: comm

Demodulate using DBPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DBPSKModulator System object

Package: comm

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.

To modulate a signal using differential binary phase shift keying:

1. Define and set up your DBPSK modulator object. See “Construction” on page 3-423.
2. Call `step` to modulate a signal according to the properties of `comm.DBPSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of DBPSK modulator object
step	Modulate using DBPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

DBPSK Signal in AWGN

Create a DBPSK modulator and demodulator pair.

```
dbpskmod = comm.DBPSKModulator(pi/4);
dpbpskdemod = comm.DBPSKDemodulator(pi/4);
```

Create an error rate calculator. Set the ComputationDelay property to 1 to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50-bit frames
- DBPSK modulate
- Pass through AWGN channel
- DBPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],50,1);
    modSig = dbpskmod(txData);
    rxSig = awgn(modSig,7);
    rxData = dpbpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0040
```

```
numErrors = errorStats(2)
```

```
numErrors = 20
```

```
numBits = errorStats(3)
```

```
numBits = 4999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: comm.DBPSKModulator
Package: comm

Reset states of DBPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DBPSKModulator` object, `H`.

step

System object: comm.DBPSKModulator
Package: comm

Modulate using DBPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

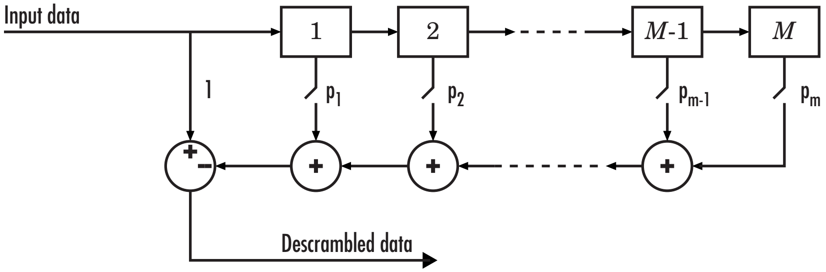
Package: comm

Descramble input signal

Description

The comm.Descrambler object descrambles a scalar or column vector input signal. The comm.Descrambler object is the inverse of the `comm.Scrambler` object. If you use the `comm.Scrambler` object in a transmitter, then you use the comm.Descrambler object in the related receiver.

This schematic shows the descrambler operation. The adders and subtracter operate modulo *N*, where *N* is the value specified by the Calculation base property.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Polynomial property, you specify the on or off state for each switch in the descrambler. To make the `comm.Descrambler` object reverse the operation of the `comm.Scrambler` object, use the same property settings in both objects. If there is no signal delay between the scrambler and the descrambler, then the InitialConditions in the two objects must be the same.

To descramble an input signal:

1. Create the comm.Descrambler object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
descrambler = comm.Descrambler
descrambler = comm.Descrambler(base,poly,cond)
descrambler = comm.Descrambler(___,Name,Value)
```

Description

`descrambler = comm.Descrambler` creates a descrambler System object. This object descrambles the input data by using a linear feedback shift register that you specify with the Polynomial property.

`descrambler = comm.Descrambler(base,poly,cond)` creates the descrambler object with the CalculationBase property set to `base`, the Polynomial property set to `poly`, and the InitialConditions property set to `cond`.

Example: `comm.Descrambler(8,'1 + z^-2 + z^-3 + z^-5 + z^-7',[0 3 2 2 5 1 7])` sets the calculation base to 8, and the descrambler polynomial and initial conditions as specified.

`descrambler = comm.Descrambler(___,Name,Value)` sets properties using one or more name-value pairs and either of the previous syntaxes. Enclose each property name in single quotes.

Example: `comm.Descrambler('CalculationBase',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`CalculationBase` — Range of input data

4 (default) | nonnegative integer

Range of input data used in the descrambler for modulo operations, specified as a nonnegative integer. The input and output of this object are integers from `0` to `CalculationBase` – `1`.

Data Types: `double`

`Polynomial` — Connections for linear feedback shift registers

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Connections for linear feedback shift registers in the descrambler, specified as a character vector, integer vector, or binary vector. The `Polynomial` property defines if each switch in the descrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + ...$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that have a coefficient of 1. In this case, the order of the descramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Data Types: `double` | `char`

`InitialConditionsSource` — Initial conditions source

`'Property'` (default) | `'Input port'`

- `'Property'` – Specify descrambler initial conditions by using the InitialConditions property.
- `'Input port'` – Specify descrambler initial conditions by using an additional input argument, initcond, when calling the object.

Data Types: `char`

`InitialConditions` — Initial conditions of descrambler registers

`[0 1 2 3]` (default) | nonnegative integer vector

Initial conditions of descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of `InitialConditions` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

`ResetInputPort` — Descrambler state reset port
`false` (default) | `true`

Descrambler state reset port, specified as `false` or `true`. If `ResetInputPort` is `true`, you can reset the descrambler object by using an additional input argument, reset, when calling the object.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

Usage

Syntax

```
descrambledOut = descrambler(signal)
descrambledOut = descrambler(signal,initcond)
descrambledOut = descrambler(signal,reset)
```

Description

`descrambledOut = descrambler(signal)` descrambles the input signal. The output is the same data type and length as the input vector.

`descrambledOut = descrambler(signal,initcond)`provides an additional input with values specifying the initial conditions of the linear feedback shift register.

This syntax applies when you set the InitialConditionsSource property of the object to `'Input port'`.

`descrambledOut = descrambler(signal,reset)` provides an additional input indicating whether to reset the state of the descrambler.

This syntax applies when you set the InitialConditionsSource property of the object to `'Property'` and the ResetInputPort to `true`.

Input Arguments

`signal` — Input signal

column vector

Input signal, specified as a column vector.

Example: `descrambledOut = descrambler([0 1 1 0 1 0])`

Data Types: `double` | `logical`

`initcond` — Initial register condition

nonnegative integer column vector

Initial descrambler register conditions when the simulation starts, specified as a nonnegative integer column vector. The length of `initcond` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Example: `descrambledOut = descrambler(signal,[0 1 1 0])` corresponds to possible initial register states for a descrambler with a polynomial order of 4 and a calculation base of 2 or higher.

Data Types: `double`

`reset` — Reset initial state of descrambler

scalar

Reset initial state of the descrambler when the simulation starts, specified as a scalar. When the value of `reset` is nonzero, the object is reset before it is called.

Example: `descrambledOut = descrambler(signal,0)` descrambles the input signal without resetting the descrambler states.

Data Types: `double`

Output Arguments

out — Descrambled output
column vector

Descrambled output, returned as a column vector with the same data type and length as signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Scramble and Descramble Data

Scramble and descramble 8-ary data using `comm.Scrambler` and `comm.Descrambler` System objects™ having a calculation base of 8.

Create scrambler and descrambler objects, specifying the calculation base, polynomial, and initial conditions using input arguments. The scrambler and descrambler polynomials are specified with different but equivalent data formats.

```
N = 8;
scrambler = comm.Scrambler(N,'1 + z^-2 + z^-3 + z^-5 + z^-7', ...
    [0 3 2 2 5 1 7]);
descrambler = comm.Descrambler(N,[1 0 1 1 0 1 0 1], ...
    [0 3 2 2 5 1 7]);
```

Scramble and descramble random integers. Display the original data, scrambled data, and descrambled data sequences.

```
data = randi([0 N-1],5,1);
scrData = scrambler(data);
deScrData = descrambler(scrData);
[data scrData deScrData]
```

```
ans = 5×3

     6     7     6
     7     5     7
     1     7     1
     7     0     7
     5     3     5
```

Verify that the descrambled data matches the original data.

```
isequal(data,deScrData)
```

```
ans = logical
   1
```

Scramble and Descramble Data with Changing Initial Conditions

Scramble and descramble quaternary data while changing the initial conditions between function calls.

Create scrambler and descrambler System objects having a calculation base of 4. Set the `InitialConditionsSource` property to `'Input port'` so you can set the initial conditions as an argument to the object.

```
N = 4;
scrambler = comm.Scrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
descrambler = comm.Descrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
```

Preallocate memory for the error vector which will be used to store errors output by the `symerr` function.

```
errVec = zeros(10,1);
```

Scramble and descramble random integers while changing the initial conditions, `initCond`, each time the loop executes. Use the `symerr` function to determine if the scrambling and descrambling operations result in symbol errors.

```
for k = 1:10
    initCond = randperm(3)';
    data = randi([0 N-1],5,1);
    scrData = scrambler(data,initCond);
    deScrData = descrambler(scrData,initCond);
    errVec(k) = symerr(data,deScrData);
end
```

Examine `errVec` to verify that the output from the descrambler matches the original data.

```
errVec
```

```
errVec = 10×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.PNSequence` | `comm.Scrambler`

Blocks
Descrambler

Introduced in R2012a

comm.DifferentialDecoder System object

Package: comm

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.

To decode a binary signal using differential decoding:

1. Define and set up your differential decoder object. See "Construction" on page 3-438.
2. Call `step` to decode a binary signal according to the properties of `comm.DifferentialDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of differential decoder object
step	Decode binary signal using differential decoding

Common to All System Objects	
release	Allow System object property value changes

Examples

Decode Differentially Encoded Signal

Create a differential encoder and decoder pair.

```
diffEnc = comm.DifferentialEncoder;
diffDec = comm.DifferentialDecoder;
```

Generate random binary data. Differentially encode and decode the data.

```
data = randi([0 1],100,1);
encData = diffEnc(data);
decData = diffDec(encData);
```

Determine the number of errors between the original data and the decoded data.

```
numErrors = biterr(data,decData)
```

```
numErrors = 0
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DifferentialEncoder`

Introduced in R2012a

reset

System object: comm.DifferentialDecoder
Package: comm

Reset states of differential decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DifferentialDecoder` object, `H`.

step

System object: comm.DifferentialDecoder
Package: comm

Decode binary signal using differential decoding

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DifferentialEncoder System object

Package: comm

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.

To encode a binary signal using differential coding:

1. Define and set up your differential encoder object. See “Construction” on page 3-443.
2. Call `step` to encode a binary signal according to the properties of `comm.DifferentialEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of differential encoder object
step	Encode binary signal using differential coding

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Differentially Encode Binary Data

Create a differential encoder object.

```
diffEnc = comm.DifferentialEncoder;
```

Generate random binary data. Encode the data.

```
data = randi([0 1],10,1);
encData = diffEnc(data)
```

```
encData = 10×1

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

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.DifferentialDecoder`

Introduced in R2012a

reset

System object: comm.DifferentialEncoder
Package: comm

Reset states of differential encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DifferentialEncoder` object, `H`.

step

System object: comm.DifferentialEncoder
Package: comm

Encode binary signal using differential coding

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DiscreteTimeVCO System object

Package: comm

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.

To generate a variable frequency sinusoid:

1. Define and set up your discrete time voltage-controlled oscillator object. See "Construction" on page 3-448 .
2. Call `step` to generate a variable frequency sinusoid according to the properties of `comm.DiscreteTimeVCO`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of discrete-time VCO object
step	Generate variable frequency sinusoid

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate FSK Signal Using Discrete Time VCO

Create a signal source System object™.

```
reader = dsp.SignalSource;
```

Generate random data and apply rectangular pulse shaping.

```
reader.Signal = randi([0 7],10,1);
reader.Signal = rectpulse(reader.Signal,100);
```

Create a signal logger and discrete time VCO System objects.

```
logger = dsp.SignalSink;
discreteVCO = comm.DiscreteTimeVCO('OutputAmplitude',8,'QuiescentFrequency',1);
```

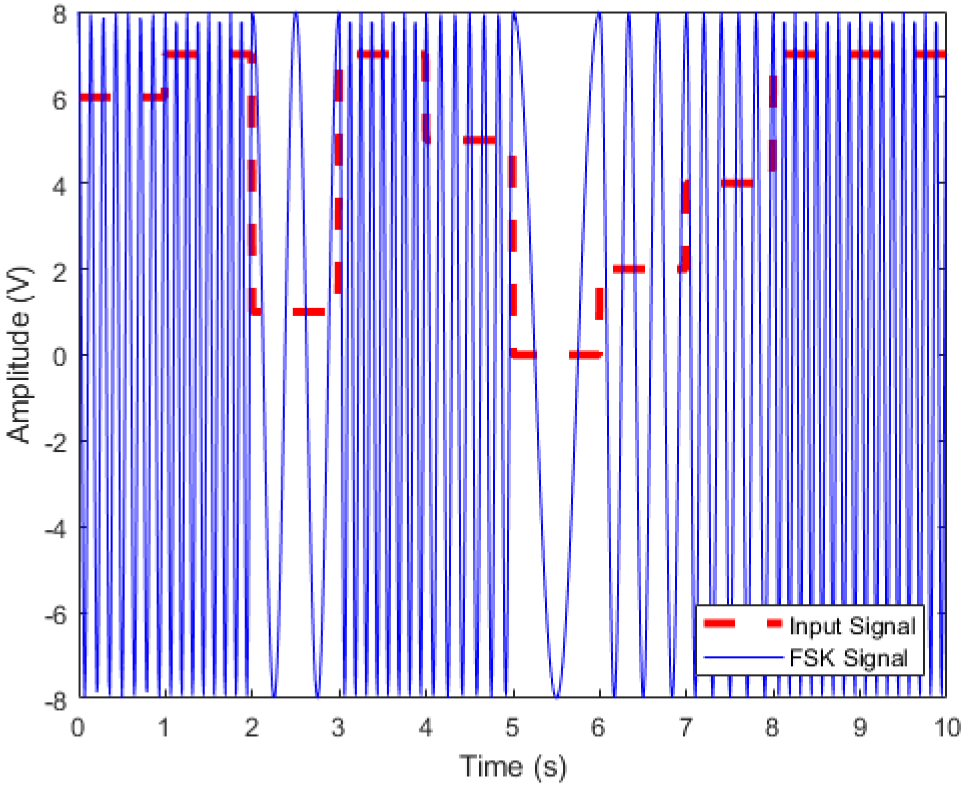
Generate an FSK signal.

```
while(~isDone(reader))
    sig = reader();
    y = discreteVCO(sig);
    logger(y);
end
oscsig = logger.Buffer;
```

Plot the generated FSK signal.

```
t = (0:length(oscsig)-1)'/discreteVCO.SampleRate;
plot(t,reader.Signal,'--r', 'LineWidth',3)
hold on
```

```
plot(t,oscsig,'-b');
hold off
xlabel('Time (s)')
ylabel('Amplitude (V)')
legend('Input Signal','FSK Signal','location','se')
```



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`)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMCarrierPhaseSynchronizer` | `comm.CarrierSynchronizer`

Introduced in R2012a

reset

System object: comm.DiscreteTimeVCO
Package: comm

Reset states of discrete-time VCO object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DiscreteTimeVCO` object, `H`.

step

System object: comm.DiscreteTimeVCO
Package: comm

Generate variable frequency sinusoid

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DPSKDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using differential phase shift keying:

1. Define and set up your DPSK modulator object. See “Construction” on page 3-455.
2. Call `step` to demodulate a signal according to the properties of `DPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0) 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 on page 3-0) 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 on page 3-0 property to true, logical data type becomes a valid option.

Methods

reset	Reset states of M-DPSK demodulator object
step	Demodulate using M-ary DPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

8-DPSK Signal in AWGN

Create a DPSK modulator and demodulator pair. Create an AWGN channel object having three bits per symbol.

```
dpskmod = comm.DPSKModulator(8,pi/8,'BitInput',true);
dpskdemod = comm.DPSKDemodulator(8,pi/8,'BitOutput',true);
channel = comm.AWGNChannel('EbNo',10,'BitsPerSymbol',3);
```

Create an error rate calculator. Set the ComputationDelay property to 1 to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 3-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],150,1);
    modSig = dpskmod(txData);
    rxSig = channel(modSig);
    rxData = dpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0098
```

```
numErrors = errorStats(2)
```

```
numErrors = 147
```

```
numBits = errorStats(3)
```

```
numBits = 14999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DPSKModulator` | `comm.DQPSKDemodulator`

Introduced in R2012a

reset

System object: comm.DPSKDemodulator
Package: comm

Reset states of M-DPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DPSKDemodulator` object, `H`.

step

System object: comm.DPSKDemodulator
Package: comm

Demodulate using M-ary DPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DPSKModulator System object

Package: comm

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.

To modulate a signal using differential phase shift keying:

1. Define and set up your DPSK modulator object. See “Construction” on page 3-462.
2. Call step to modulate a signal according to the properties of comm.DPSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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` on page 3-0). 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` on page 3-0) 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) shifts the output phase. This shift is (`PhaseRotation` on page 3-0 $+\ 2\times\pi\times m$/`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

reset	Reset states of M-DPSK modulator object
step	Modulate using M-ary DPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

8-DPSK Signal in AWGN

Create a DPSK modulator and demodulator pair. Create an AWGN channel object having three bits per symbol.

```
dpskmod = comm.DPSKModulator(8,pi/8,'BitInput',true);
dpskdemod = comm.DPSKDemodulator(8,pi/8,'BitOutput',true);
channel = comm.AWGNChannel('EbNo',10,'BitsPerSymbol',3);
```

Create an error rate calculator. Set the ComputationDelay property to 1 to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 3-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate

- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],150,1);
    modSig = dpskmod(txData);
    rxSig = channel(modSig);
    rxData = dpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0098
```

```
numErrors = errorStats(2)
```

```
numErrors = 147
```

```
numBits = errorStats(3)
```

```
numBits = 14999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: comm.DPSKModulator
Package: comm

Reset states of M-DPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DPSKModulator` object, `H`.

step

System object: comm.DPSKModulator
Package: comm

Modulate using M-ary DPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DQPSKDemodulator System object

Package: comm

Demodulate using DQPSK method

Description

The `DQPSKDemodulator` object demodulates a signal that was modulated using the differential quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using differential quadrature phase shift keying:

1. Define and set up your DQPSK modulator object. See “Construction” on page 3-469.
2. Call `step` to demodulate a signal according to the properties of `DQPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 **exp**($j\times$ `PhaseRotation` on page 3-0 $+ j\times 2\times\pi\times {}^{m}\!/_{4}$)$\times$(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` on page 3-0 property to `true`, `logical` data type becomes a valid option.

Methods

reset	Reset states of DQPSK demodulator object
step	Demodulate using DQPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

DQPSK Signal in AWGN

Create a DQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
dqpskmod = comm.DQPSKModulator('BitInput',true);
dqpskdemod = comm.DQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',6,'BitsPerSymbol',2);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 2-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate

- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],100,1);
    modSig = dqpskmod(txData);
    rxSig = channel(modSig);
    rxData = dqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0170
```

```
numErrors = errorStats(2)
```

```
numErrors = 170
```

```
numBits = errorStats(3)
```

```
numBits = 9999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: comm.DQPSKDemodulator
Package: comm

Reset states of DQPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DQPSKDemodulator` object, `H`.

step

System object: comm.DQPSKDemodulator
Package: comm

Demodulate using DQPSK method

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.DQPSKModulator System object

Package: comm

Modulate using DQPSK method

Description

The DQPSKModulator object modulates using the differential quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using differential quadrature phase shift keying:

1 Define and set up your DQPSK modulator object. See "Construction" on page 3-476.
2 Call step to modulate a signal according to the properties of comm.DQPSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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 $0 \le m \le 3$ shifts the output phase. This shift is

(`PhaseRotation` on page 3-0 $+ \; 2\times\pi\times{}^{m}\!/_{4}$) radians from the previous output phase.

The output symbol is **exp**($j\times$`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

reset	Reset states of DQPSK modulator object
step	Modulate using DQPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

DQPSK Signal in AWGN

Create a DQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
dqpskmod = comm.DQPSKModulator('BitInput',true);
dqpskdemod = comm.DQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',6,'BitsPerSymbol',2);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 2-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],100,1);
    modSig = dqpskmod(txData);
    rxSig = channel(modSig);
    rxData = dqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0170
```

```
numErrors = errorStats(2)
```

```
numErrors = 170
```

```
numBits = errorStats(3)
```

```
numBits = 9999
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DPSKModulator` | `comm.DQPSKDemodulator`

Introduced in R2012a

reset

System object: comm.DQPSKModulator
Package: comm

Reset states of DQPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DQPSKModulator` object, `H`.

step

System object: comm.DQPSKModulator
Package: comm

Modulate using DQPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.EarlyLateGateTimingSynchronizer System object

Package: comm

Recover symbol timing phase using early-late gate method

Note `comm.EarlyLateGateTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

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.

To recover the symbol timing phase of the input signal :

1. Define and set up your early late gate timing synchronizer object. See “Construction” on page 3-483.
2. Call `step` to recover the symbol timing phase of the input signal according to the properties of `comm.EarlyLateGateTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 , 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` on page 3-0 property to `false`.

Methods

reset	Reset states of early-late gate timing phase synchronizer
step	Recover symbol timing phase using early-late gate method

Common to All System Objects	
`release`	Allow System object property value changes

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;

% Create System objects
    hMod = comm.RectangularQAMModulator(M, ...
        'NormalizationMethod', 'Average power');
    hTxFilter = comm.RaisedCosineTransmitFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'OutputSamplesPerSymbol', L);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel('NoiseMethod',  ...
        'Signal to noise ratio (SNR)', 'SNR', snrdB, ...
        'SignalPower', 1/L);
    hRxFilter = comm.RaisedCosineReceiveFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'InputSamplesPerSymbol', L, ...
        'DecimationFactor', 1);
    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` on page 3-0 and `ResetCondition` on page 3-0 properties.

See Also

`comm.MSKTimingSynchronizer` | `comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: comm.EarlyLateGateTimingSynchronizer
Package: comm

Reset states of early-late gate timing phase synchronizer

Syntax

```
reset(H)
```

Note `comm.EarlyLateGateTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

`reset(H)` resets the states of early-late gate timing phase synchronizer for the `EarlyLateGateTimingSynchronizer` object `H`.

step

System object: comm.EarlyLateGateTimingSynchronizer
Package: comm

Recover symbol timing phase using early-late gate method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Note `comm.EarlyLateGateTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ErrorRate System object

Package: comm

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.

To obtain the error rate:

1. Define and set up your error rate object. See "Construction" on page 3-490.
2. Call `step` to compare input data from a transmitter with input data from a receiver and calculate the error rate according to the properties of `comm.ErrorRate`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

reset	Reset states of error rate calculator object
step	Compute bit or symbol error rate of input data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Calculate Error Statistics

Create two binary vectors and determine the error statistics.

Create a bit error rate counter object.

```
errorRate = comm.ErrorRate;
```

Create an arbitrary binary data vector.

```
x = [1 0 1 0 1 0 1 0 1 0]';
```

Introduce errors to the first and last bits.

```
y = x;
y(1) = ~y(1);
y(end) = ~y(end);
```

Calculate the error statistics.

```
z = errorRate(x,y);
```

The first element of the vector z is the bit error rate.

```
z(1)
```

```
ans = 0.2000
```

The second element of z is the total error count.

```
z(2)
```

```
ans = 2
```

The third element of z is the total number of bits.

```
z(3)
```

```
ans = 10
```

Calculate BER between Transmitted and Received Signal

Create an 8-DPSK modulator and demodulator pair that work with binary data.

```
dpskModulator = comm.DPSKModulator('ModulationOrder',8,'BitInput',true);
dpskDemodulator = comm.DPSKDemodulator('ModulationOrder',8,'BitOutput',true);
```

Create an error rate calculator, accounting for the three bit (one symbol) transient caused by the differential modulation.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Calculate the BER for 10 frames.

```
BER = zeros(10,1);

for i= 1:10
    txData = randi([0 1],96,1);           % Generate binary data
    modData = dpskModulator(txData);      % Modulate
    rxSig = awgn(modData,7);              % Pass through AWGN channel
    rxData = dpskDemodulator(rxSig);      % Demodulate
    errors = errorRate(txData,rxData);    % Compute error statistics
    BER(i) = errors(1);                   % Save BER data
end
```

Display the BER.

```
BER
```

```
BER = 10×1

    0.1613
    0.1640
    0.1614
    0.1496
    0.1488
    0.1309
    0.1405
    0.1399
    0.1370
    0.1411
```

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` on page 3-0 and `CustomSamples` on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`alignsignals` | `finddelay`

Introduced in R2012a

reset

System object: comm.ErrorRate
Package: comm

Reset states of error rate calculator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ErrorRate` object, `H`.

step

System object: comm.ErrorRate
Package: comm

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

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.EVM System object

Package: comm

Measure error vector magnitude

Description

The `comm.EVM` (error vector magnitude) System object measures the modulator or demodulator performance of an impaired signal.

To measure error vector magnitude:

1. Define and set up your EVM object. See “Construction” on page 3-499.
2. Call `step` to measure the modulator or demodulator performance according to the properties of `comm.EVM`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`EVM = comm.EVM` creates an error vector magnitude object, `EVM`. This object measures the amount of impairment in a modulated signal.

`EVM = comm.EVM(Name,Value)` creates an `EVM` object 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`).

Example: `EVM = comm.EVM('ReferenceSignalSource','Estimated from reference constellation')` creates an object, `EVM`, that measures the RMS EVM of a received signal by using a reference constellation.

Properties

Normalization

Normalization method

Normalization method used in EVM calculation, specified as one of the following: 'Average reference signal power' (default), 'Average constellation power', or 'Peak constellation power'.

AverageConstellationPower

Average constellation power

Average constellation power, specified in watts as a positive real scalar. This property is available when Normalization is 'Average constellation power'. The default is 1.

PeakConstellationPower

Peak constellation power

Peak constellation power, specified in watts as a positive real scalar. This property is available when Normalization is 'Peak constellation power'. The default is 1.

ReferenceSignalSource

Reference signal source

Reference signal source, specified as either 'Input port' (default) or 'Estimated from reference constellation'. To provide an explicit reference signal against which the input signal is measured, set this property to 'Input port'. To measure the EVM of the input signal against a reference constellation, set this property to 'Estimated from reference constellation'.

ReferenceConstellation

Reference constellation

Reference constellation, specified as a vector. This property is available when the ReferenceSignalSource property is 'Estimated from reference constellation'.

The default is [0.7071 - 0.7071i; -0.7071 - 0.7071i; -0.7071 + 0.7071i; 0.7071 + 0.7071i], which corresponds to a standard QPSK constellation. You can derive constellation points by using modulation functions or objects. For example, to derive the reference constellation for a 16-QAM signal, you can use qammod(0:15,16).

MeasurementIntervalSource

Measurement interval source

Measurement interval source, specified as one of the following: 'Input length' (default), 'Entire history', 'Custom', or 'Custom with periodic reset'. This property affects the RMS and maximum EVM outputs only.

- To calculate EVM using only the current samples, set this property to 'Input length'.
- To calculate EVM for all samples, set this property to 'Entire history'.
- To calculate EVM over an interval you specify and to use a sliding window, set this property to 'Custom'.
- To calculate EVM over an interval you specify and to reset the object each time the measurement interval is filled, set this property to 'Custom with periodic reset'.

MeasurementInterval

Measurement interval

Measurement interval over which the EVM is calculated, specified in samples as a real positive integer. This property is available when MeasurementIntervalSource is 'Custom' or 'Custom with periodic reset'. The default is 100.

AveragingDimensions

Averaging dimensions

Averaging dimensions, specified as a positive integer or row vector of positive integers. This property determines the dimensions over which the averaging is performed. For example, to average across the rows, set this property to 2. The default is 1.

The object supports variable-size inputs over the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must remain constant between step calls. For example, if the input has size [4 3 2] and Averaging dimensions is [1 3], the output size is [1 3 1], and the second dimension must remain fixed at 3.

MaximumEVMOutputPort

Maximum EVM measurement output port

Maximum EVM measurement output port, specified as a logical scalar. To create an output port for maximum EVM measurements, set this property to true. The default is false.

XPercentileEVMOutputPort

X-percentile EVM measurement output port

X-percentile EVM measurement output port, specified as a logical scalar. To create an output port for *X*-percentile EVM measurements, set this property to true. The *X*-percentile EVM measurements persist until you reset the object. These measurements are calculated by using all of the input frames since the last reset. The default is false.

XPercentileValue

X-percentile value

X-percentile value below which *X*% of the EVM measurements fall, specified as a real scalar from 0 to 100. This property is available when XPercentileEVMOutputPort is true. The default is 95.

SymbolCountOutputPort

Symbol count output port

Symbol count output port, specified as a logical scalar. To output the number of accumulated symbols used to calculate the *X*-percentile EVM measurements, set this property to true. This property is available when XPercentileEVMOutputPort is true. The default is false.

Methods

reset	Reset states of EVM measurement object
step	Measure error vector magnitude

Common to All System Objects	
release	Allow System object property value changes

Examples

Measure EVM of Noisy 16-QAM Modulated Signal

Create an EVM object. Configure it using name-value pairs to output maximum EVM, 90th percentile EVM, and the symbol count.

```
evm = comm.EVM('MaximumEVMOutputPort',true,...
    'XPercentileEVMOutputPort',true, 'XPercentileValue',90,...
    'SymbolCountOutputPort',true);
```

Generate random data symbols. Apply 16-QAM modulation. The modulated signal serves as the reference for the subsequent EVM measurements.

```
data = randi([0 15],1000,1);
refSym = qammod(data,16,'UnitAveragePower',true);
```

Pass the modulated signal through an AWGN channel.

```
rxSym = awgn(refSym,20);
```

Measure the EVM of the noisy signal.

```
[rmsEVM,maxEVM,pctEVM,numSym] = evm(refSym,rxSym)
```

```
rmsEVM = 9.8775

maxEVM = 26.8385

pctEVM = 14.9750

numSym = 1000
```

Estimate Received EVM

Generate filtered QAM data and pass it through an AWGN channel. Compute the symbol error rate, and estimate the EVM of the received signal.

Create modulation, demodulation, channel, and filter System objects™.

```
modulator = comm.RectangularQAMModulator(16);
demodulator = comm.RectangularQAMDemodulator(16);
channel = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)',...
    'SNR',15,'SignalPower',10);

txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',4);
rxfilter = comm.RaisedCosineReceiveFilter('InputSamplesPerSymbol',4, ...
    'DecimationFactor',4);
```

Create an EVM object to output RMS and maximum EVM measurements.

```
evm = comm.EVM('MaximumEVMOutputPort',true, ...
    'ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',constellation(modulator));
```

Create an error rate object and account for the signal delay through the transmit and receive filters. For a filter, the group delay is equal to 1/2 of the `FilterSpanInSymbols` property.

```
rxd = (txfilter.FilterSpanInSymbols + rxfilter.FilterSpanInSymbols)/2;
errorRate = comm.ErrorRate('ReceiveDelay',rxd);
```

Perform the following channel operations:

- Generate random data symbols.
- Apply 16-QAM modulation.
- Filter the modulated data through a raised cosine Tx filter.
- Pass the transmitted signal through an AWGN channel.
- Filter the received data through a raised cosine Rx filter.
- Demodulate the filtered data.

```
txData = randi([0 15],1000,1);
modData = modulator(txData);
```

```
txSig = txfilter(modData);
rxSig = channel(txSig);
filtSig = rxfilter(rxSig);
rxData = demodulator(filtSig);
```

Calculate the error statistics and display the symbol error rate.

```
errStats = errorRate(txData,rxData);
symErrRate = errStats(1)
```

```
symErrRate = 0.0222
```

Measure and display the received RMS EVM and maximum EVM values. Take the filter delay into account by deleting the first `rxd+1` symbols. Because there are symbol errors, the EVM may not be totally accurate.

```
[rmsEVM,maxEVM] = evm(filtSig(rxd+1:end))
```

```
rmsEVM = 17.2966
```

```
maxEVM = 40.1595
```

Measure EVM Using Reference Constellation

Generate random data symbols, and apply 8-PSK modulation.

```
d = randi([0 7],2000,1);
txSig = pskmod(d,8,pi/8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,30);
```

Create an EVM object. Measure the RMS EVM using the transmitted signal as the reference.

```
evm = comm.EVM;
rmsEVM1 = evm(txSig,rxSig);
```

Release the EVM object. Configure the object to estimate the EVM of the received signal against a reference constellation.

```
release(evm)
evm.ReferenceSignalSource = 'Estimated from reference constellation';
evm.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Measure the RMS EVM using only the received signal as an input. Verify that it matches the result obtained when using a reference signal.

```
rmsEVM2 = evm(rxSig);
[rmsEVM1 rmsEVM2]
```

```
ans = 1×2

    3.1524    3.1524
```

Measure EVM Using Custom Measurement Interval

Measure the EVM of a noisy 8-PSK signal using two types of custom measurement intervals. Display the results.

Set the number of frames, M, and the number of subframes per frame, K.

```
M = 2;
K = 5;
```

Set the number of symbols in a subframe. Calculate the corresponding frame length.

```
sfLen = 100;
frmLen = K*sfLen
```

```
frmLen = 500
```

Create an EVM object. Configure the object to use a custom measurement interval equal to the frame length.

```
evm1 = comm.EVM('MeasurementIntervalSource','Custom', ...
    'MeasurementInterval',frmLen);
```

Configure the object to measure EVM using an 8-PSK reference constellation.

```
evm1.ReferenceSignalSource = 'Estimated from reference constellation';
evm1.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Create an EVM object, and configure it use a 500-symbol measurement interval with a periodic reset. Configure the object to measure EVM using an 8-PSK reference constellation.

```
evm2 = comm.EVM('MeasurementIntervalSource','Custom with periodic reset', ...
    'MeasurementInterval',frmLen);
evm2.ReferenceSignalSource = 'Estimated from reference constellation';
evm2.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Initialize the EVM and signal-to-noise arrays.

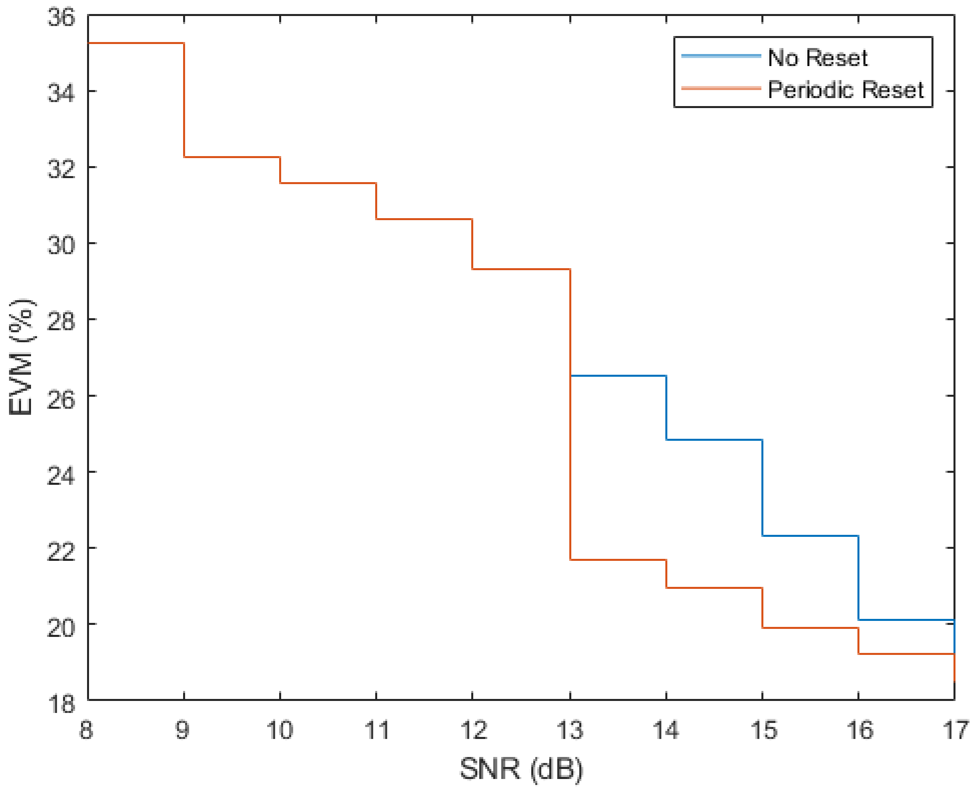
```
rmsEVM1 = zeros(K,M);
rmsEVM2 = zeros(K,M);
snrdB = zeros(K,M);
```

Measure the EVM for a noisy 8-PSK signal using both objects. The SNR increases by 1 dB from subframe to subframe. For `evm1`, the 500 most recent symbols are used to compute the estimate. In this case, a sliding window is used so that an entire frame of data is always processed. For `evm2`, the symbols are cleared each time a new frame is encountered.

```
for m = 1:M
    for k = 1:K
        data = randi([0 7],sfLen,1);
        txSig = pskmod(data,8,pi/8);
        snrdB(k,m) = k+(m-1)*K+7;
        rxSig = awgn(txSig,snrdB(k,m));
        rmsEVM1(k,m) = evm1(rxSig);
        rmsEVM2(k,m) = evm2(rxSig);
    end
end
```

Display the EVM measured using the two approaches. The windowing used in the first case provides an averaging across the subframes. In the second case, the EVM object resets after the first frame so that the calculated EVM values more accurately reflect the current SNR.

```
stairs(snrdB(:),[rmsEVM1(:) rmsEVM2(:)])
xlabel('SNR (dB)')
ylabel('EVM (%)')
legend('No Reset','Periodic Reset')
```



Measure EVM Across Different Dimensions

Create OFDM modulator and demodulator objects.

```
ofdmmod = comm.OFDMModulator('FFTLength',32,'NumSymbols',4);
ofdmdemod = comm.OFDMDemodulator('FFTLength',32,'NumSymbols',4);
```

Determine the number of subcarriers and symbols in the OFDM signal.

```
ofdmDims = info(ofdmmod);
numSC = ofdmDims.DataInputSize(1)
numSC = 21
```

```
numSym = ofdmDims.DataInputSize(2)
```

```
numSym = 4
```

Generate random symbols and apply QPSK modulation.

```
msg = randi([0 3],numSC,numSym);
modSig = pskmod(msg,4,pi/4);
```

OFDM modulate the QPSK signal. Pass the signal through an AWGN channel. Demodulate the noisy signal.

```
txSig = ofdmmod(modSig);
rxSig = awgn(txSig,10,'measured');
demodSig = ofdmdemod(rxSig);
```

Create an EVM object, where the result is averaged over the subcarriers. Measure the EVM. There are four entries corresponding to each of the 4 OFDM symbols.

```
evm = comm.EVM('AveragingDimensions',1);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 1×4

   27.4354   23.6279   22.6772   23.1699
```

Overwrite the EVM object, where the result is averaged over the OFDM symbols. Measure the EVM. There are 21 entries corresponding to each of the 21 subcarriers.

```
evm = comm.EVM('AveragingDimensions',2);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 21×1

   28.8225
   17.8536
   18.6809
   20.8872
   22.3532
   24.7197
   30.1954
   33.4899
   36.2847
   21.4230
       ⋮
```

Measure the EVM and average over both the subcarriers and the OFDM symbols.

```
evm = comm.EVM('AveragingDimensions',[1 2]);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 24.2986
```

Plot Time-Varying EVM for OFDM Signal

Calculate and plot the EVM of an OFDM signal. The signal consists of two packets separated by an interval.

Create System objects to:

- OFDM modulate a signal
- Introduce phase noise
- Plot time-varying signals

```
ofdmmod = comm.OFDMModulator('FFTLength',256,'NumSymbols',2);

pnoise = comm.PhaseNoise('Level',-60,'FrequencyOffset',20,'SampleRate',1000);

tscope = dsp.TimeScope('YLabel','EVM (%)','YLimits',[0 40], ...
    'SampleRate',1000,'TimeSpan',1.2, ...
    'ShowGrid',true);
```

Create an EVM object. To generate a time-varying estimate of the EVM, set the `AveragingDimensions` property to `2`.

```
evm = comm.EVM('MaximumEVMOutputPort',false, ...
    'ReferenceSignalSource','Input port', ...
    'AveragingDimensions',2);
```

Determine the input data dimensions of the OFDM modulator.

```
modDims = info(ofdmmod)
```

```
modDims = struct with fields:
    DataInputSize: [245 2]
```

```
        OutputSize: [544 1]
```

Create QPSK-modulated random data for the first packet. Apply OFDM modulation.

```
data = randi([0 3],modDims.DataInputSize);
qpskSig = pskmod(data,4,pi/4);
txSig1 = ofdmmod(qpskSig);
```

Create a second data packet.

```
data = randi([0 3],modDims.DataInputSize);
qpskSig = pskmod(data,4,pi/4);
txSig2 = ofdmmod(qpskSig);
```

Concatenate the two packets and include an interval in which nothing is transmitted.

```
txSig = [txSig1; zeros(112,1); txSig2];
```

Apply I/Q amplitude and phase imbalance to the transmitted signal.

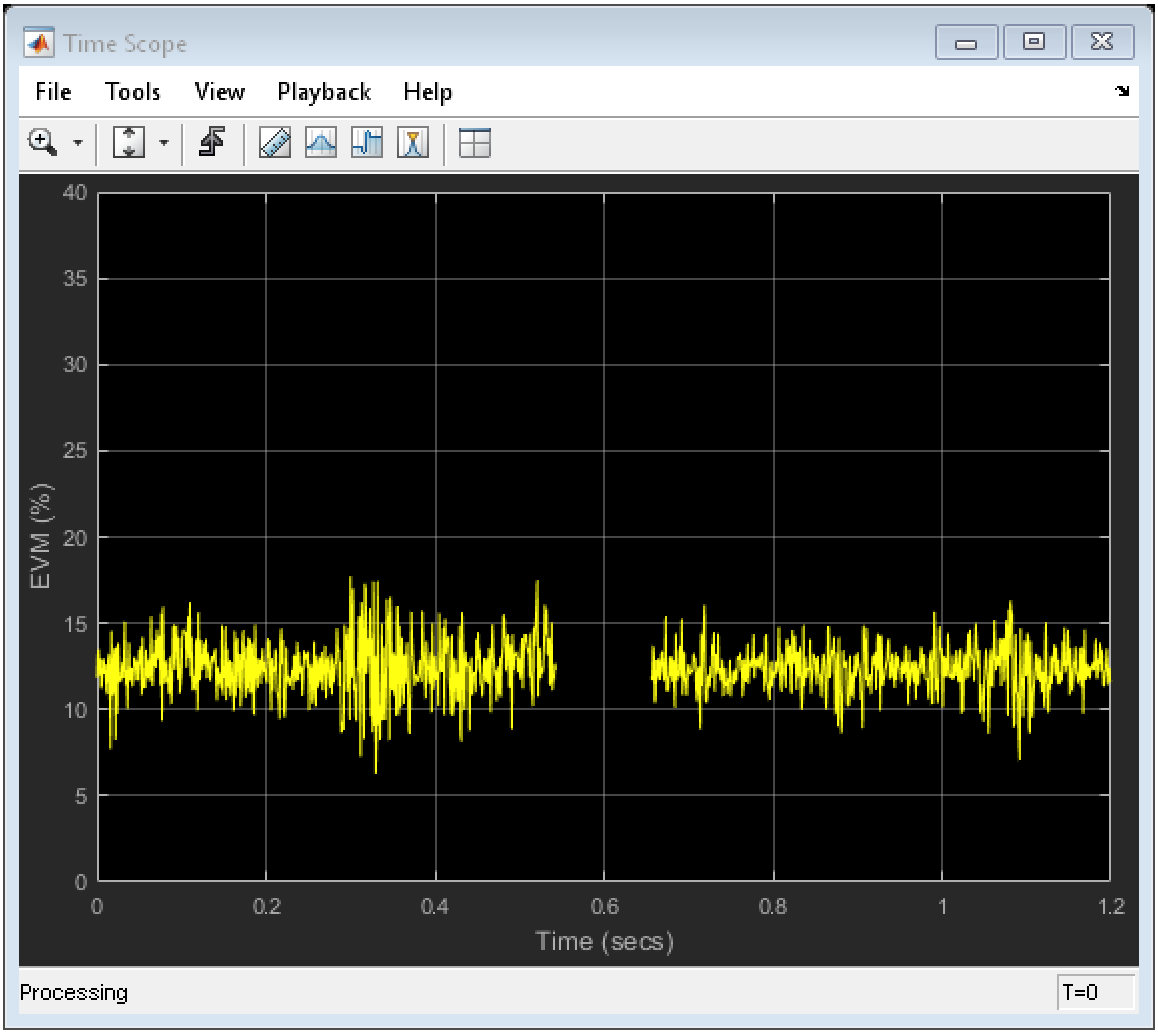
```
rxSigIQimb = iqimbal(txSig,2,5);
```

Apply phase noise.

```
rxSig = pnoise(rxSigIQimb);
```

Measure the EVM of the received signal, and plot its time-varying EVM.

```
e = evm(txSig,rxSig);
tscope(e)
```



Algorithms

Both the EVM block and the EVM object provide three 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 block or object calculates the RMS EVM value differently for each normalization method.

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	$EVM_{RMS}(percent) = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{avg}}} * 100$
Peak power	$EVM_{RMS}(percent) = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\max}}} *100$

Where:

- $$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$
- I_k = In-phase measurement of the *k*th symbol in the burst
- Q_k = Quadrature phase measurement of the *k*th 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 $EVM_{\max} = \max\limits_{k \in [1,...,N]} \{EVM_k\}$, where k is the kth symbol in a burst of length N.

The definition for EVM_k varies depending upon which normalization method you select for computing measurements. The block or object supports these algorithms.

EVM Normalization	Algorithm
Reference signal	$EVM_k = \sqrt{\dfrac{e_k}{\frac{1}{N}\sum\limits_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$EVM_k = \sqrt{\dfrac{e_k}{P_{avg}}} * 100$
Peak power	$EVM_k = \sqrt{\dfrac{e_k}{P_{\max}}} * 100$

The block or object computes the *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 fall.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.CCDF` | `comm.MER`

Introduced in R2012a

reset

System object: comm.EVM
Package: comm

Reset states of EVM measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `EVM` object, `H`.

step

System object: comm.EVM
Package: comm

Measure error vector magnitude

Syntax

```
RMSEVM = step(EVM,REFSYM,RXSYM)
RMSEVM = step(EVM,RXSYM)
[___,MAXEVM] = step(___)
[___,XEVM] = step(___)
[___,NUMSYM] = step(___)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`RMSEVM = step(EVM,REFSYM,RXSYM)` returns the measured root-mean-square EVM, `RMSEVM`, of the received signal `RXSYM`, based on reference signal `REFSYM`. EVM values are measured as a percentage.

`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. All outputs of the object are of data type `double`. To set the interval over which the EVM is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`RMSEVM = step(EVM,RXSYM)` returns the measured EVM of received signal `RXSYM` based on a reference signal specified in the `ReceivedConstellation` property.

`[___,MAXEVM] = step(___)` returns the maximum EVM, `MAXEVM`, given either of the two previous syntaxes.

To return the maximum EVM value, set the `MaximumEVMOutputPort` property to `true`. To set the interval over which the maximum EVM is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`[___,XEVM] = step(___)` returns the *X*-percentile EVM, `XEVM`.

To return the X-percentile EVM, set the `XPercentileEVMOutputPort` property to `true`. `XEVM` is the EVM below which *X*% of the measurements fall, where *X* is set by the `XPercentileValue` property. `XEVM` is measured using all the input frames since the last reset.

`[___,NUMSYM] = step(___)` returns the number of symbols, `NUMSYM`, used to calculate the *X*-percentile EVM.

To return `NUMSYM`, set the `SymbolCountOutputPort` property to `true`. `NUMSYM` is measured using all the input frames since the last reset.

Note `EVM` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.EyeDiagram System object

Package: comm

Display eye diagram of time-domain signals

Description

The eye diagram System object displays multiple traces of a modulated signal to produce an eye diagram. You can use the object to reveal the modulation characteristics of the signal, such as the effects of pulse shaping or channel distortions. The eye diagram can measure signal characteristics and plot horizontal and vertical bathtub curves when the jitter and noise comply with the dual-Dirac model [1].

To display the eye diagram of an input signal:

1. Create a `comm.EyeDiagram` object and set the properties of the object.
2. Call `step` to display the eye diagram of the signal.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ed = comm.EyeDiagram` returns an eye diagram object, `ed`, using the default properties.

`ed = comm.EyeDiagram(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
ed = comm.EyeDiagram('DisplayMode','2D color histogram', ...
                          'OversamplingMethod','Input interpolation');
```

Properties

`Name` — Caption to display on the eye diagram window
`'Eye Diagram'` (default) | character vector

Name displayed on the eye diagram window, specified as a character vector. Tunable.

`SampleRate` — Input signal sample rate (Hz)
`1` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive real scalar.

`SamplesPerSymbol` — Number of samples per symbol
`8` (default) | positive integer scalar

Number of samples per symbol, specified as a positive integer scalar. Tunable.

`SampleOffset` — Number of samples to omit before plotting the first point
`0` (default) | nonnegative integer scalar

Number of samples to omit before plotting the first point, specified as a nonnegative integer scalar. To avoid irregular behavior, specify the offset to be less than the product of `SamplesPerSymbol` and `SamplePerTrace`.

`SymbolsPerTrace` — Number of symbols per trace
`2` (default) | positive integer scalar

Number of symbols per trace, specified as a positive integer scalar. To obtain eye measurements and visualize bathtub curves, use the default value of `2`. Tunable.

`TracesToDisplay` — Number of traces to display
`40` (default) | positive integer scalar

Number of traces to display, specified as a positive integer scalar. This property is available when the `DisplayMode` property is specified as `'Line plot'`. Tunable.

`DisplayMode` — Eye diagram display mode
`'Line plot'` (default) | `'2D color histogram'`

Eye diagram display mode, specified as `'Line plot'` or `'2D color histogram'`. Tunable.

- Specify `'Line plot'` to overlay traces by plotting one line for each of the last `TracesToDisplay` traces.
- Specify `'2D color histogram'` to display a color gradient that shows how often the input matches different time and amplitude values.

`EnableMeasurements` — Enable eye diagram measurements

`false` (default) | `true`

Enable eye diagram measurements, specified as a logical scalar. Tunable.

`ShowBathtub` — Enable visualization of bathtub curves

`'None'` (default) | `'Horizontal'` | `'Vertical'` | `'Both'`

Enable visualization of bathtub curves, specified as `'None'`, `'Horizontal'`, `'Vertical'`, or `'Both'`. This property is available when `EnableMeasurements` is `true`. Tunable.

`OverlayHistogram` — Histogram overlay

`'None'` (default) | `'Jitter'` | `'Noise'`

Histogram overlay, specified as `'None'`, `'Jitter'`, or `'Noise'`.

- To overlay a horizontal histogram on the eye diagram, set this property to `'Jitter'`.
- To overlay a vertical histogram on the eye diagram, set this property to `'Noise'`.

This property is available when `DisplayMode` is `'2D color histogram'` and `EnableMeasurements` is `true`. Tunable.

`DecisionBoundary` — Amplitude level threshold

`0` (default) | scalar

Amplitude level threshold in V, specified as a scalar. This property separates the different signaling regions for horizontal (jitter) histograms, and is available when `EnableMeasurements` is `true`. This property is tunable, but the jitter histograms reset when the property changes.

For non-return-to-zero (NRZ) signals, set `DecisionBoundary` to 0. For return-to-zero (RZ) signals, set `DecisionBoundary` to half the maximum amplitude.

`EyeLevelBoundaries` — Time range for calculating eye levels

`[40 60]` (default) | vector

Time range for calculating eye levels, specified as a two-element vector. These values are expressed as a percentage of the symbol duration. This property is available when `EnableMeasurements` is `true`. Tunable.

`RiseFallThresholds` — Amplitude levels of the rise and fall transitions

`[10 90]` (default) | vector

Amplitude levels of the rise and fall transitions, specified as a two-element vector. These values are expressed as a percentage of the eye amplitude. This property is available when `EnableMeasurements` is `true`. This property is tunable but the crossing histograms of the rise and fall thresholds reset when it is changed.

`Hysteresis` — Amplitude tolerance of the horizontal crossings

`0` (default) | scalar

Amplitude tolerance of the horizontal crossings in V, specified as a scalar. Increase hysteresis to provide more tolerance to spurious crossings due to noise. This property is available when `EnableMeasurements` is `true`. This property is tunable, but the jitter and the rise and fall histograms reset when the property changes.

`BERThreshold` — BER used for eye measurements

`1e-12` (default) | nonnegative scalar from 0 to 0.5

BER used for eye measurements, specified as a nonnegative scalar from 0 to 0.5. The value is used to make measurements of random jitter, total jitter, horizontal eye openings, and vertical eye openings. This property is available when `EnableMeasurements` is `true`. Tunable.

`BathtubBER` — BER values used to calculate openings of bathtub curves

`[0.5 10.^-(1:12)]` | vector

BER values used to calculate openings of bathtub curves, specified as a vector whose elements range from 0 to 0.5. Horizontal and vertical eye openings are calculated for each of the values specified by this property. This property is available when `EnableMeasurements` is `true` and `ShowBathtub` is `'Both'`, `'Horizontal'`, or `'Vertical'`. Tunable.

`MeasurementDelay` — Duration of initial data discarded from measurements

`0` (default) | nonnegative scalar

Duration of initial data discarded from measurements in seconds, specified as a nonnegative scalar. This property is available when `EnableMeasurements` is `true`.

`OversamplingMethod` — Oversampling method

`'None'` (default) | `'Input interpolation'` | `'Histogram interpolation'`

Oversampling method, specified as `'None'`, `'Input interpolation'`, or `'Histogram interpolation'`. This property is available when `DisplayMode` is `'2D color histogram'`. Tunable.

To plot eye diagrams as quickly as possible, set `OversamplingMethod` to `'None'`. The drawback to not oversampling is that the plots look pixelated when the number of samples per trace is small.

To create smoother, less-pixelated plots using a small number of samples per trace, set `OversamplingMethod` to`'Input interpolation'` or `'Histogram interpolation'`. Input interpolation is the faster interpolation method and produces good results when the signal-to-noise ratio (SNR) is high. With a lower SNR, this oversampling method is not recommended because it introduces a bias to the centers of the histogram ranges. Histogram interpolation is not as fast as the other techniques, but it provides good results even when the SNR is low.

`ColorScale` — Color scale of the histogram

`'Linear'` (default) | `'Logarithmic'`

Color scale of the histogram, specified as `'Linear'` or `'Logarithmic'`. Change this property if certain areas of the histogram include a disproportionate number of points. Use the `'Logarithmic'` option for eye diagrams having sharp peaks, where the signal repetitively matches specific time and amplitude values. This property is available when `DisplayMode` is `'2D color histogram'`. Tunable.

`ColorFading` — Color fading

`false` (default) | `true`

Color fading, specified as a logical scalar. To fade the points in the display as the interval of time after they are first plotted increases, set this property to `true`. This animation resembles an oscilloscope. This property is available when `DisplayMode` is `'Line plot'`. Tunable.

`ShowImaginaryEye` — Show imaginary signal component

`false` (default) | `true`

Show imaginary signal component, specified as a logical scalar. To view the imaginary or quadrature component of the input signal, set this property to `true`. This property is available when `EnableMeasurements` is `false`. Tunable.

`YLimits` — Y-axis limits

`[-1.1 1.1]` (default) | two-element vector

Y-axis limits of the eye diagram in V, specified as a two-element vector. The first element corresponds to *ymin* and the second to *ymax*. The second element must be greater than the first. Tunable.

`ShowGrid` — Enable grid display

`false` (default) | `true`

Enable grid display on eye diagram, specified as a logical scalar. To display a grid on the eye diagram, set this property to `true`. Tunable.

`Position` — Scope window position

vector

Scope window position in pixels, specified as a four-element vector of the form `[left bottom width height]`. Tunable.

Methods

hide	Hide scope window
horizontalBathtub	Horizontal bathtub curve
jitterHistogram	Jitter histogram
measurements	Measure eye diagram parameters
noiseHistogram	Noise histogram
reset	Reset states of eye diagram object
show	Make scope window visible
step	Plot eye diagram of input signal
verticalBathtub	Vertical bathtub curve

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Eye Diagram of Filtered QPSK Signal

Specify the sample rate and the number of output samples per symbol parameters.

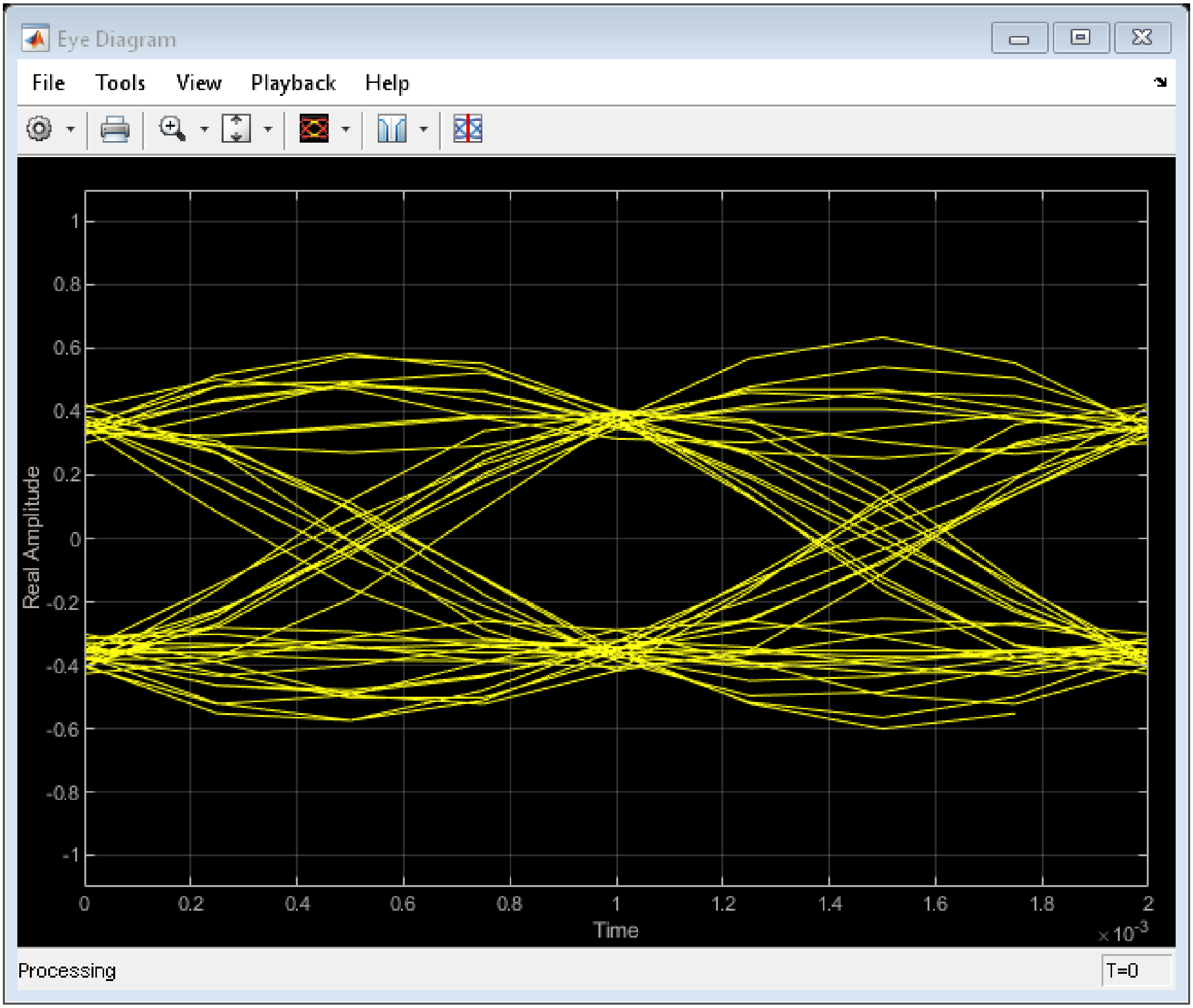
```
fs = 1000;
sps = 4;
```

Create transmit filter and eye diagram objects.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps);
ed = comm.EyeDiagram('SampleRate',fs*sps,'SamplesPerSymbol',sps);
```

Generate random symbols and apply QPSK modulation. Then filter the modulated signal and display the eye diagram.

```
data = randi([0 3],1000,1);
modSig = pskmod(data,4,pi/4);

txSig = txfilter(modSig);
ed(txSig)
```



Effect of Interpolation on 2D Histogram Eye Diagrams

Show the effects of different interpolation methods on 2-D histograms for different signal-to-noise ratio (SNR) conditions.

Create GMSK modulator and eye diagram System objects. Specify that the eye diagram displays using a 2-D color histogram and plots the real and imaginary signals.

```
gmsk = comm.GMSKModulator('PulseLength',3);
ed = comm.EyeDiagram('DisplayMode','2D color histogram', ...
    'ShowImaginaryEye',true,'YLimits',[-2 2]);
```

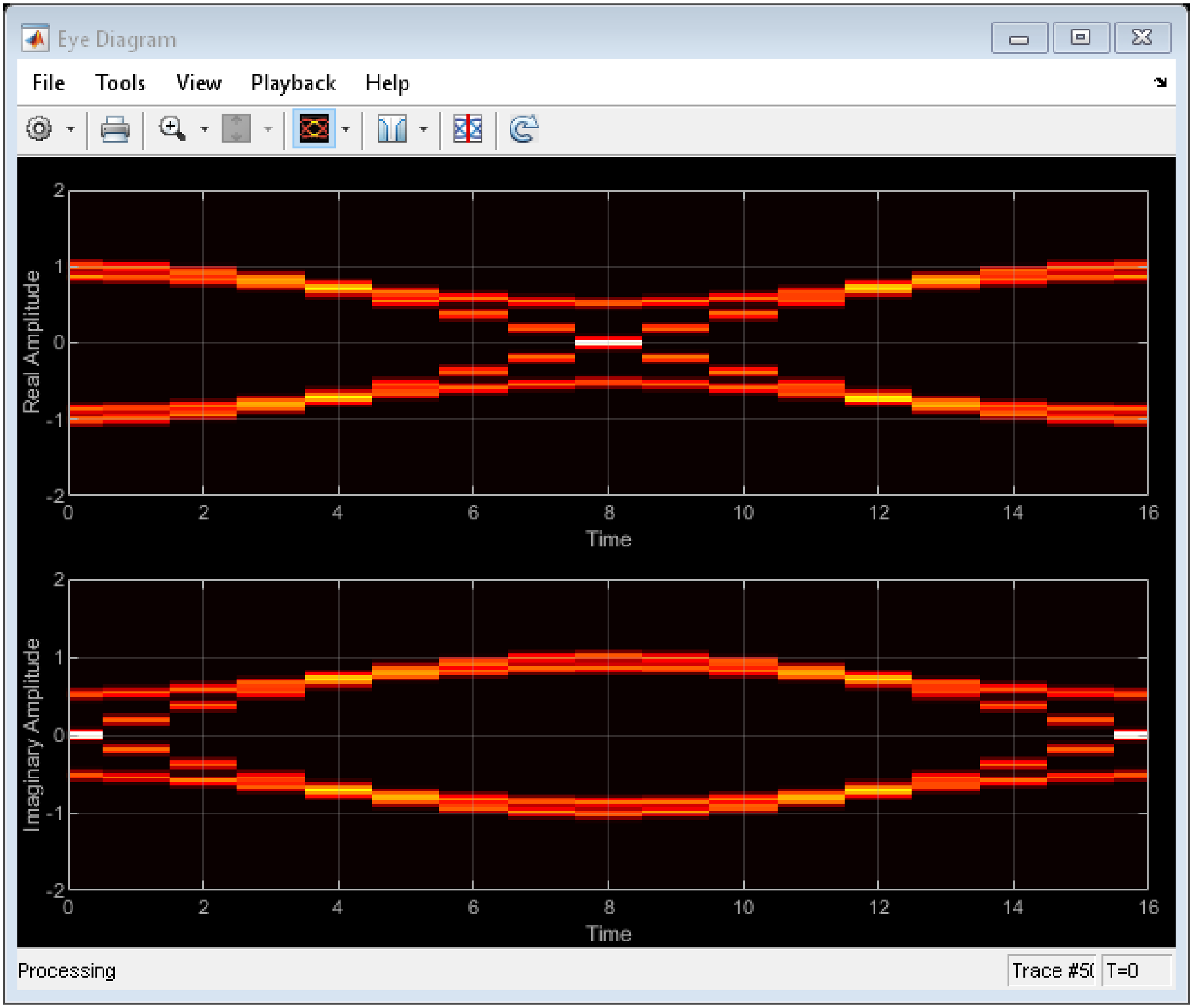
Generate bipolar data and apply GMSK modulation.

```
d = 2*randi([0 1],1e4,1)-1;
x = gmsk(d);
```

```
%Pass the signal through an AWGN channel having a 25 dB SNR and with a fixed seed for r
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,25,'measured',randStream);
```

Display the eye diagram.

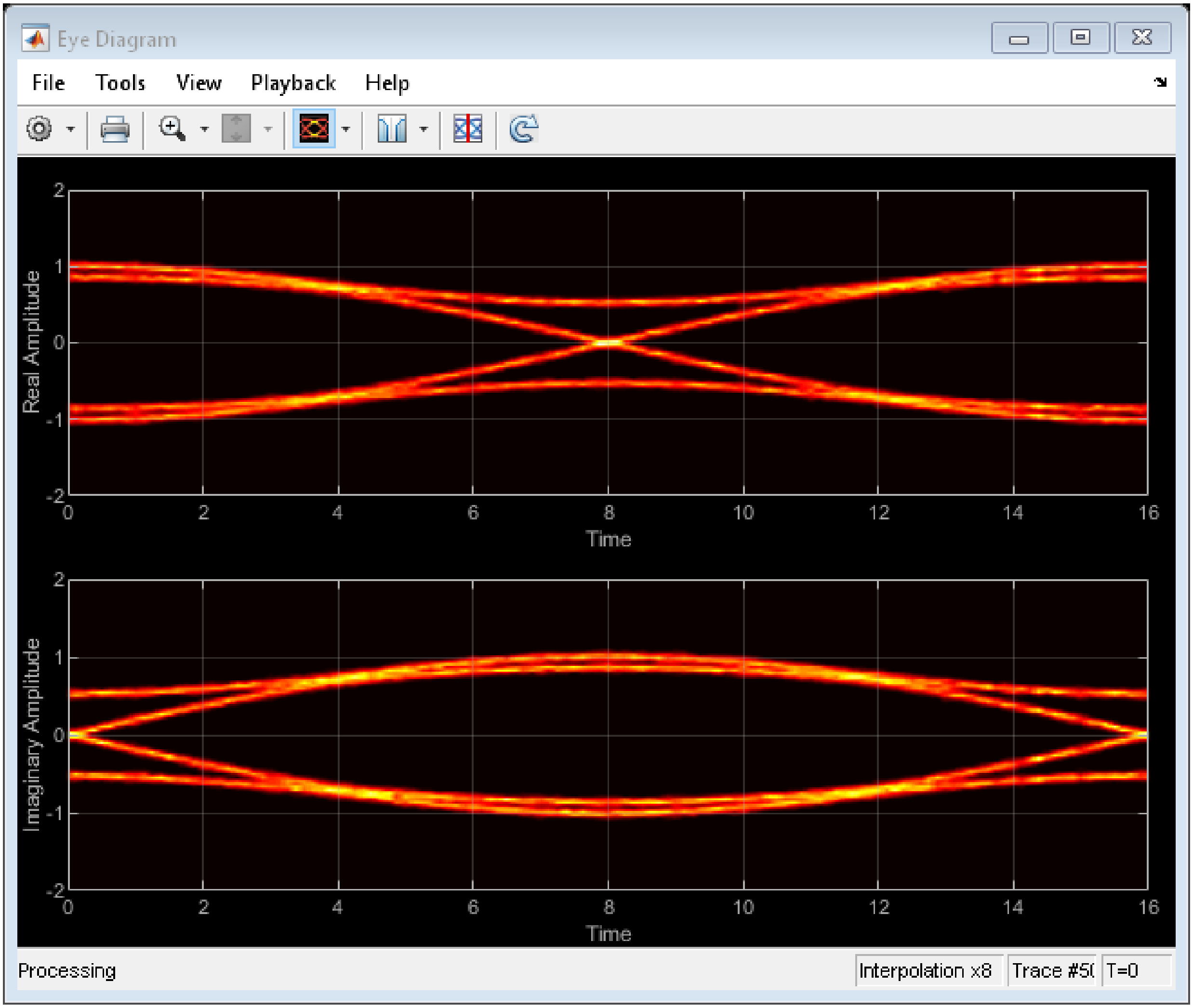
```
ed(y)
```



For a small number of samples per trace (16), the lack of interpolation causes piecewise-continuous behavior.

To compensate for the piecewise-continuous behavior, set the `OversamplingMethod` property to `'Input interpolation'`. Reset the object and display the eye diagram.

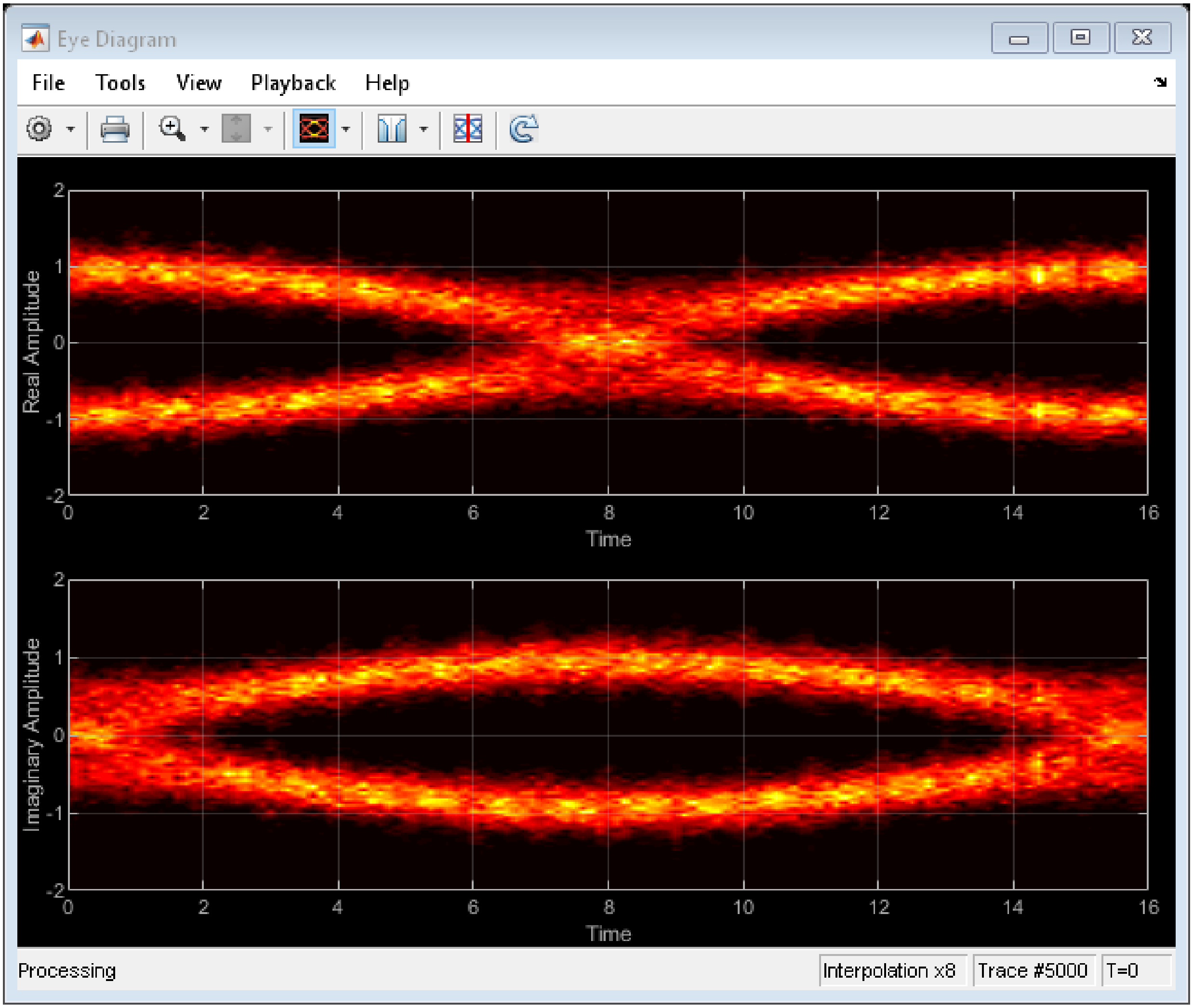
```
ed.OversamplingMethod = 'Input interpolation';
reset(ed)
ed(y)
```



The interpolation smooths the eye diagram.

Now pass the GMSK-modulated signal through an AWGN channel having a 10 dB SNR. Display the eye diagram.

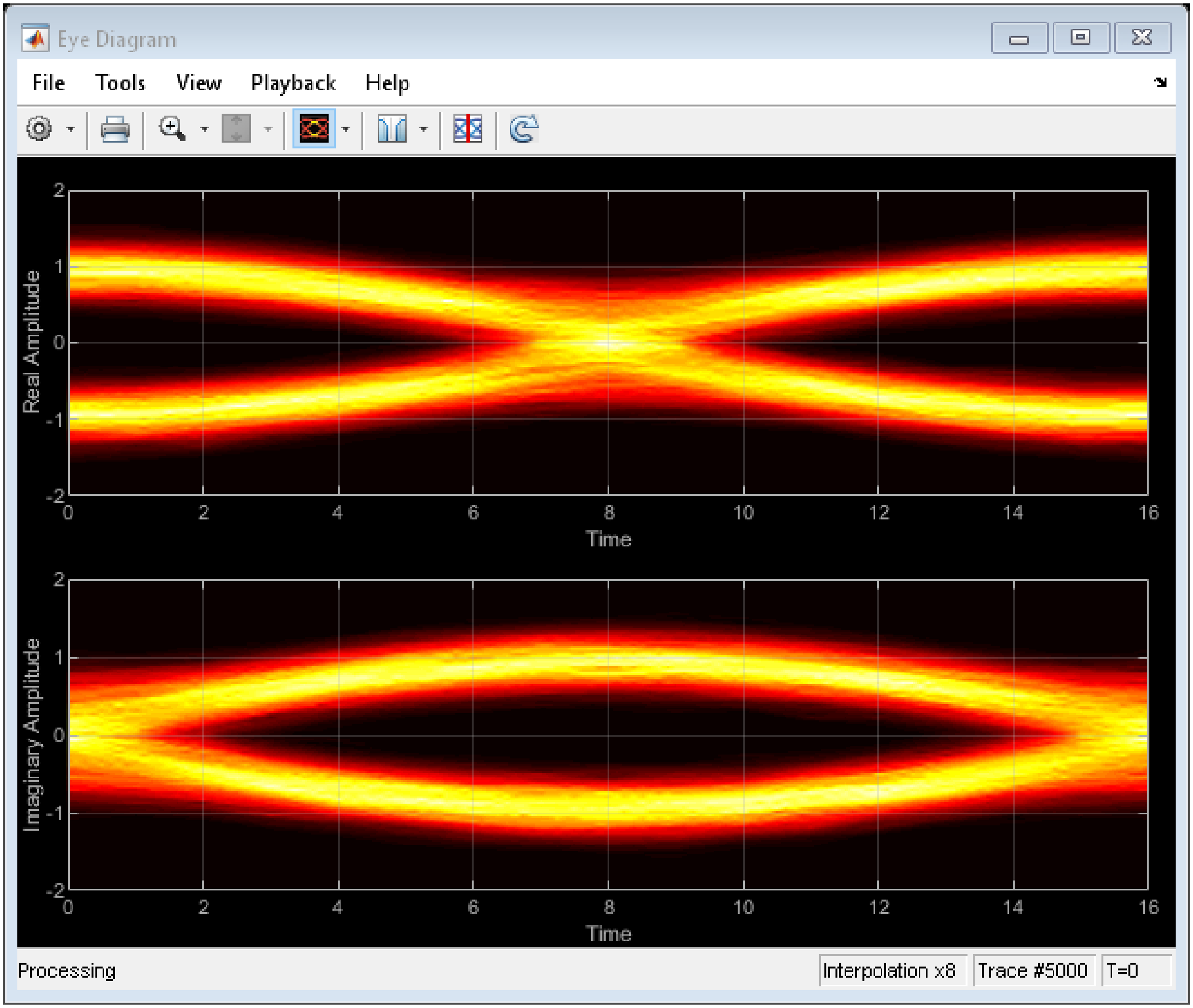
```
y = awgn(x,10,'measured',randStream);
reset(ed)
ed(y)
```



The vertical striping is the result of input interpolation, which has limited accuracy in low-SNR conditions.

Set the `OversamplingMethod` property to `'Histogram interpolation'`. Plot the eye diagram.

```
ed.OversamplingMethod = 'Histogram interpolation';
reset(ed)
ed(y)
```



The eye diagram plot now renders accurately because the histogram interpolation method works for all SNR values. This method results in increased execution time.

Eye Diagram Jitter Measurements and Bathtub Curve Plots

Visualize the eye diagram of a dual-dirac input signal. Compute eye measurements, and visualize horizontal and vertical bathtub curves. Overlay the horizontal (jitter) histogram.

Specify the sample rate, the samples per symbol, and the number of traces.

```
fs = 10000;
sps = 200;
numTraces = 2000;
```

Create an eye diagram object having these properties:

- 2-D color histogram display
- Logarithmic color scale
- Jitter histogram overlay
- Horizontal and vertical bathtub curves
- Y-axis limits of [-1.3 1.3]
- Increased window height

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'OverlayHistogram','Jitter', ...
    'ShowBathtub','Both','YLimits', [-1.3 1.3]);
ed.Position = ed.Position + [0 0 0 120];
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

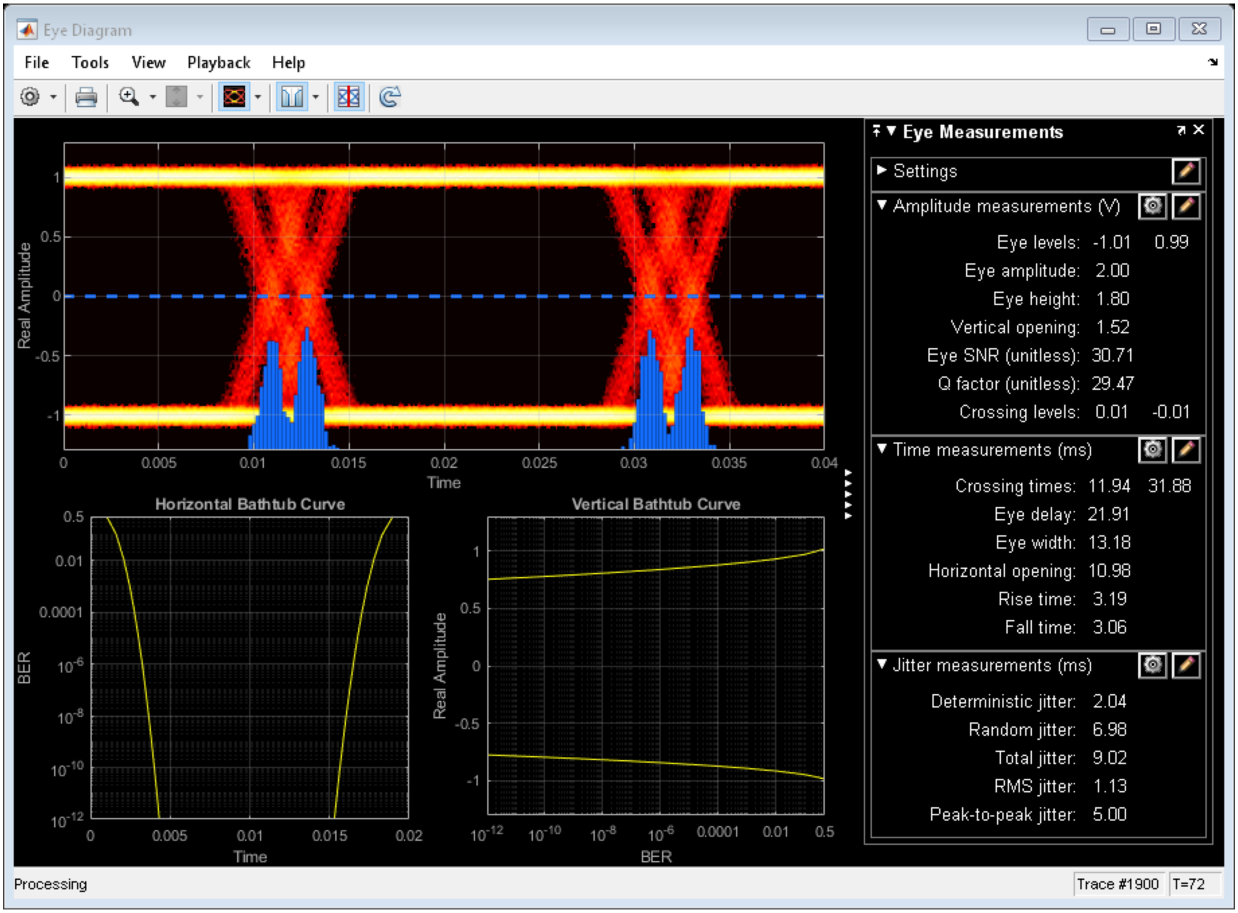
Generate two symbols for each trace.

```
symbols = src.generate(numTraces*2);
```

Process the data in packets of 40e3 symbols, add noise, and display the eye diagram.

```
for idx = 1:(numTraces-1)/100
    x = symbols(1+(idx-1)*100*2*sps:idx*100*2*sps); % Read 40,000 symbols
    y = awgn(x,30);                                 % Add noise
```

```
        ed(y);                                              % Display eye diagram
    end
```



Eye Measurements

Settings

Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.52
Eye SNR (unitless): 30.71
Q factor (unitless): 29.47
Crossing levels: 0.01 -0.01

Time measurements (ms)

Crossing times: 11.94 31.88
Eye delay: 21.91
Eye width: 13.18
Horizontal opening: 10.98
Rise time: 3.19
Fall time: 3.06

Jitter measurements (ms)

Deterministic jitter: 2.04
Random jitter: 6.98
Total jitter: 9.02
RMS jitter: 1.13
Peak-to-peak jitter: 5.00

Jitter and Noise Histogram Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Plot the jitter and noise histograms.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

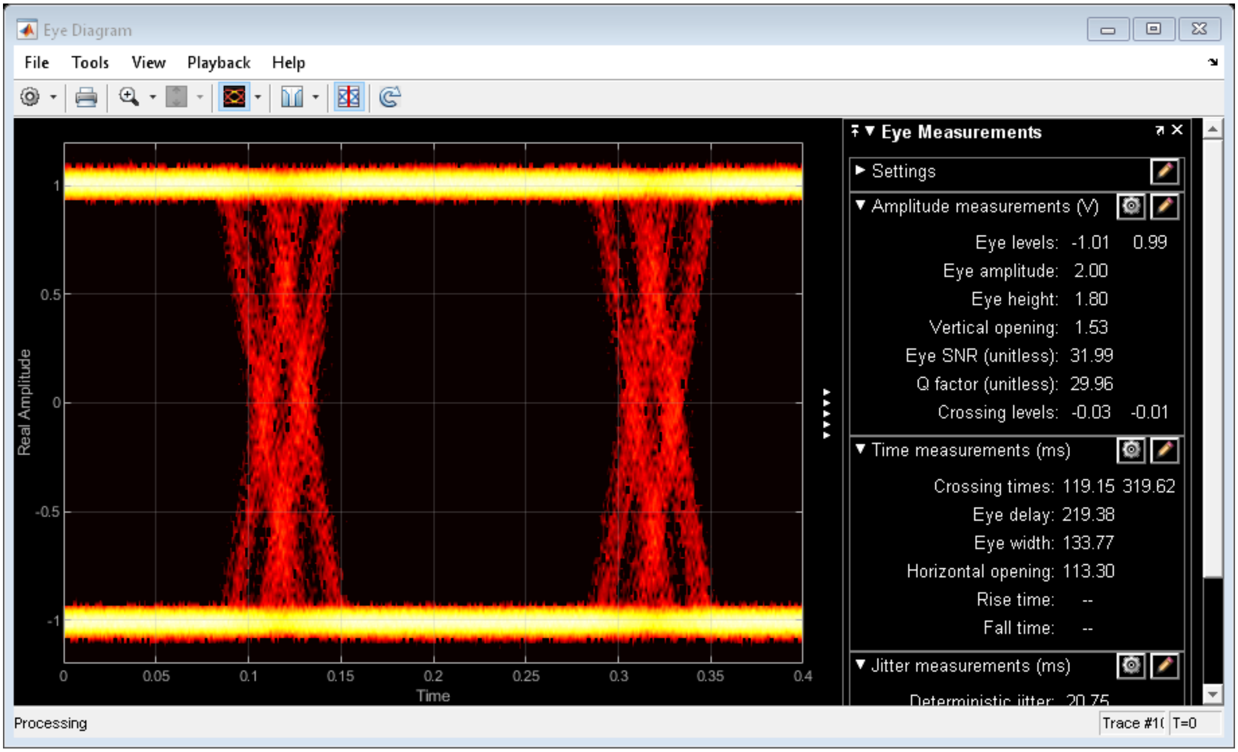
```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

Generate two symbols for each trace.

```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
ed(y)
```



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.99
Q factor (unitless): 29.96
Crossing levels: -0.03 -0.01

▼ Time measurements (ms)

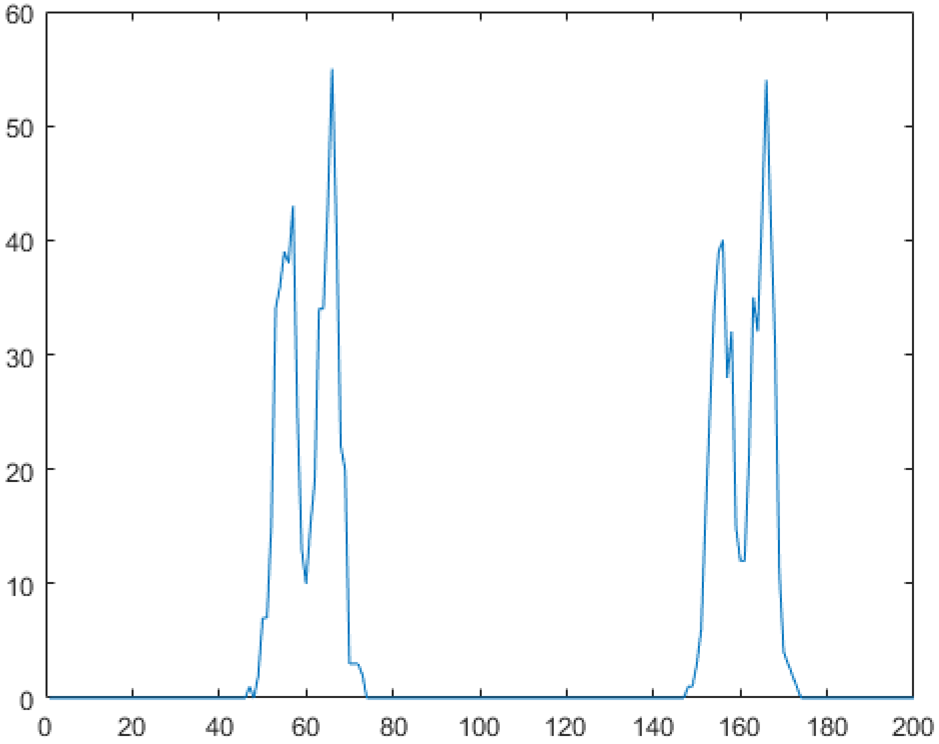
Crossing times: 119.15 319.62
Eye delay: 219.38
Eye width: 133.77
Horizontal opening: 113.30
Rise time: --
Fall time: --

▼ Jitter measurements (ms)

Deterministic jitter: 20.75

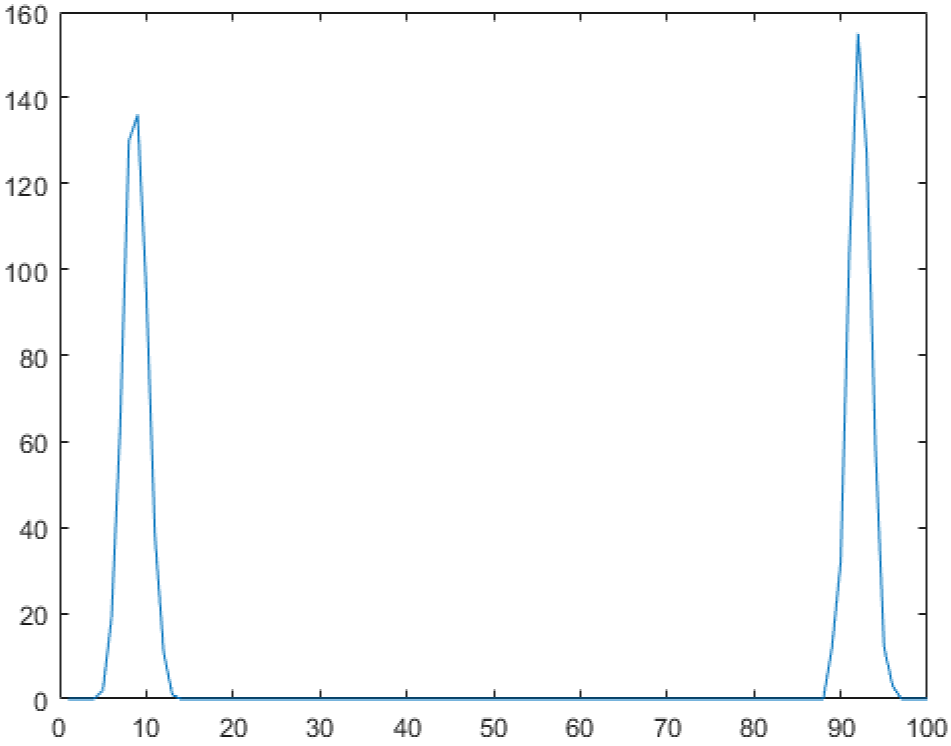
Calculate the jitter histogram count for each bin by using the `jitterHistogram` method. Plot the histogram.

```
jbins = jitterHistogram(ed);
plot(jbins)
```



Calculate the noise histogram count for each bin by using the `noiseHistogram` method. Plot the histogram.

```
nbins = noiseHistogram(ed);
plot(nbins)
```



Horizontal and Vertical Bathtub Curve Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Generate and plot the horizontal and vertical bathtub curves.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'ShowBathtub','Both','YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-5e-04 5e-04],'RandomStd',2e-4);
```

Generate two symbols for each trace.

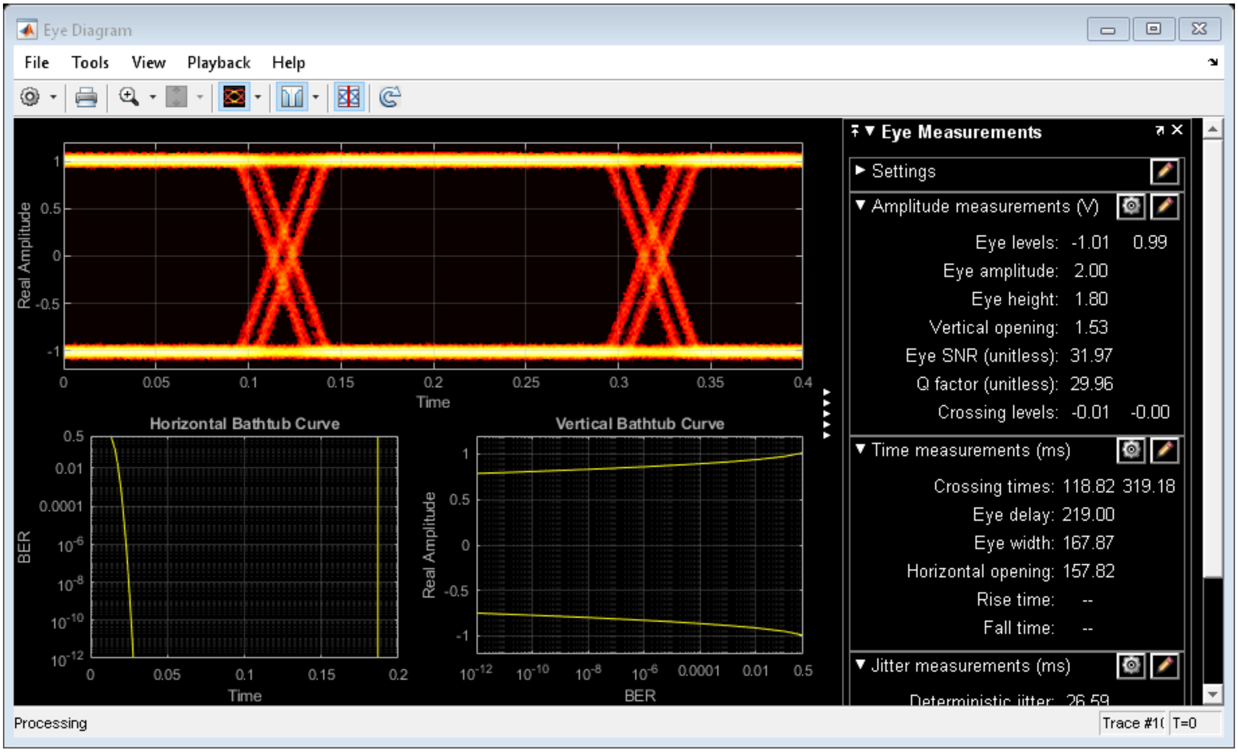
```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Display the eye diagram.

```
ed(y)
```



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

```
                Eye levels:  -1.01    0.99
             Eye amplitude:   2.00
                Eye height:   1.80
          Vertical opening:   1.53
        Eye SNR (unitless):  31.97
       Q factor (unitless):  29.96
           Crossing levels:  -0.01   -0.00
```

▼ Time measurements (ms)

```
            Crossing times: 118.82 319.18
                 Eye delay: 219.00
                 Eye width: 167.87
        Horizontal opening: 157.82
                 Rise time:    --
                 Fall time:    --
```

▼ Jitter measurements (ms)

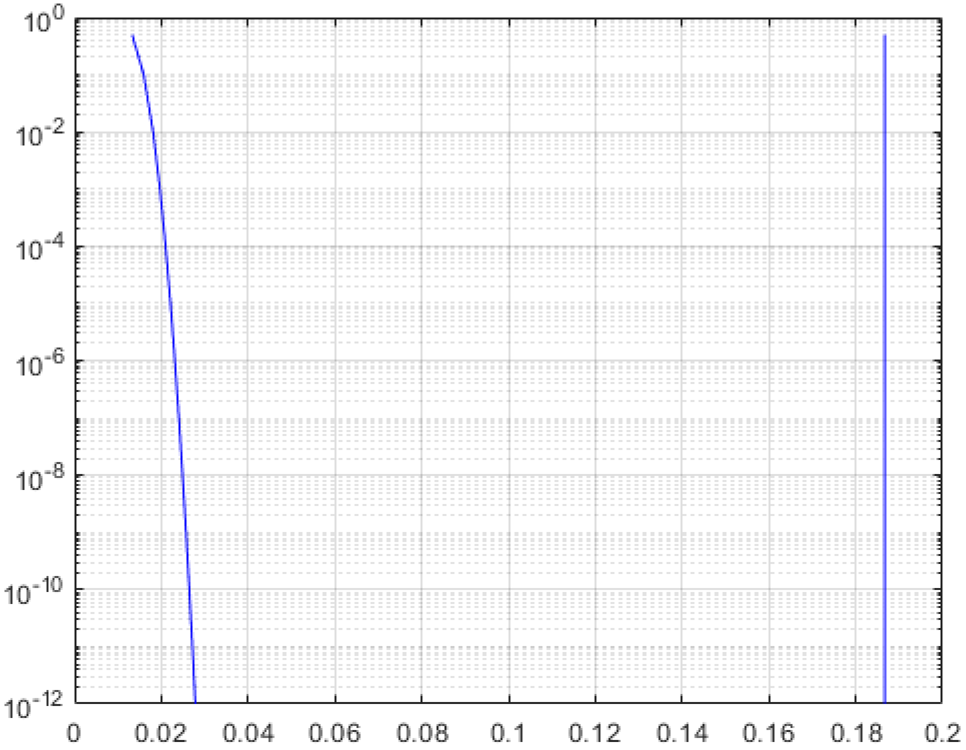
```
       Deterministic jitter:  26.59
```

Generate the horizontal bathtub data for the eye diagram. Plot the curve.

```
hb = horizontalBathtub(ed)
```

```
hb = 1x13 struct array with fields:
    BER
    LeftThreshold
    RightThreshold
```

```
semilogy([hb.LeftThreshold],[hb.BER],'b',[hb.RightThreshold],[hb.BER],'b')
grid
```

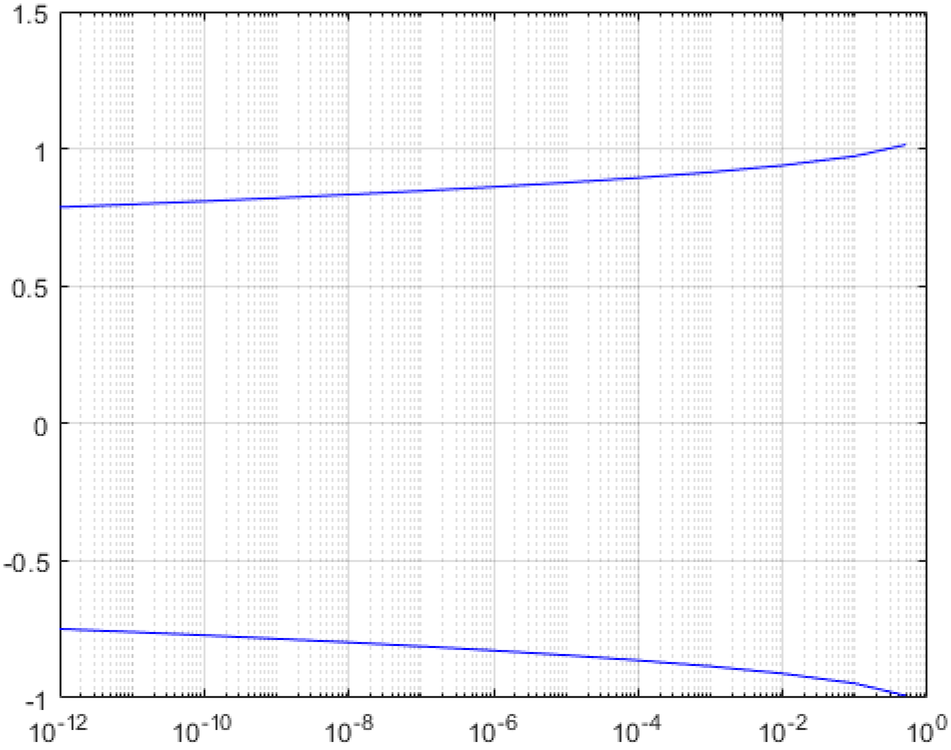


Generate the vertical bathtub data for the eye diagram. Plot the curve.

```
vb = verticalBathtub(ed)
```

```
vb = 1x13 struct array with fields:
    BER
    UpperThreshold
    LowerThreshold
```

```
semilogx([vb.BER],[vb.LowerThreshold],'b',[vb.BER],[vb.UpperThreshold],'b')
grid
```



Rise and Fall Time of NRZ Signal

Create a combined jitter object having random jitter with a 2e-4 standard deviation.

```
jtr = commsrc.combinedjitter('RandomJitter','on','RandomStd',2e-4);
```

Generate an NRZ signal having random jitter and 3 ms rise and fall times.

```
genNRZ = commsrc.pattern('Jitter',jtr,'RiseTime',3e-3,'FallTime',3e-3);
x = generate(genNRZ,2000);
```

Pass the signal through an AWGN channel with fixed seed for repeatable results.

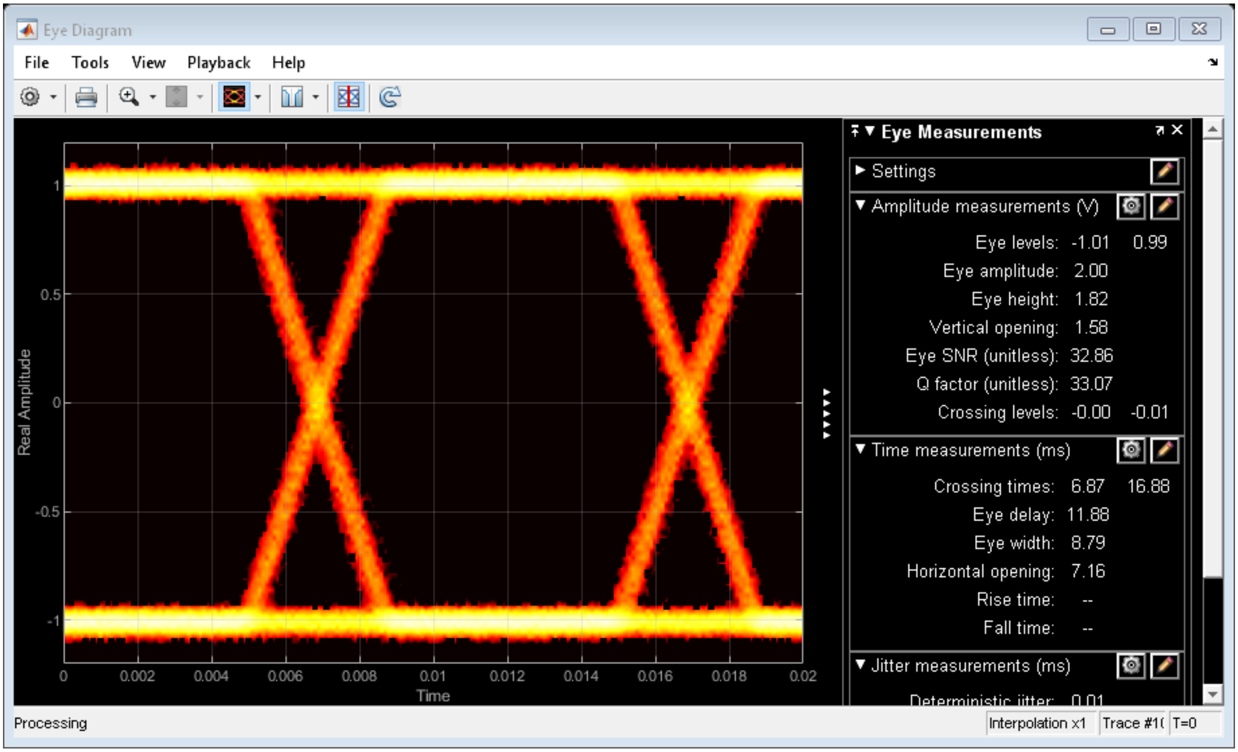
```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Create an eye diagram object. Enable the measurements.

```
ed = comm.EyeDiagram('SamplesPerSymbol',genNRZ.SamplesPerSymbol, ...
    'SampleRate',genNRZ.SamplingFrequency,'SampleOffset',genNRZ.SamplesPerSymbol/2, ..
    'EnableMeasurements',true,'DisplayMode','2D color histogram', ...
    'OversamplingMethod','Input interpolation','ColorScale','Logarithmic','YLimits',[-
```

To compute the rise and fall times, determine the rise and fall thresholds from the eye level and eye amplitude measurements. Plot the eye diagram to calculate these parameters.

```
ed(y)
```



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.82
Vertical opening: 1.58
Eye SNR (unitless): 32.86
Q factor (unitless): 33.07
Crossing levels: -0.00 -0.01

▼ Time measurements (ms)

Crossing times: 6.87 16.88
Eye delay: 11.88
Eye width: 8.79
Horizontal opening: 7.16
Rise time: --
Fall time: --

▼ Jitter measurements (ms)

Deterministic jitter: 0.01

Pass the signal through the eye diagram object again to measure the rise and fall times.

```
ed(y)
hide(ed)
```

Display the data by using the `measurements` method.

```
eyestats = measurements(ed);
riseTime = eyestats.RiseTime
fallTime = eyestats.FallTime


riseTime =

    0.0030


fallTime =

    0.0030
```

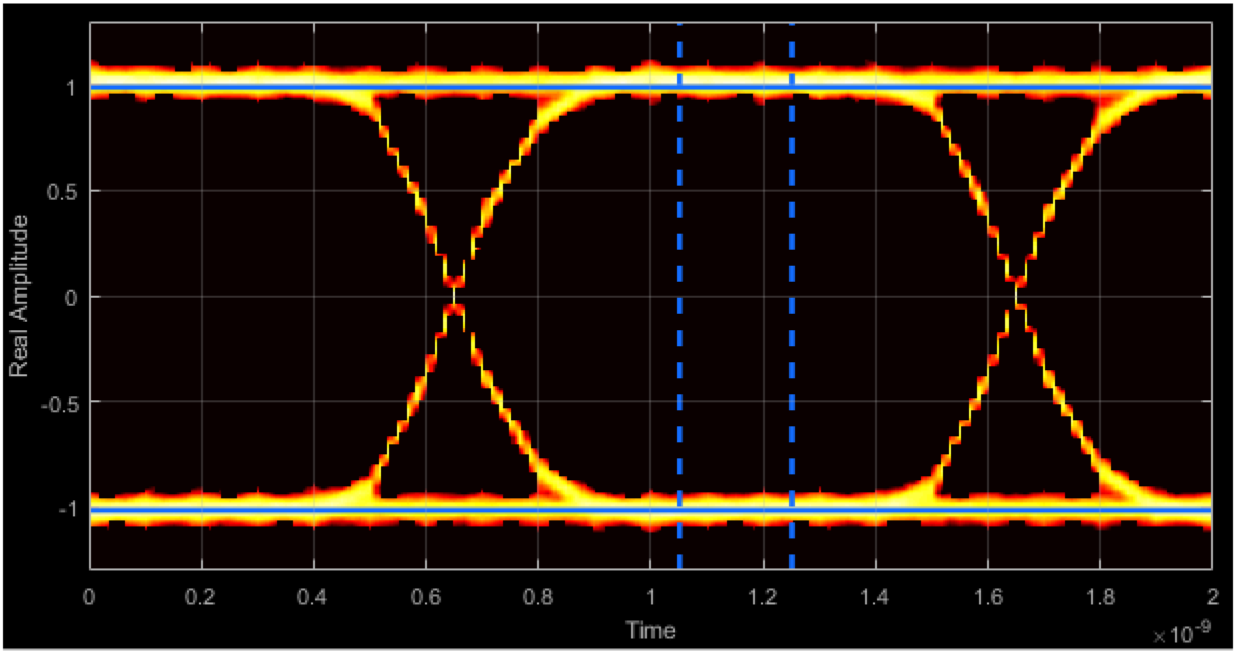
The measured values match the 3 ms specification.

Measurements

To open the measurements panel, click on the **Eye Measurements** button or select Tools > Measurements > Eye Measurements from the toolbar menu.

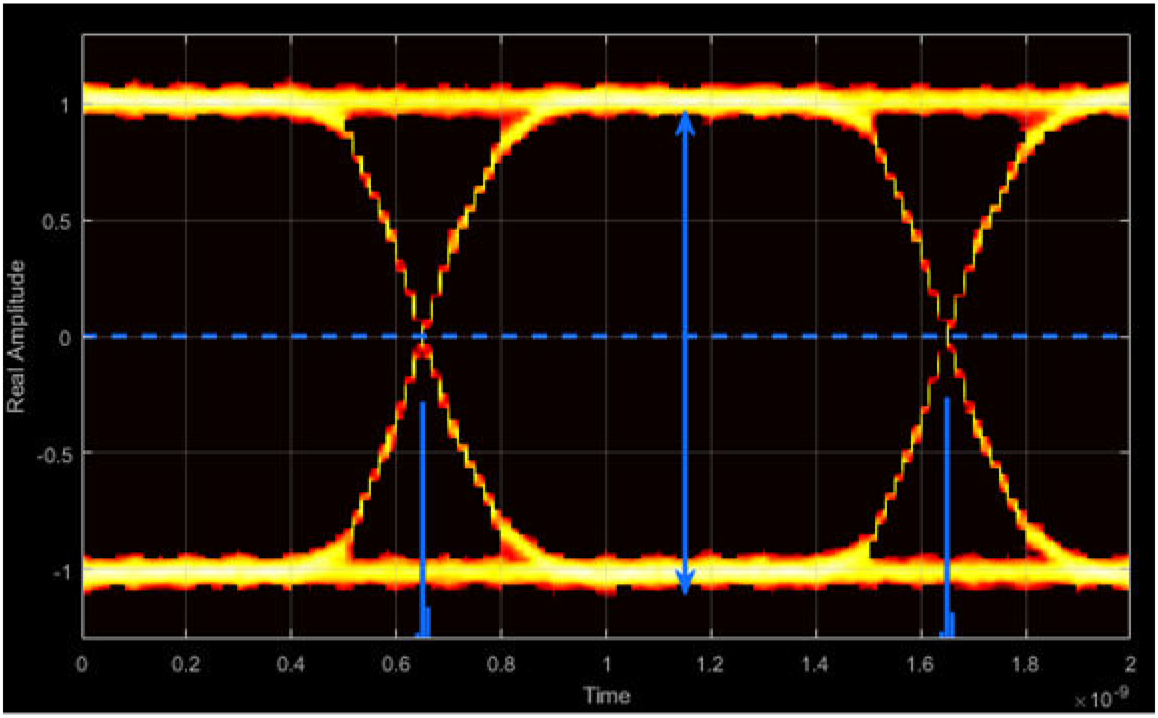
Eye Levels — Amplitude level used to represent data bits

Eye level is the amplitude level used to represent data bits. For the displayed NRZ signal, the levels are –1 V and +1 V. The eye levels are calculated by averaging the 2-D histogram within the eye level boundaries.



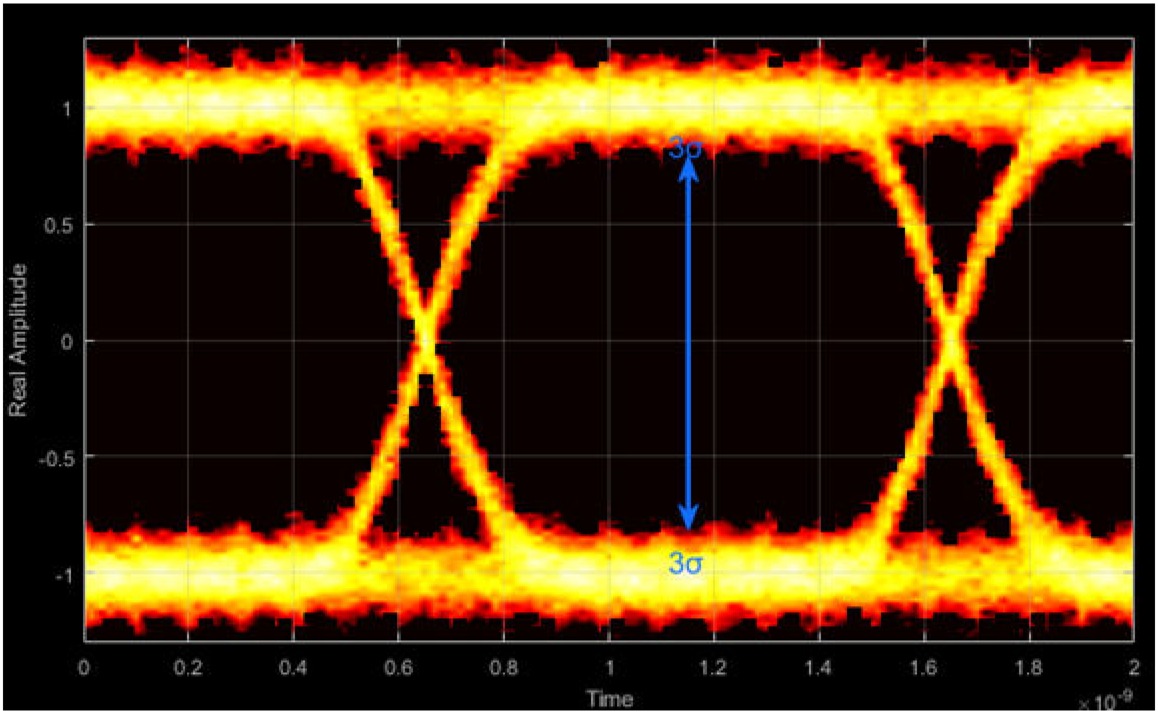
`Eye Amplitude` — Distance between eye levels

Eye amplitude is the distance in V between the mean value of two eye levels.



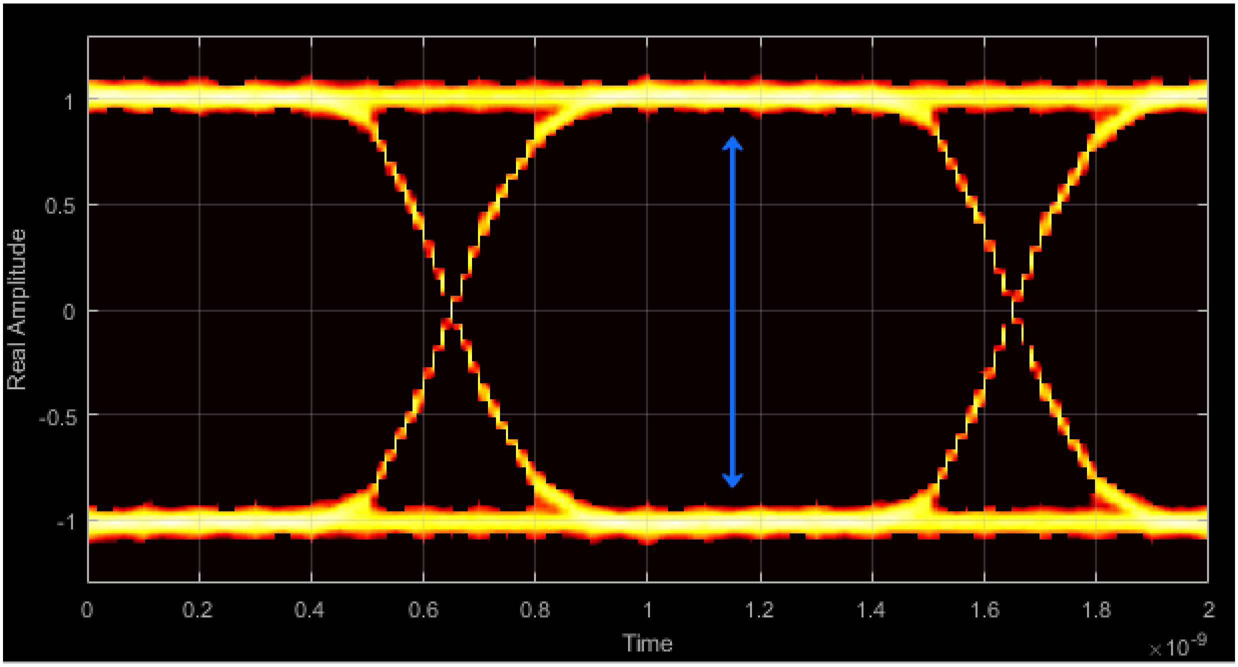
Eye Height — Statistical minimum distance between eye levels

Eye height is the distance between μ – 3σ of the upper eye level and μ + 3σ of the lower eye level. μ is the mean of the eye level and σ is the standard deviation.



`Vertical Opening` — Distance between BER threshold points

The vertical opening is the distance between the two points that correspond to the BER threshold. For example, for a BER threshold of 10^{-12}, these points correspond to the 7σ distance from each eye level.



Eye SNR — Signal-to-noise ratio

The eye SNR is the ratio of the eye level difference to the difference of the vertical standard deviations corresponding to each eye level:

$$\mathrm{SNR} = \frac{L_1 - L_0}{\sigma_1 - \sigma_0},$$

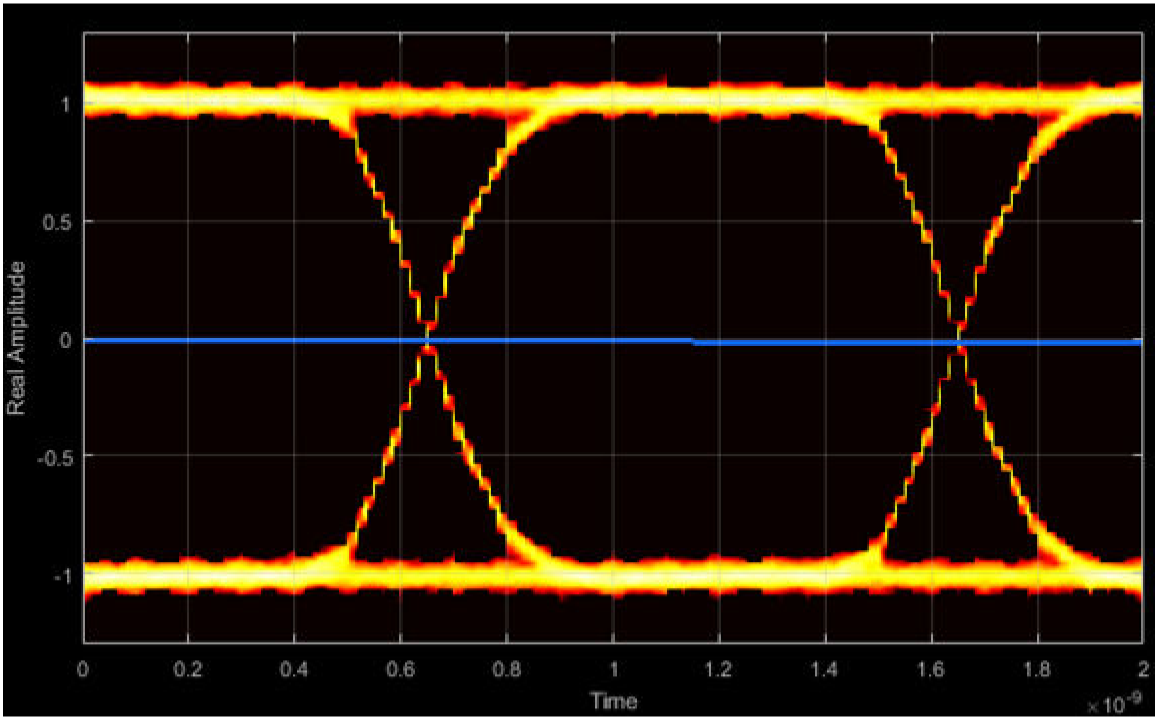
where L_1 and L_0 represent the means of the upper and lower eye levels and σ_1 and σ_0 represent their standard deviations.

Q Factor — Quality factor

The Q factor is calculated using the same formula as the Eye SNR. However, the standard deviations of the vertical histograms are replaced with those computed with the dual-Dirac analysis.

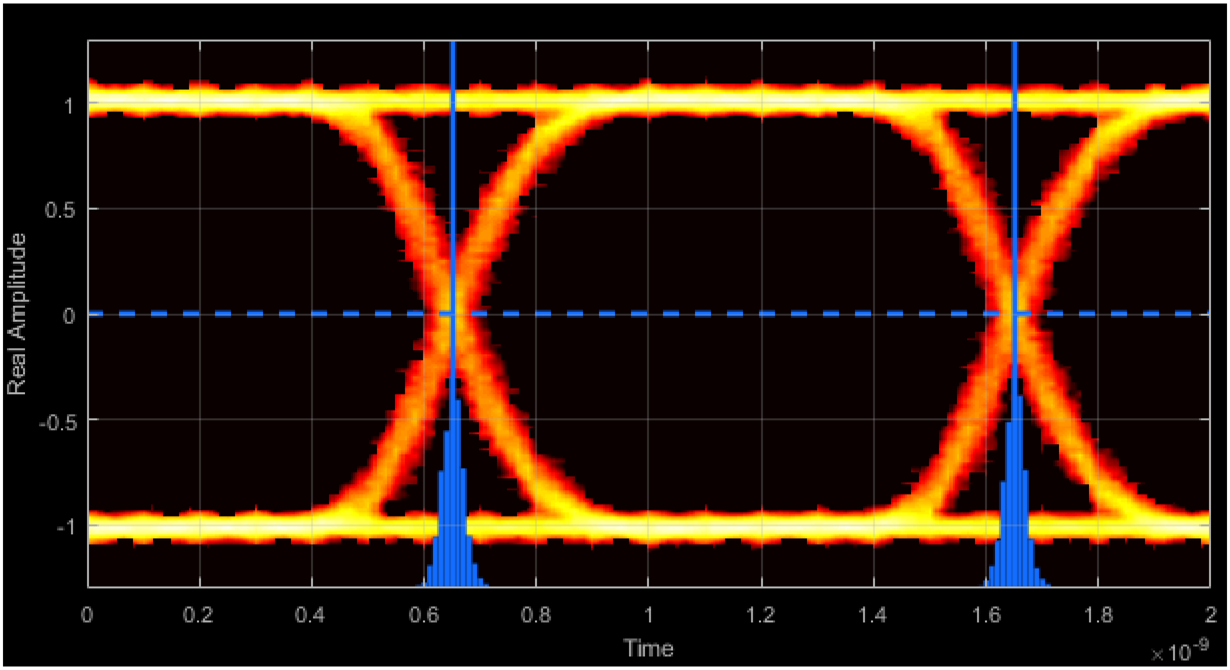
`Crossing Levels` — Amplitude levels for eye crossings

The crossing levels are the amplitude levels at which the eye crossings occur.



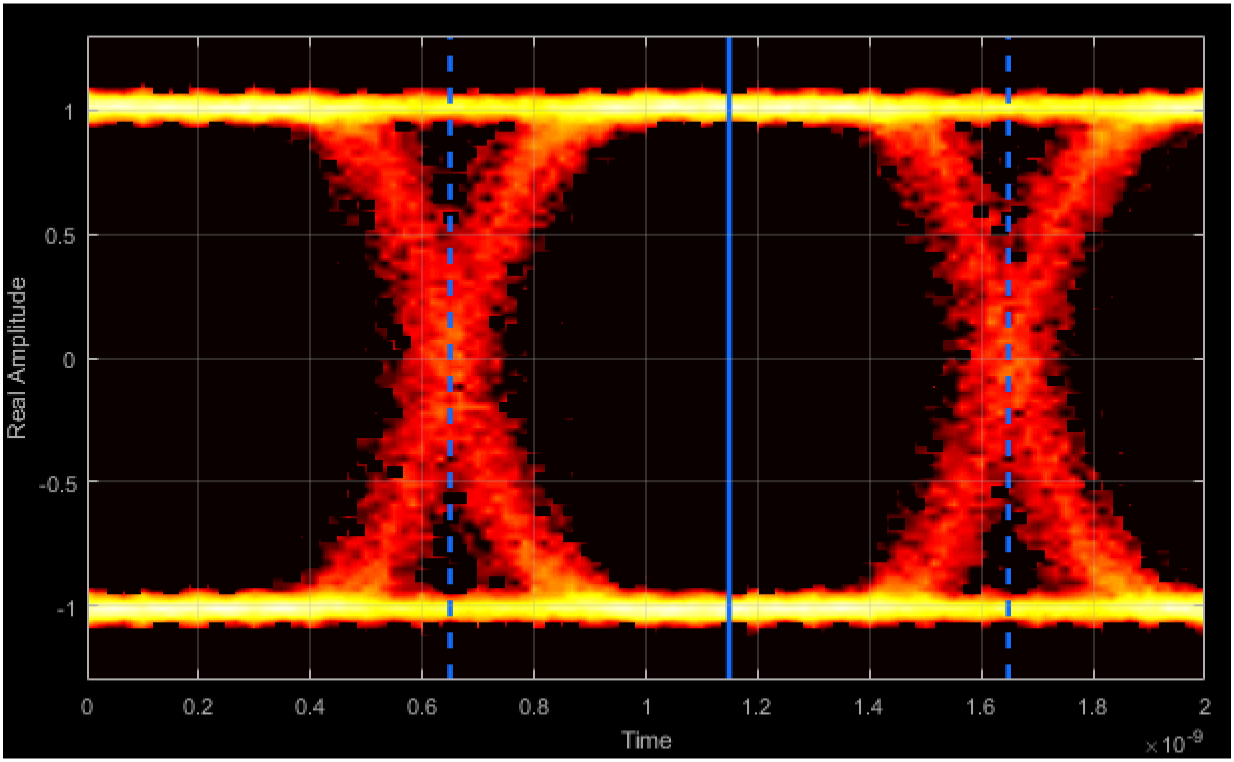
`Crossing Times` — Times for which crossings occur

The crossing times are the times at which the crossings occur. The times are computed as the mean values of the horizontal (jitter) histograms.



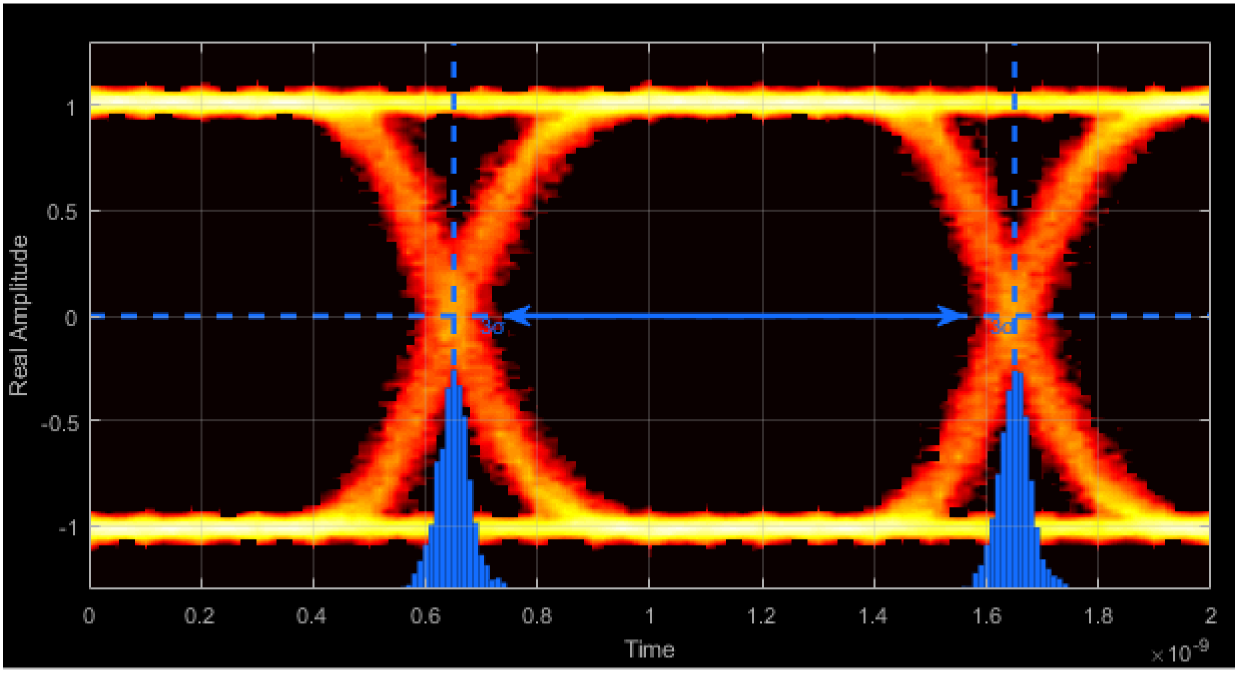
Eye Delay — Mean time between eye crossings

Eye delay is the midpoint between the two crossing times.



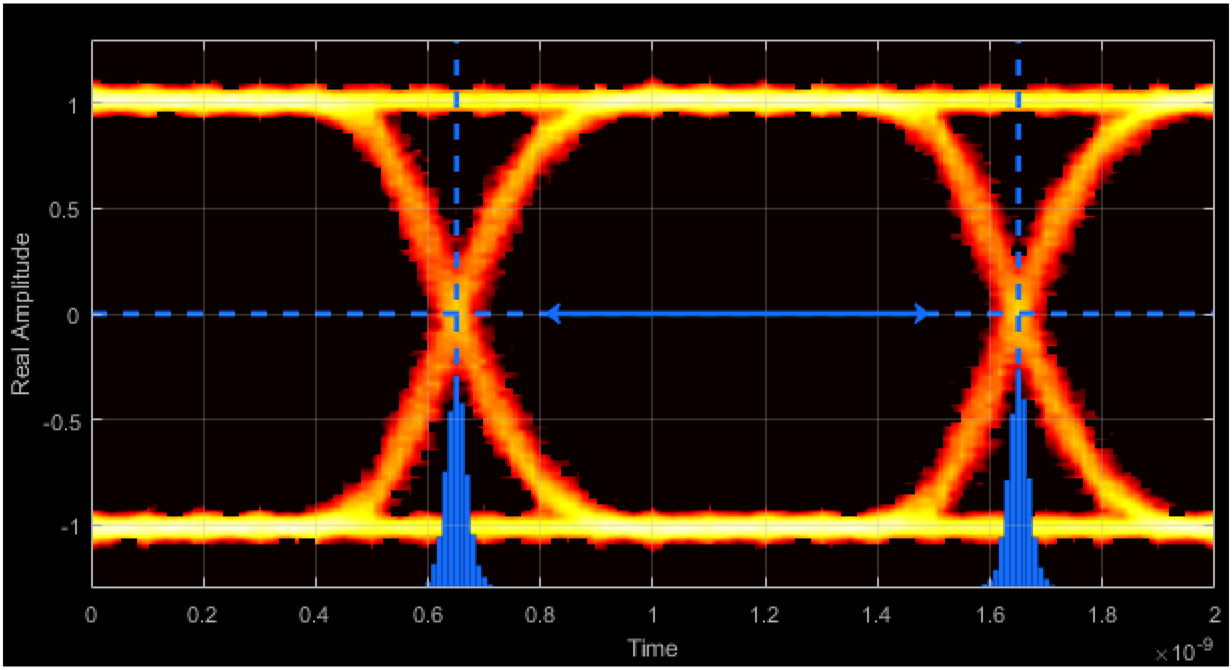
Eye Width — Statistical minimum time between eye crossings

Eye width is the horizontal distance between μ + 3σ of the left crossing time and μ – 3σ of the right crossing time. μ is the mean of the jitter histogram and σ is the standard deviation.



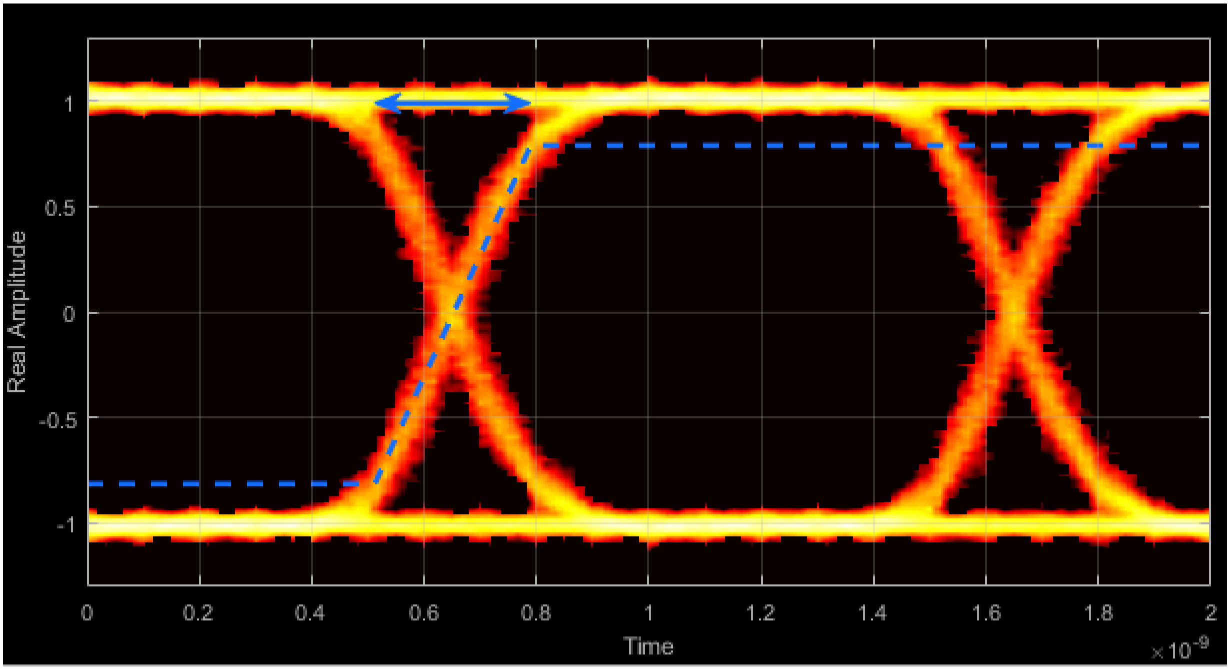
`Horizontal Opening` — Time between BER threshold points

The horizontal opening is the distance between the two points that correspond to the BER threshold. For example, for a 10^{-12} BER, these two points correspond to the 7σ distance from each crossing time.



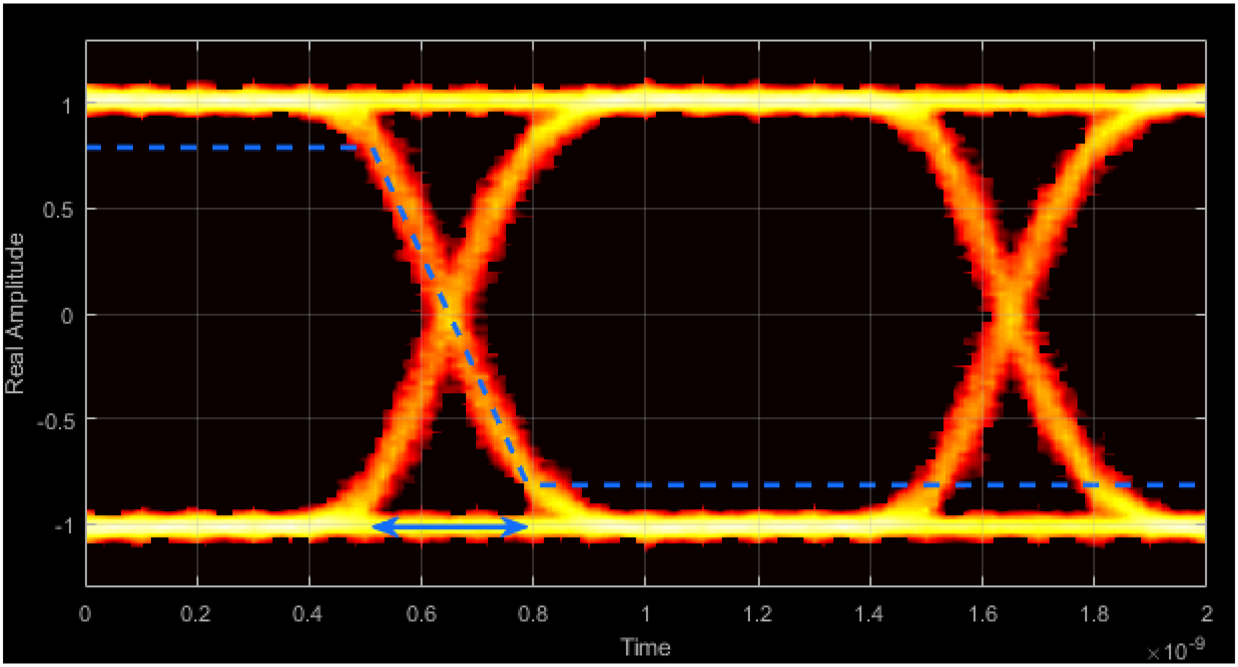
`Rise Time` — Time to transition from low to high

Rise time is the mean time between the low and high thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



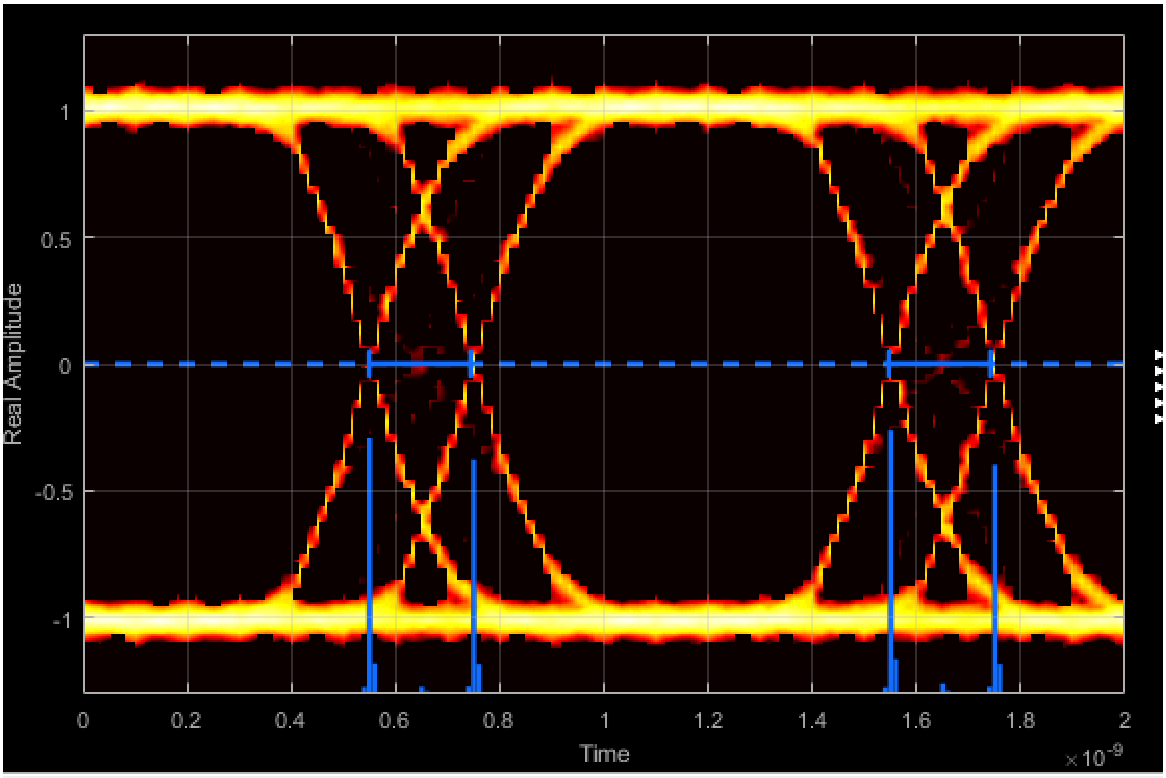
`Fall Time` — Time to transition from high to low

Fall time is the mean time between the high and low thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



`Deterministic Jitter` — Deterministic deviation from ideal signal timing

The deterministic jitter (DJ) is the distance between the two peaks of the dual-Dirac histograms. The probability density function (PDF) of DJ is composed of two delta functions.



`Random Jitter` — Random deviation from ideal signal timing

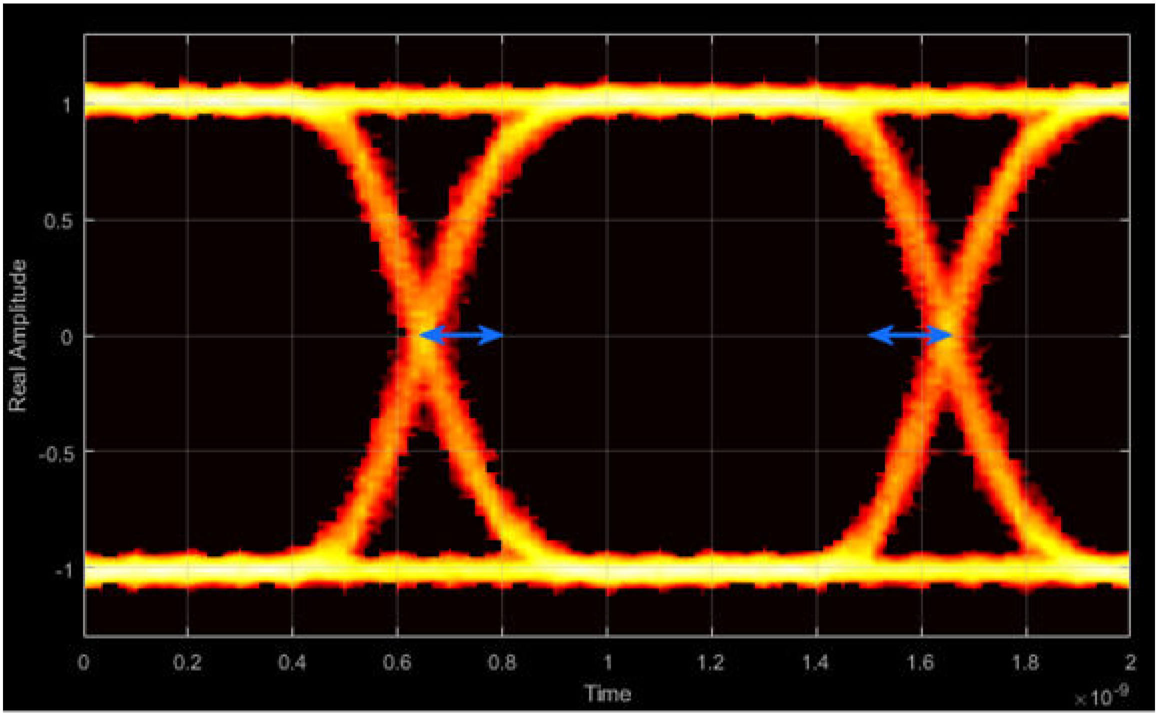
The random jitter (RJ) is the Gaussian unbounded jitter component. The random component of jitter is modeled as a zero-mean Gaussian random variable with a specified standard-deviation, σ. The random jitter is computed as:

$$\mathrm{RJ} = (Q_L + Q_R)\sigma \ ,$$

where

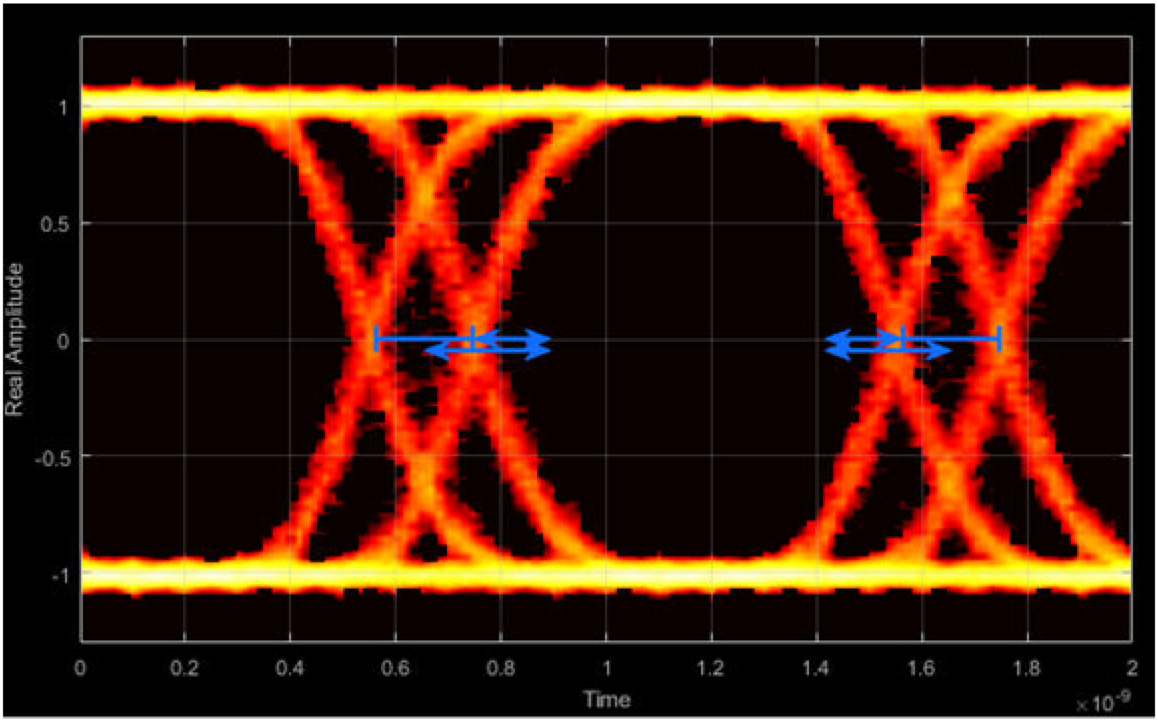
$$Q = \sqrt{2}\,\mathrm{erfc}^{-1}\left(2\frac{BER}{\rho}\right).$$

BER is the specified BER threshold. ρ is the amplitude of the left and right Dirac function, which is determined from the bin counts of the jitter histograms.

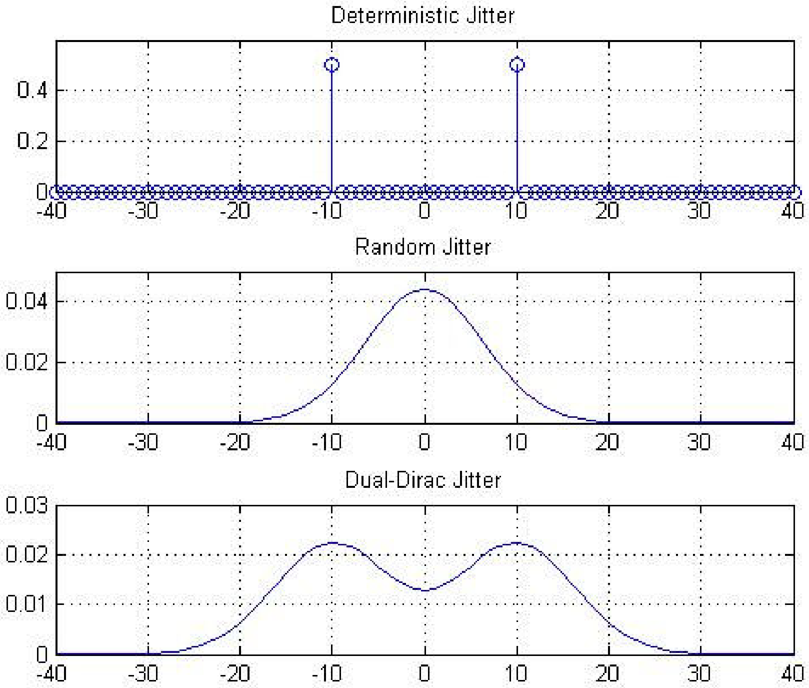


`Total Jitter` — Deviation from ideal signal timing

Total jitter (TJ) is the sum of the deterministic and random jitter, such that TJ = DJ + RJ.

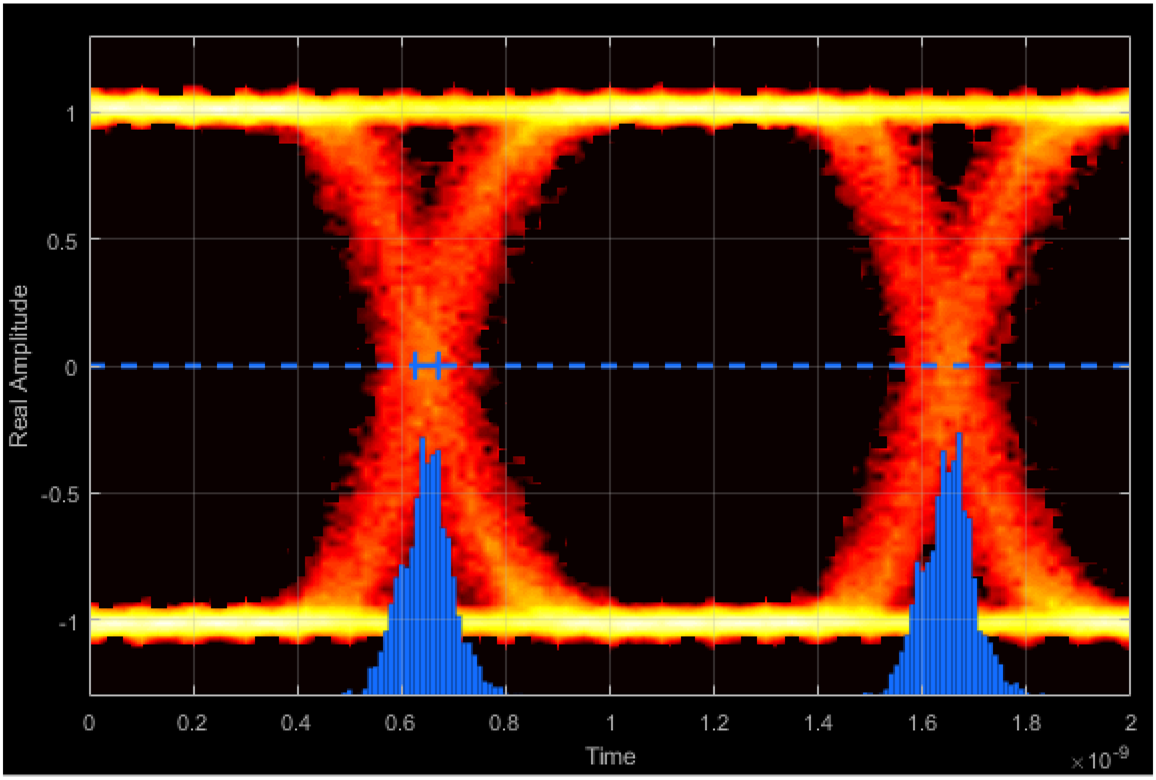


The total jitter PDF is the convolution of the DJ PDF and the RJ PDF.



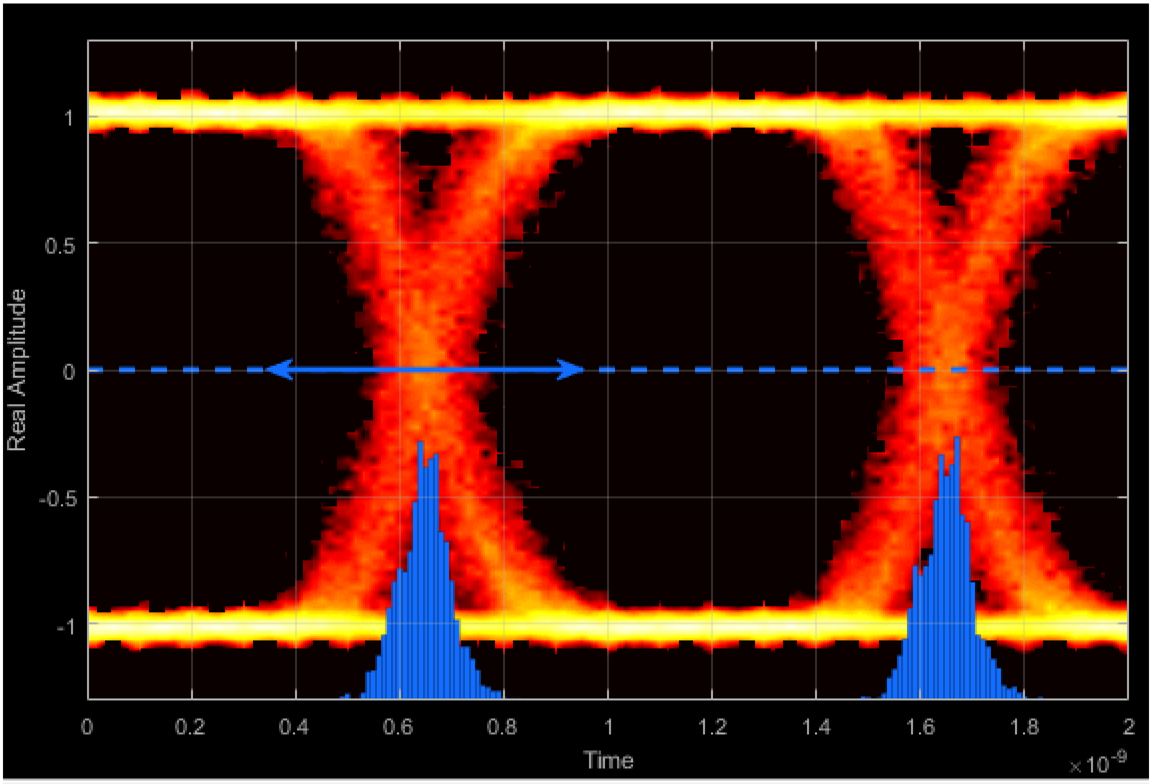
`RMS Jitter` — Standard deviation of jitter

RMS jitter is the standard deviation of the jitter calculated in the horizontal (jitter) histogram at the decision boundary.



`Peak-to-Peak Jitter` — Distance between extreme data points of histogram

Peak-to-peak jitter is the maximum horizontal distance between the left and right nonzero values in the horizontal histogram of each crossing time.



Programmatic Configuration

You can programmatically configure the scope properties with callbacks or within scripts by using a scope configuration object as describe in “Control Scopes Programmatically” (Simulink).

References

[1] Stephens, Ransom. "Jitter Analysis: The Dual-Dirac Model, RJ/DJ, and Q-Scale." Agilent Technologies Whitepaper. December 2004.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

System Objects
`comm.ConstellationDiagram`

Blocks
Eye Diagram

Functions
`eyediagram`

Introduced in R2016b

hide

System object: comm.EyeDiagram
Package: comm

Hide scope window

Syntax

```
hide(ed)
```

Description

`hide(ed)` hides the eye diagram window associated with System object `ed`.

See Also

`comm.EyeDiagram.show`

Introduced in R2016b

horizontalBathtub

System object: comm.EyeDiagram
Package: comm

Horizontal bathtub curve

Syntax

```
s = horizontalBathtub(ed)
```

Description

`s = horizontalBathtub(ed)` returns a structure array, `s`, for eye diagram object `ed`. Each structure in `s` contains a BER level and the corresponding left and right thresholds between jitter and horizontal eye opening.

Note This method is available when both of these conditions apply:

- `EnableMeasurements` is `true`
- `ShowBathtub` is `'Horizontal'` or `'Both'`

Examples

Horizontal and Vertical Bathtub Curve Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Generate and plot the horizontal and vertical bathtub curves.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'ShowBathtub','Both','YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-5e-04 5e-04],'RandomStd',2e-4);
```

Generate two symbols for each trace.

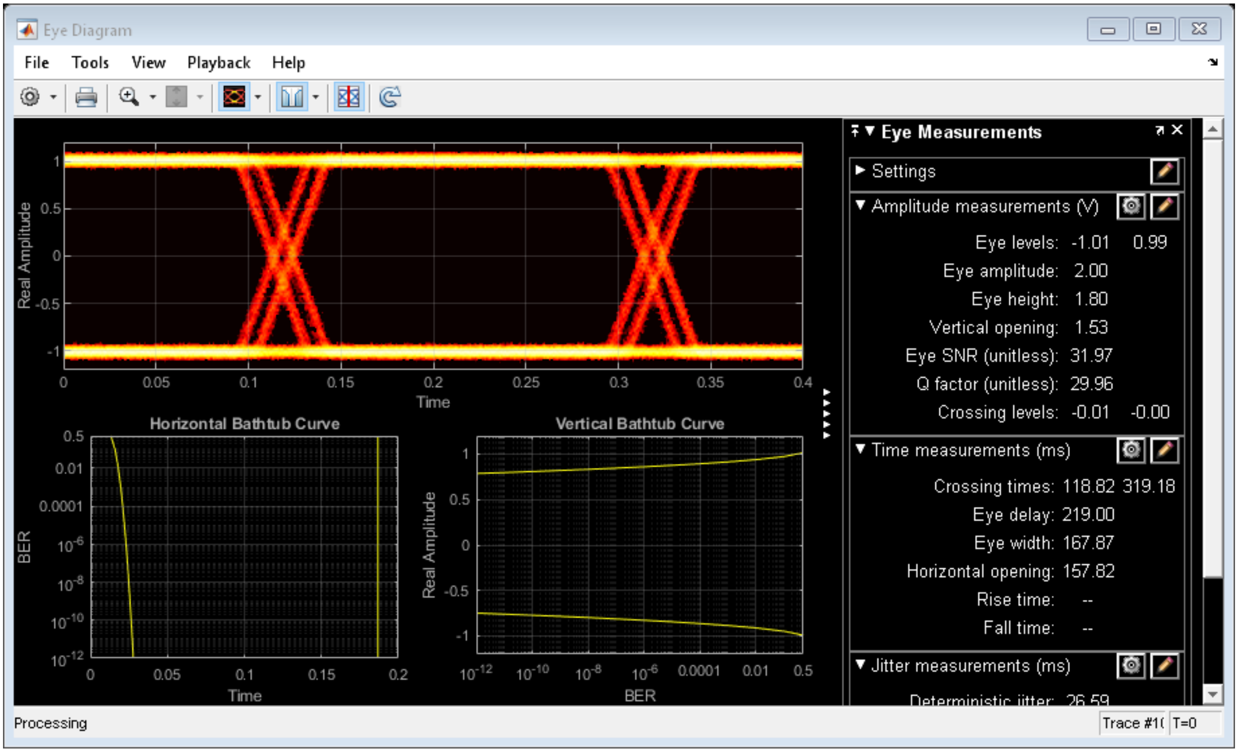
```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Display the eye diagram.

```
ed(y)
```



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.97
Q factor (unitless): 29.96
Crossing levels: -0.01 -0.00

Time measurements (ms)

Crossing times: 118.82 319.18
Eye delay: 219.00
Eye width: 167.87
Horizontal opening: 157.82
Rise time: --
Fall time: --

Jitter measurements (ms)

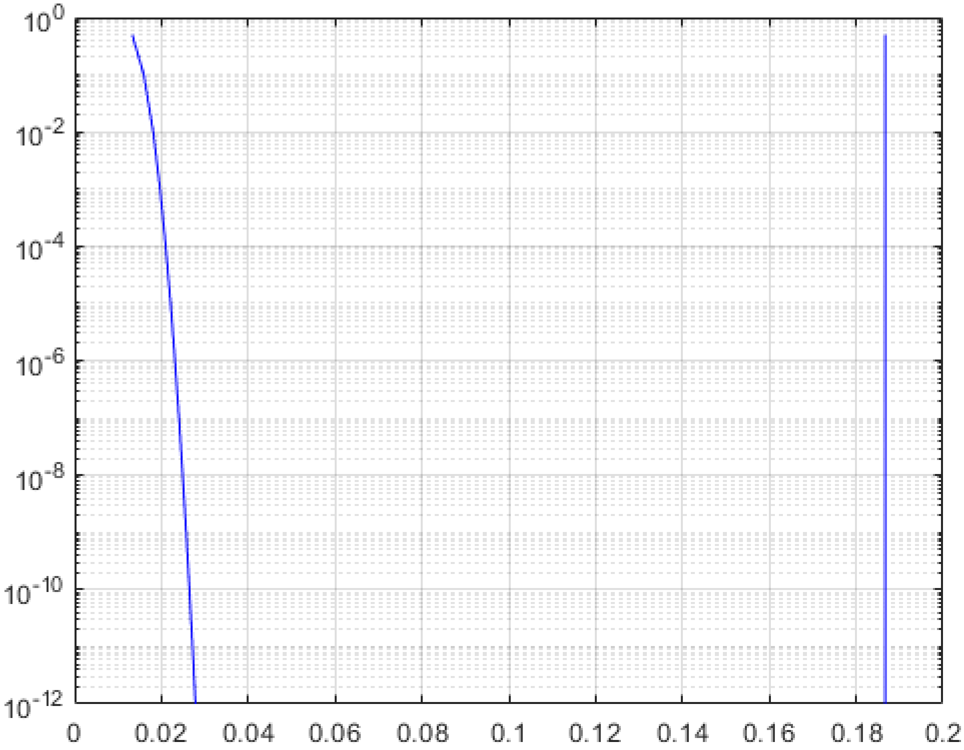
Deterministic jitter: 26.59

Generate the horizontal bathtub data for the eye diagram. Plot the curve.

```
hb = horizontalBathtub(ed)
```

```
hb = 1x13 struct array with fields:
    BER
    LeftThreshold
    RightThreshold
```

```
semilogy([hb.LeftThreshold],[hb.BER],'b',[hb.RightThreshold],[hb.BER],'b')
grid
```

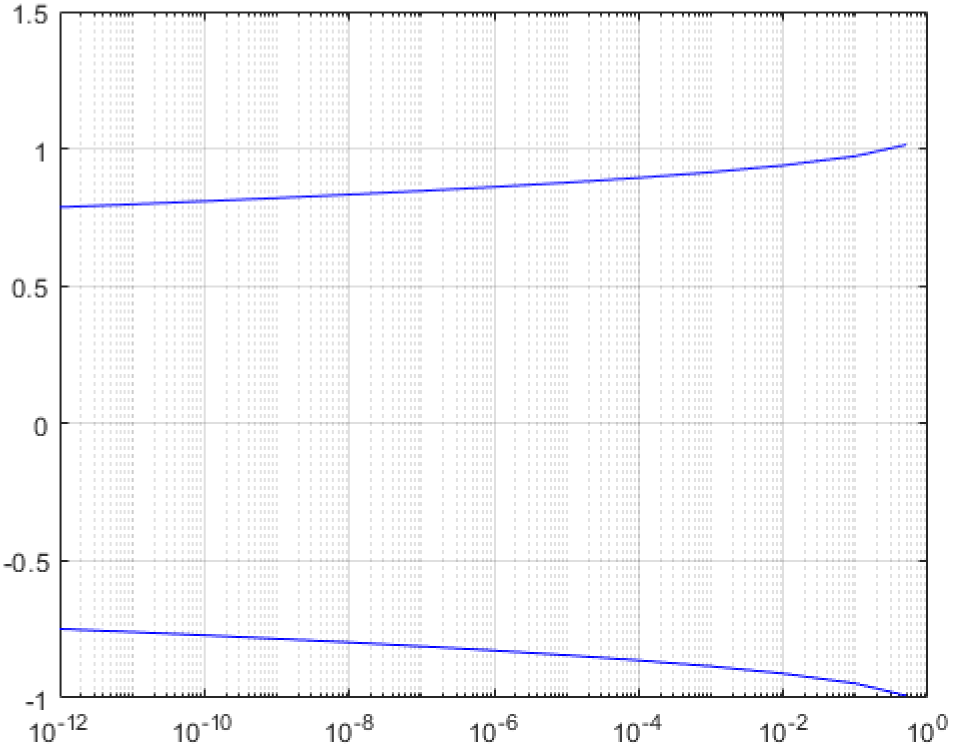


Generate the vertical bathtub data for the eye diagram. Plot the curve.

```
vb = verticalBathtub(ed)
```

```
vb = 1x13 struct array with fields:
    BER
    UpperThreshold
    LowerThreshold
```

```
semilogx([vb.BER],[vb.LowerThreshold],'b',[vb.BER],[vb.UpperThreshold],'b')
grid
```



Introduced in R2016b

jitterHistogram

System object: comm.EyeDiagram
Package: comm

Jitter histogram

Syntax

```
jh = jitterHistogram(ed)
```

Description

`jh = jitterHistogram(ed)` returns the bin counts for decision boundary crossings set in eye diagram object `ed`.

Note This method is available when `EnableMeasurements` is `true`.

Examples

Jitter and Noise Histogram Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Plot the jitter and noise histograms.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

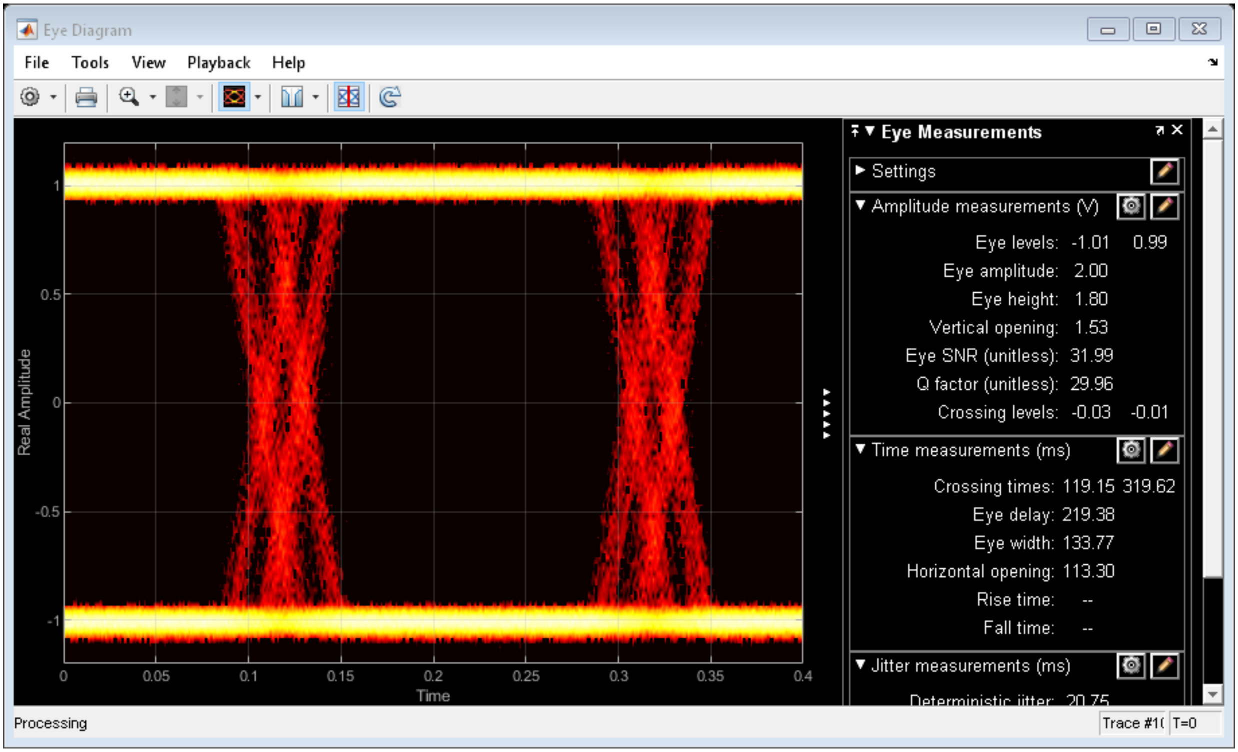
```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

Generate two symbols for each trace.

```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
ed(y)
```



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.99
Q factor (unitless): 29.96
Crossing levels: -0.03 -0.01

Time measurements (ms)

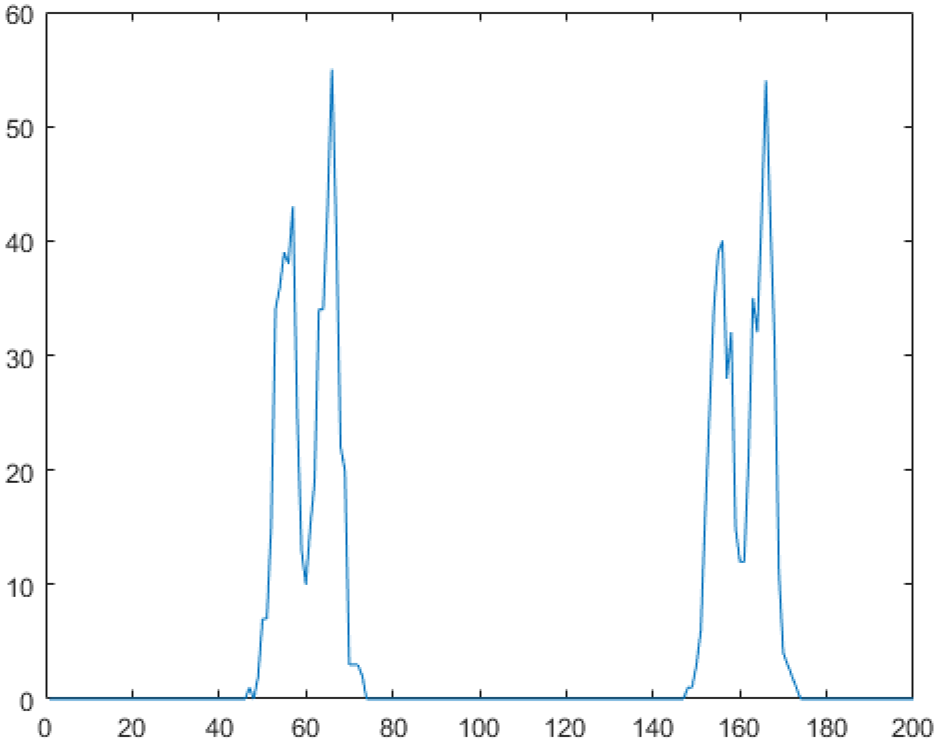
Crossing times: 119.15 319.62
Eye delay: 219.38
Eye width: 133.77
Horizontal opening: 113.30
Rise time: --
Fall time: --

Jitter measurements (ms)

Deterministic jitter: 20.75

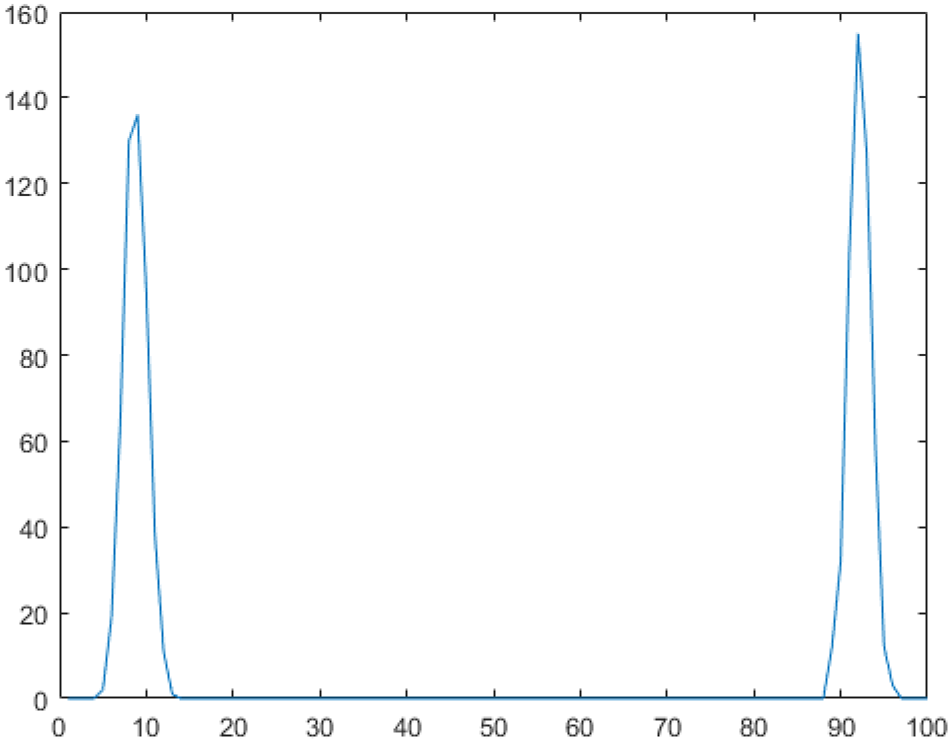
Calculate the jitter histogram count for each bin by using the `jitterHistogram` method. Plot the histogram.

```
jbins = jitterHistogram(ed);
plot(jbins)
```



Calculate the noise histogram count for each bin by using the `noiseHistogram` method. Plot the histogram.

```
nbins = noiseHistogram(ed);
plot(nbins)
```



Introduced in R2016b

measurements

System object: comm.EyeDiagram
Package: comm

Measure eye diagram parameters

Syntax

```
m = measurements(ed)
```

Description

`m = measurements(ed)` returns the amplitude, time, and jitter measurements calculated by eye diagram object `ed`.

Note This method is available when `EnableMeasurements` is `true`.

Examples

Rise and Fall Time of NRZ Signal

Create a combined jitter object having random jitter with a 2e-4 standard deviation.

```
jtr = commsrc.combinedjitter('RandomJitter','on','RandomStd',2e-4);
```

Generate an NRZ signal having random jitter and 3 ms rise and fall times.

```
genNRZ = commsrc.pattern('Jitter',jtr,'RiseTime',3e-3,'FallTime',3e-3);
x = generate(genNRZ,2000);
```

Pass the signal through an AWGN channel with fixed seed for repeatable results.

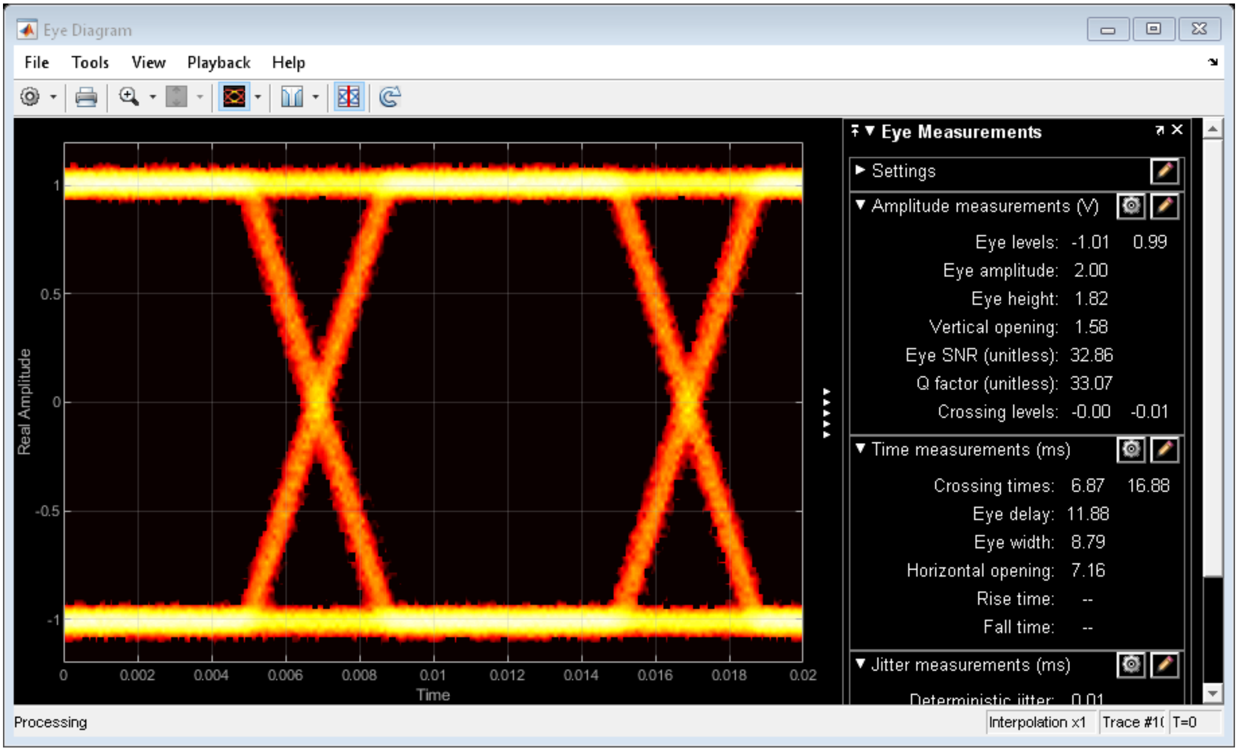
```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Create an eye diagram object. Enable the measurements.

```
ed = comm.EyeDiagram('SamplesPerSymbol',genNRZ.SamplesPerSymbol, ...
    'SampleRate',genNRZ.SamplingFrequency,'SampleOffset',genNRZ.SamplesPerSymbol/2, ..
    'EnableMeasurements',true,'DisplayMode','2D color histogram', ...
    'OversamplingMethod','Input interpolation','ColorScale','Logarithmic','YLimits',[-1
```

To compute the rise and fall times, determine the rise and fall thresholds from the eye level and eye amplitude measurements. Plot the eye diagram to calculate these parameters.

```
ed(y)
```



```
Eye Measurements
Settings
Amplitude measurements (V)
            Eye levels:  -1.01    0.99
         Eye amplitude:   2.00
            Eye height:   1.82
      Vertical opening:   1.58
    Eye SNR (unitless):  32.86
   Q factor (unitless):  33.07
       Crossing levels:  -0.00   -0.01
Time measurements (ms)
        Crossing times:   6.87   16.88
             Eye delay:  11.88
             Eye width:   8.79
    Horizontal opening:   7.16
             Rise time:   --
             Fall time:   --
Jitter measurements (ms)
   Deterministic jitter:  0.01
```

Pass the signal through the eye diagram object again to measure the rise and fall times.

```
ed(y)
hide(ed)
```

Display the data by using the `measurements` method.

```
eyestats = measurements(ed);
riseTime = eyestats.RiseTime
fallTime = eyestats.FallTime
```

```
riseTime =

    0.0030


fallTime =

    0.0030
```

The measured values match the 3 ms specification.

Introduced in R2016b

noiseHistogram

System object: comm.EyeDiagram
Package: comm

Noise histogram

Syntax

```
nh = noiseHistogram(ed)
```

Description

`nh = noiseHistogram(ed)` returns the bin counts for the signal values at the vertical opening (eye delay) as set in eye diagram object `ed`.

Note This method is available when both of these conditions apply:

- `EnableMeasurements` is `true`
- `DisplayMode` is `'2D color histogram'`

Examples

Jitter and Noise Histogram Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Plot the jitter and noise histograms.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

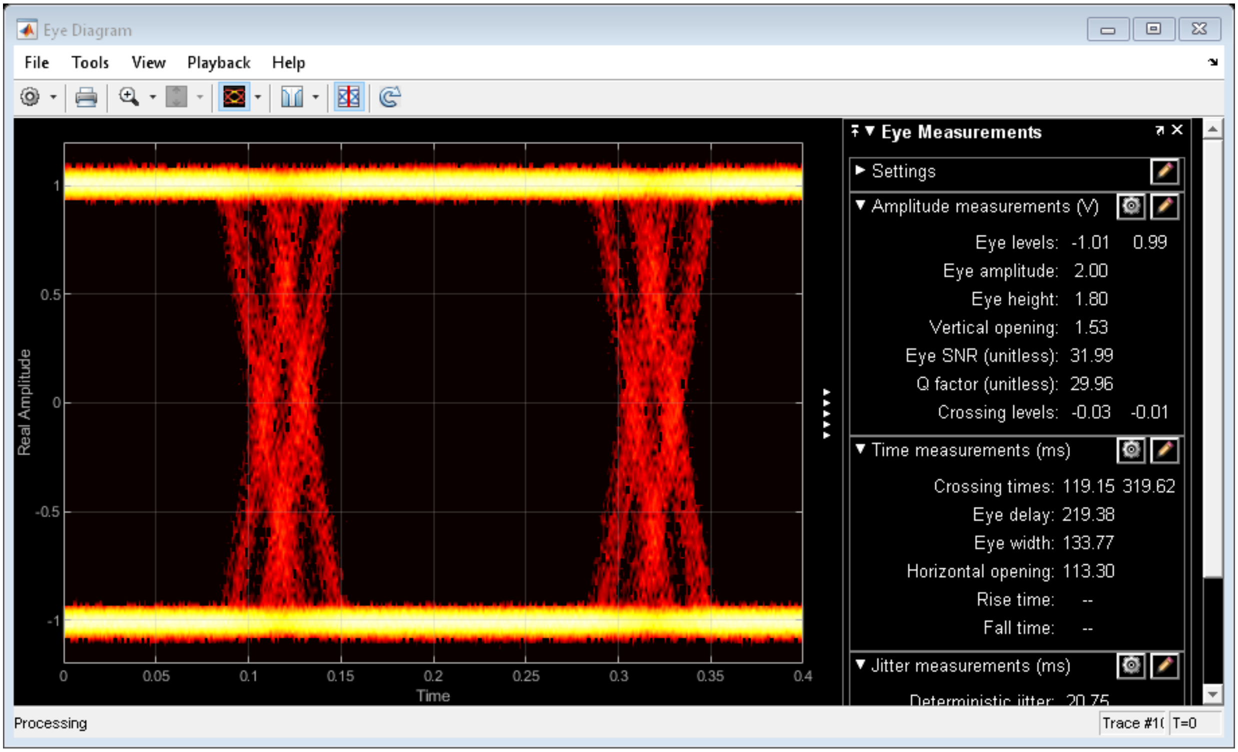
```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

Generate two symbols for each trace.

```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
ed(y)
```



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.99
Q factor (unitless): 29.96
Crossing levels: -0.03 -0.01

Time measurements (ms)

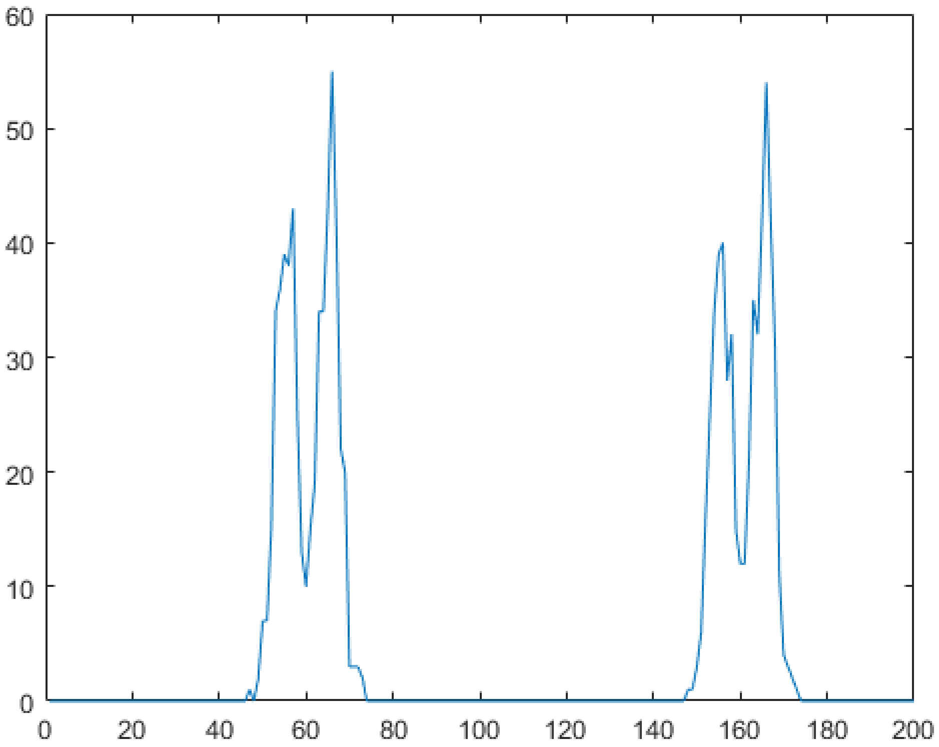
Crossing times: 119.15 319.62
Eye delay: 219.38
Eye width: 133.77
Horizontal opening: 113.30
Rise time: --
Fall time: --

Jitter measurements (ms)

Deterministic jitter: 20.75

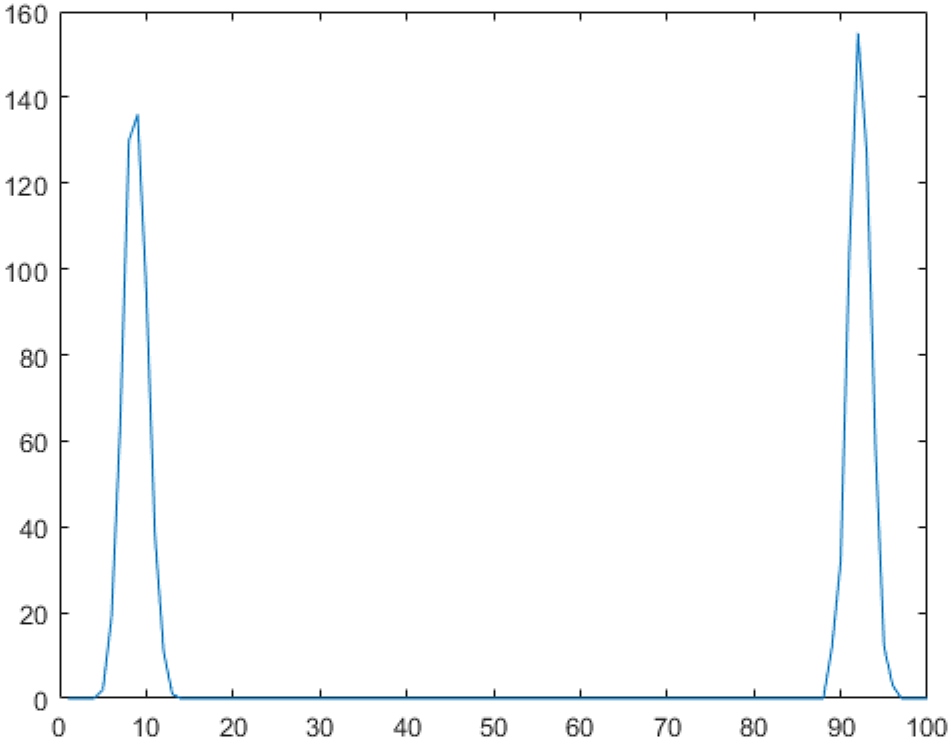
Calculate the jitter histogram count for each bin by using the `jitterHistogram` method. Plot the histogram.

```
jbins = jitterHistogram(ed);
plot(jbins)
```



Calculate the noise histogram count for each bin by using the `noiseHistogram` method. Plot the histogram.

```
nbins = noiseHistogram(ed);
plot(nbins)
```



Introduced in R2016b

reset

System object: comm.EyeDiagram
Package: comm

Reset states of eye diagram object

Syntax

```
reset(ed)
```

Description

`reset(ed)` resets the states of the `EyeDiagram` object, `ed`.

Introduced in R2016b

show

System object: comm.EyeDiagram
Package: comm

Make scope window visible

Syntax

`show(ed)`

Description

`show(ed)` makes the eye diagram window associated with System object `ed` visible.

See Also

`comm.EyeDiagram.hide`

Introduced in R2016b

step

System object: comm.EyeDiagram
Package: comm

Plot eye diagram of input signal

Syntax

```
step(ed,x)
ed(x)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(ed,x)` plots the eye diagram of input signal `x` using `comm.EyeDiagram` object `ed`.

`ed(x)` is equivalent to the first syntax.

Note `ed` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Examples

Eye Diagram of Filtered QPSK Signal

Specify the sample rate and the number of output samples per symbol parameters.

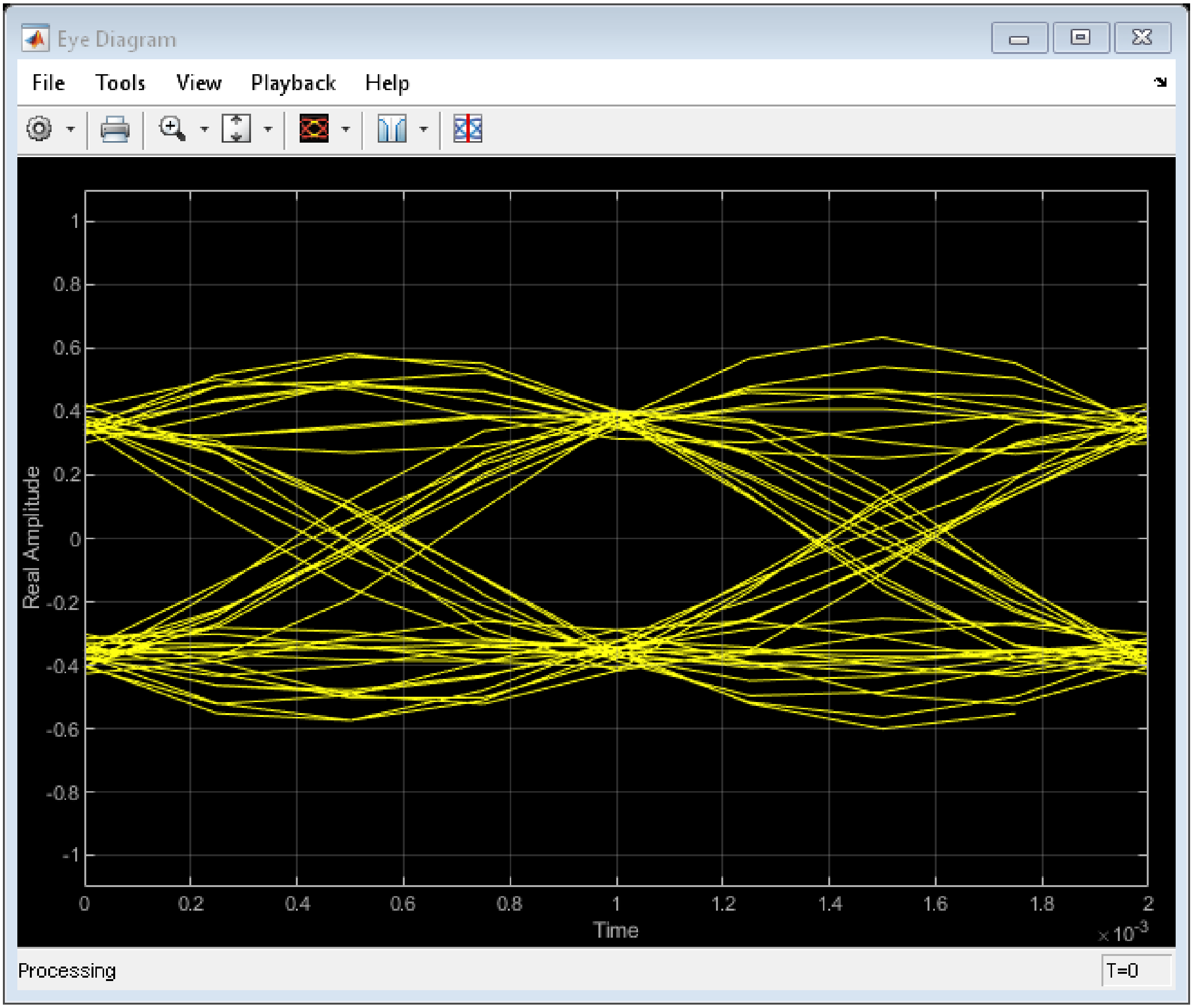
```
fs = 1000;
sps = 4;
```

Create transmit filter and eye diagram objects.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps);
ed = comm.EyeDiagram('SampleRate',fs*sps,'SamplesPerSymbol',sps);
```

Generate random symbols and apply QPSK modulation. Then filter the modulated signal and display the eye diagram.

```
data = randi([0 3],1000,1);
modSig = pskmod(data,4,pi/4);

txSig = txfilter(modSig);
ed(txSig)
```



Introduced in R2016b

verticalBathtub

System object: comm.EyeDiagram
Package: comm

Vertical bathtub curve

Syntax

```
s = verticalBathtub(ed)
```

Description

`s = verticalBathtub(ed)` returns a structure array, `s`, for eye diagram object `ed`. Each structure in `s` contains a BER level and the corresponding upper and lower thresholds between noise and vertical eye opening.

Note This method is available when both of these conditions apply:

- `EnableMeasurements` is `true`
- `ShowBathtub` is `'Vertical'` or `'Both'`

Examples

Horizontal and Vertical Bathtub Curve Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Generate and plot the horizontal and vertical bathtub curves.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps,'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'ShowBathtub','Both','YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps,'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on','DiracJitter','on', ...
    'DiracDelta',[-5e-04 5e-04],'RandomStd',2e-4);
```

Generate two symbols for each trace.

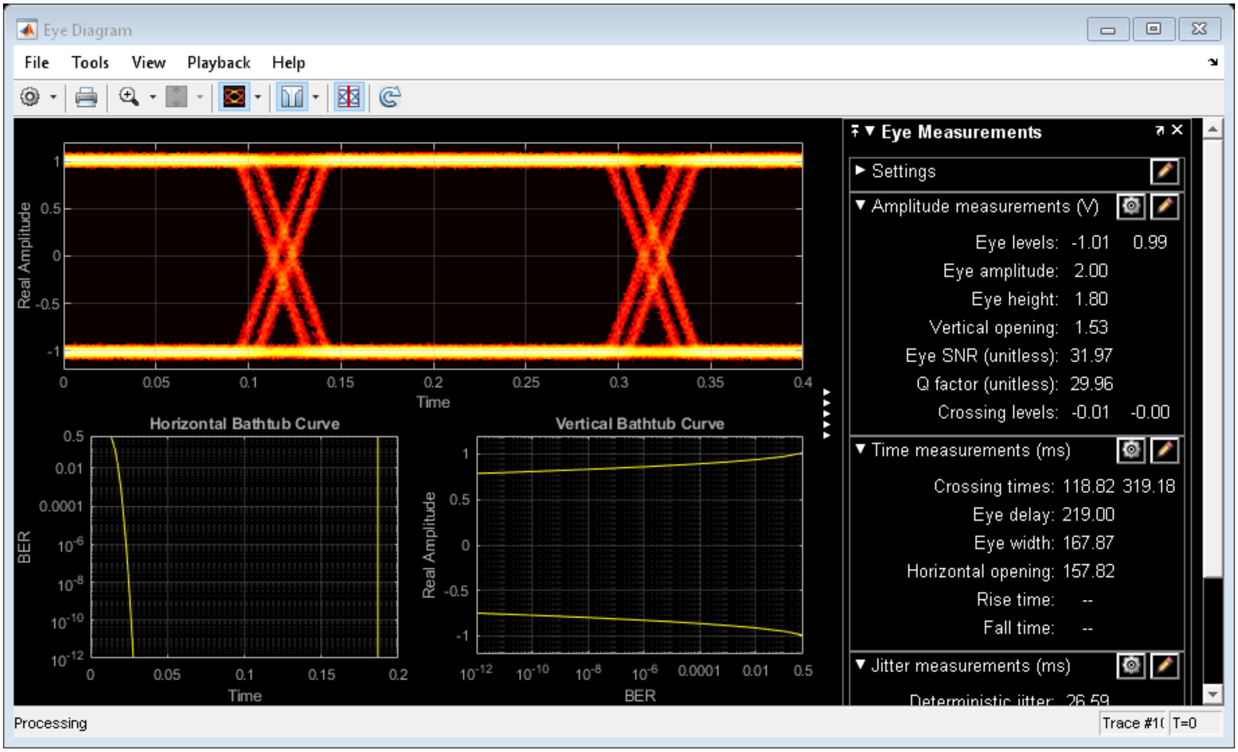
```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Display the eye diagram.

```
ed(y)
```



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

```
          Eye levels:  -1.01    0.99
       Eye amplitude:   2.00
          Eye height:   1.80
    Vertical opening:   1.53
   Eye SNR (unitless):  31.97
  Q factor (unitless):  29.96
     Crossing levels:  -0.01   -0.00
```

▼ Time measurements (ms)

```
     Crossing times: 118.82 319.18
          Eye delay: 219.00
          Eye width: 167.87
 Horizontal opening: 157.82
          Rise time:    --
          Fall time:    --
```

▼ Jitter measurements (ms)

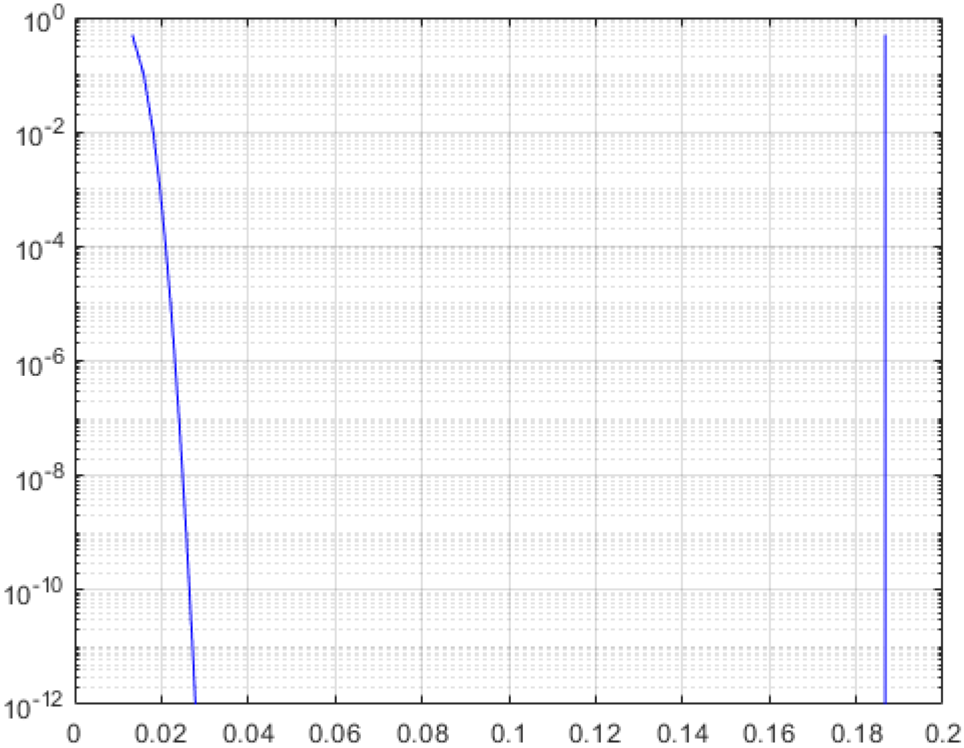
```
 Deterministic jitter: 26.59
```

Generate the horizontal bathtub data for the eye diagram. Plot the curve.

```
hb = horizontalBathtub(ed)
```

```
hb = 1x13 struct array with fields:
    BER
    LeftThreshold
    RightThreshold
```

```
semilogy([hb.LeftThreshold],[hb.BER],'b',[hb.RightThreshold],[hb.BER],'b')
grid
```

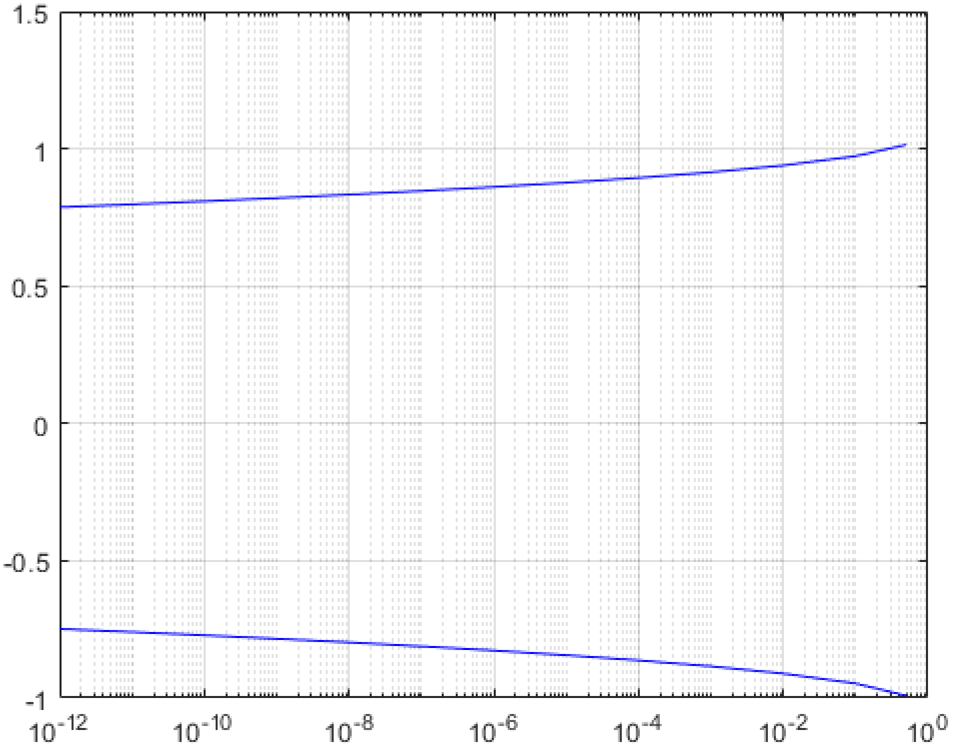


Generate the vertical bathtub data for the eye diagram. Plot the curve.

```
vb = verticalBathtub(ed)
```

```
vb = 1x13 struct array with fields:
    BER
    UpperThreshold
    LowerThreshold
```

```
semilogx([vb.BER],[vb.LowerThreshold],'b',[vb.BER],[vb.UpperThreshold],'b')
grid
```



Introduced in R2016b

comm.FMBroadcastDemodulator System object

Package: comm

Demodulate broadcast FM signal

Description

The `comm.FMBroadcastDemodulator` System object demodulates a complex baseband FM signal and filters the signal with a de-emphasis filter to produce an audio signal. If the `Stereo` property is set to `true`, the object performs stereo decoding. If the `RBDS` property is set to `true`, the object also demodulates the RDS/RBDS waveform. For more details, see “Algorithms” on page 3-601.

To demodulate a complex baseband FM signal:

1. Define and set up the `comm.FMBroadcastDemodulator` object. See “Construction” on page 3-592.
2. Call `step` to demodulate the complex baseband FM signal according to the properties of `comm.FMBroadcastDemodulator`.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fmbDemod = comm.FMBroadcastDemodulator` creates a demodulator System object, `fmbDemod`, that frequency demodulates an input signal.

`fmbDemod = comm.FMBroadcastDemodulator(Name,Value)` creates an FM demodulator object, `fmbDemod`, 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`).

`fmbDemod = comm.FMBroadcastDemodulator(MOD)` creates an FM demodulator object, `fmbDemod`, whose properties are determined by the corresponding FM modulator object, `MOD`.

Properties

`SampleRate`

Input signal sample rate (Hz)

Specify the sample rate of the input signal in Hz as a positive real scalar. The default value is `240e3`. This property is nontunable.

`FrequencyDeviation`

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM demodulator in Hz as a positive real scalar. The default value is `75e3`. System bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe. This property is nontunable.

`FilterTimeConstant`

Time constant of the de-emphasis filter (s)

Specify the de-emphasis lowpass filter time constant as a positive real scalar. The default value is `7.5e-05`. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe. This property is nontunable.

`AudioSampleRate`

Audio sample rate of the output signal (Hz)

Specify the output audio sample rate as a positive real scalar. The default value is `48000`. This property is nontunable.

PlaySound

Flag to enable or disable audio playback

To playback the output signal on the default audio device, set this property to `true`. The default is `false`. This property is nontunable.

BufferSize

Buffer size of the audio device

Specify the size of the buffer (in samples) that the object uses to communicate with an audio device as a positive scalar integer. The default is `4096`. This property is available only when `PlaySound` is `true`. This property is nontunable.

Stereo

Flag to enable or disable stereo audio

Set this property to `true` to demodulate a stereophonic audio signal. Set to `false` if the input signal is monophonic. The default is `false`. This property is nontunable.

RBDS

Flag to demodulate RDS/RBDS waveform

If `RBDS` is set to `true`, the second output of the `step` method is the baseband RDS/RBDS waveform. The default value is `false`. This property is nontunable.

RBDSSamplesPerSymbol

Oversampling factor of RDS/RBDS output

Specify the number of samples of the RDS/RBDS output as a positive integer. The RDS/RBDS sample rate is given by `RBDSSamplesPerSymbol × 1187.5` Hz. According to the RDS/RBDS standard, the sample rate of each bit is 1187.5 Hz.

This property applies only when you set `RBDS` to `true`.

The default is 10.

RBDSCostasLoop

Option to recover phase of RDS/RBDS signal

Specify whether a Costas loop is used to recover the phase of the RDS/RBDS signal. Set this option to `true` for radio stations that do not lock the `57` kHz RDS/RBDS signal in phase with the third harmonic of the `19` kHz pilot tone.

This property applies only when you set `RBDS` to `true`.

The default value is `false`.

Methods

info	Filter information about FM broadcast demodulator
reset	Reset states of the FM broadcast demodulator object
step	Apply FM broadcast demodulation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate and Demodulate a Streaming Audio Signal

Modulate and demodulate an audio signal with the FM broadcast modulator and demodulator objects. Plot the frequency responses of the input and demodulated signals.

Create an audio file reader System object™ and read the file `guitartune.wav`. Set the `SamplesPerFrame` property to include the entire file.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',44100);
x = audio();
```

Create spectrum analyzer objects to plot the spectra of the modulated and demodulated signals.

```
SAaudio = dsp.SpectrumAnalyzer('SampleRate',44100,'ShowLegend',true, ...
    'Title','Audio Signal', ...
    'ChannelNames',{'Input Signal' 'Demodulated Signal'});
```

```
SAfm = dsp.SpectrumAnalyzer('SampleRate',152e3, ...
    'Title','FM Broadcast Signal');
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',200e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate,'SampleRate',200e3);
```

Use the `info` method to determine the audio decimation factor of the filter in the modulator object. The length of the sequence input to the object must be an integer multiple of the object's decimation factor.

```
info(fmbMod)
```

```
ans = struct with fields:
       AudioDecimationFactor: 441
    AudioInterpolationFactor: 2000
        RBDSDecimationFactor: 19
     RBDSInterpolationFactor: 320
```

Use the `info` method to determine the audio decimation factor of the filter in the demodulator object.

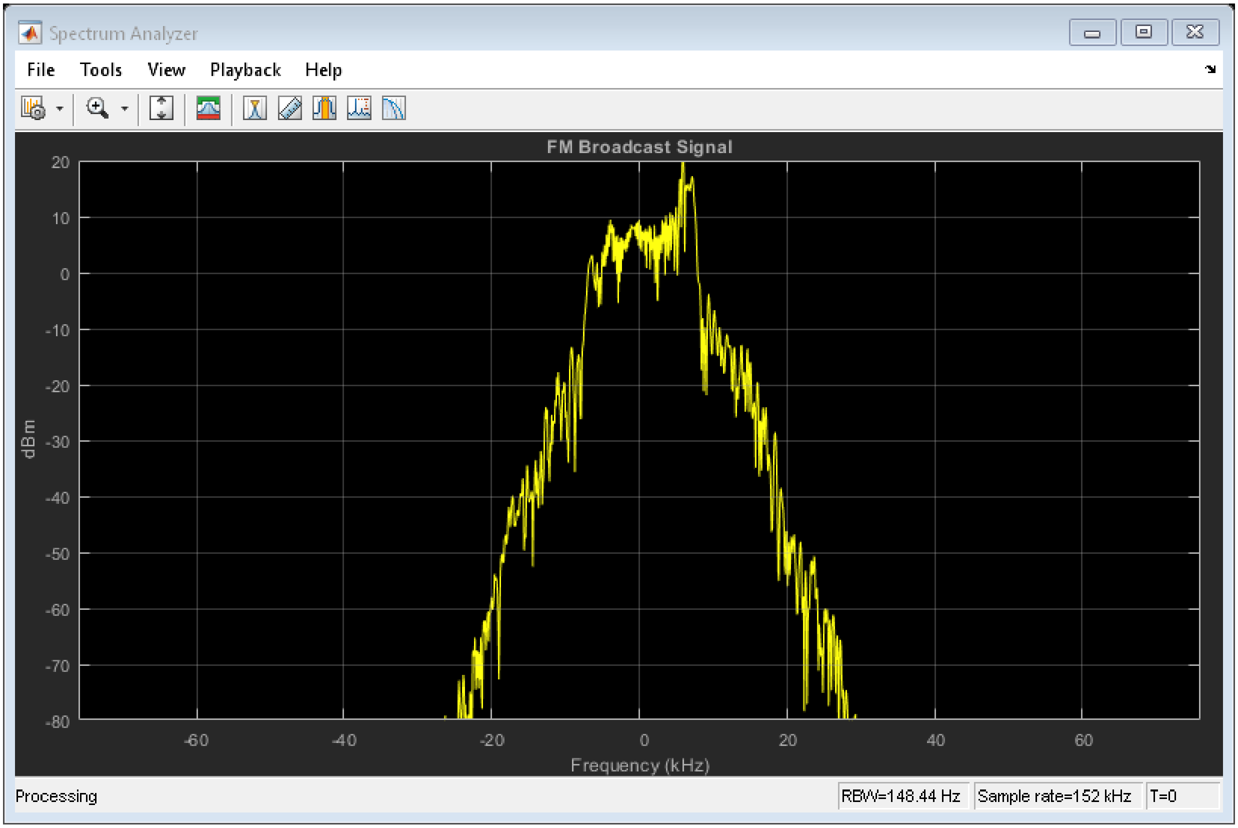
```
info(fmbDemod)
```

```
ans = struct with fields:
       AudioDecimationFactor: 50
    AudioInterpolationFactor: 57
        RBDSDecimationFactor: 50
     RBDSInterpolationFactor: 57
```

The audio decimation factor of the modulator is a multiple of the audio frame length of 44100. The audio decimation factor of the demodulator is an integer multiple of the 200000 samples data sequence length of the modulator output.

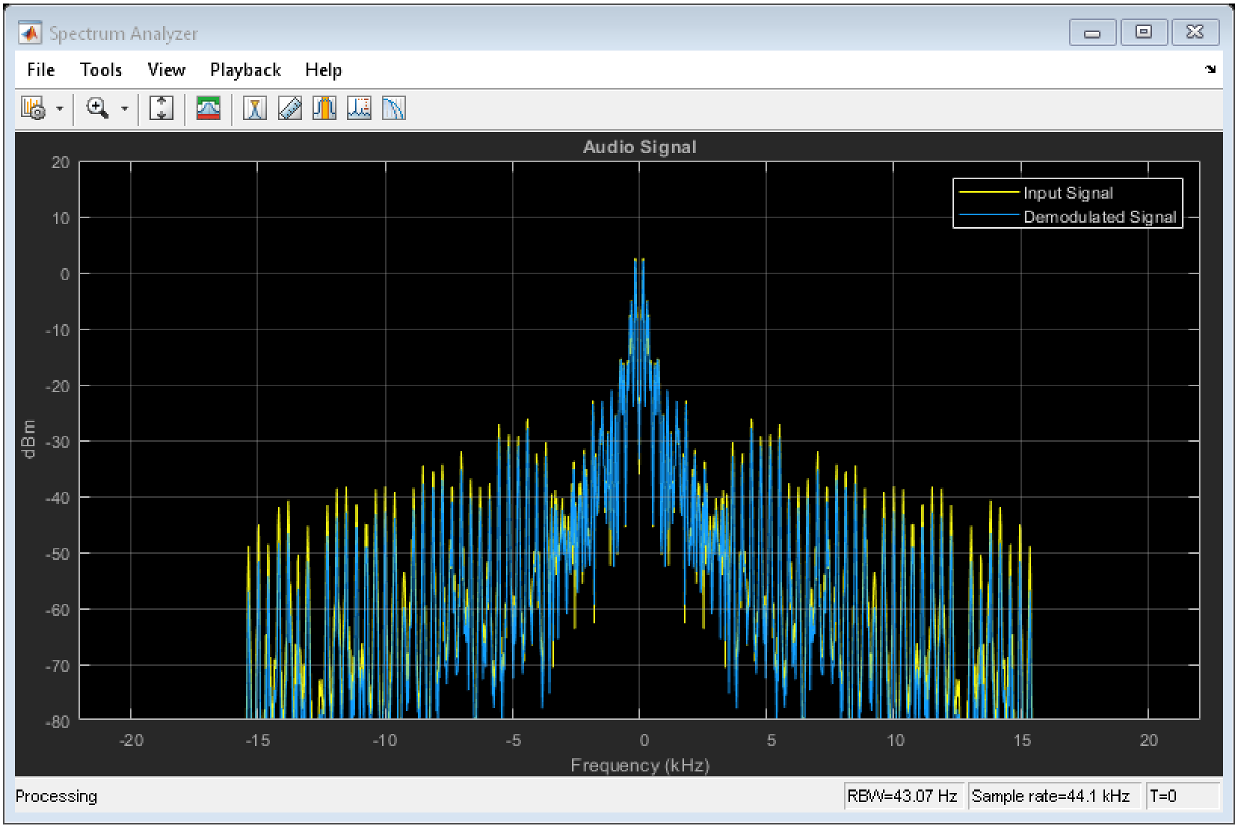
Modulate the audio signal and plot its spectrum.

```
y = fmbMod(x);
SAfm(y)
```



Demodulate y and plot the resultant spectrum. Compare the input signal spectrum with the demodulated signal spectrum. The spectra are similar except that demodulated signal has smaller high frequency components.

```
z = fmbDemod(y);
SAaudio([x z])
```



FM Broadcast a Streaming Audio Signal

Modulate and demodulate a streaming audio signal with the FM broadcast modulator and demodulator objects. Play the audio signal using a default audio device.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an audio file reader System object™ and read the file `guitartune.wav`.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',4410);
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator. Set the `PlaySound` property of the demodulator to `true` to enable audio playback.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3,'PlaySound',true);
```

Read the audio data in frames of length 4410, apply FM broadcast modulation, demodulate the FM signal and playback the audio input.

```
while ~isDone(audio)
    audioData = audio();
    modData = fmbMod(audioData);
    demodData = fmbDemod(modData);
end
```

FM Modulate and Demodulate an RBDS Waveform

Generate a basic RBDS waveform, FM modulate it with an audio signal, and then demodulate it.

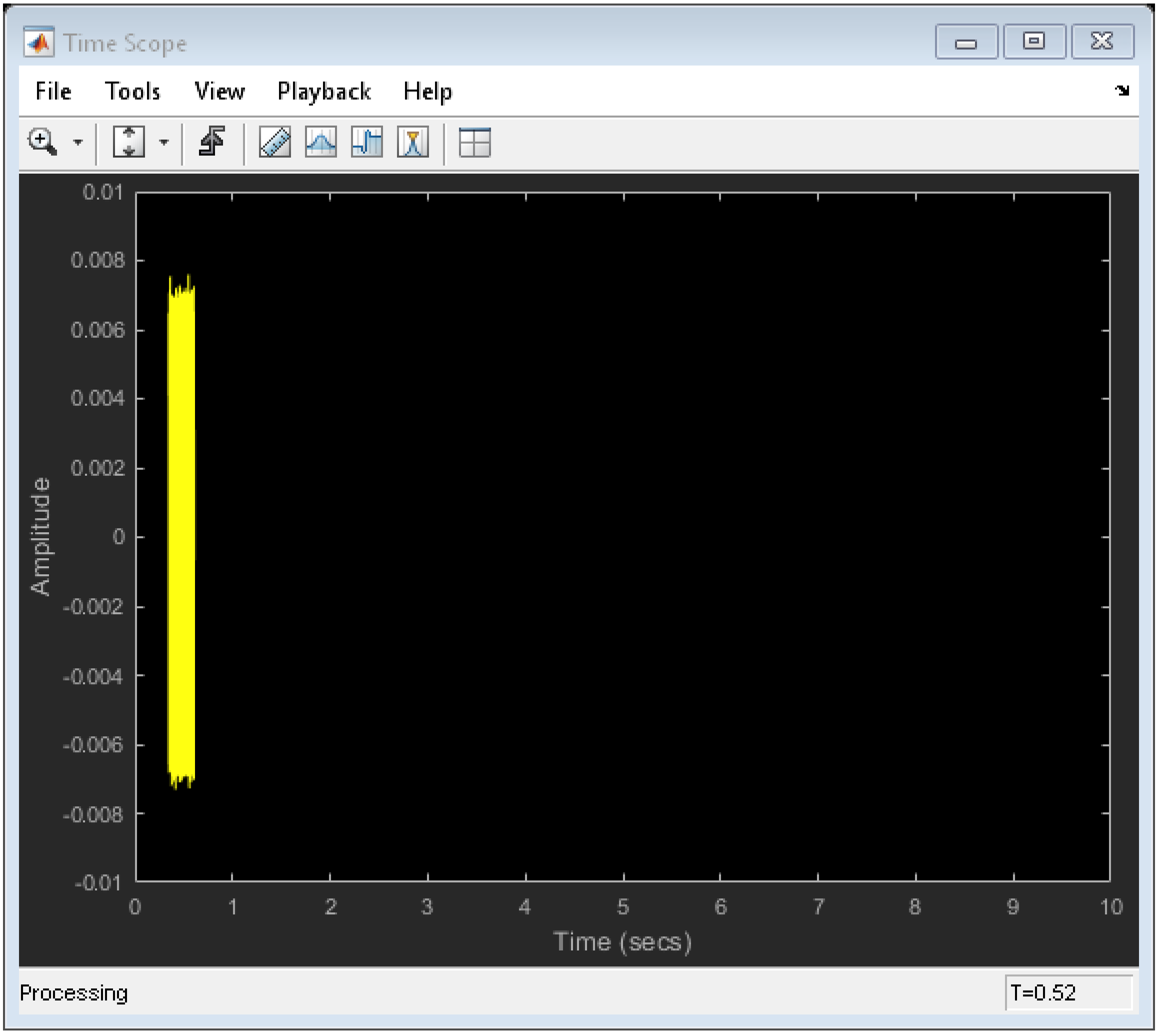
Note: This example runs only in R2017a or later.

Create a RBDS waveform with 19 groups per frame and 10 samples per symbol. The sample rate of the RBDS waveform is given by 1187.5 x 10. Set the audio sample rate to 1187.5 x 40.

```
groupLen = 104;
sps = 10;
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRa
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sp
fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = dsp.TimeScope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the current audio input. Generate RBDS information at the same configured rate as audio. FM modulate the stereo audio with RBDS information. Add additive white Gaussian noise. FM demodulate the audio and RBDS waveforms. View the waveforms in a time scope.

```
for idx = 1:7
    input = afr();
    rbdsWave = rbds();
    yFM = fmMod([input input], rbdsWave);
    rcv = awgn(yFM, 40);
    [audioRcv, rbdsRcv] = fmDemod(rcv);
    scope(rbdsRcv);
end
```

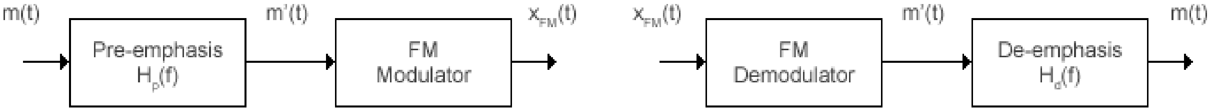


Algorithms

The FM Broadcast demodulator includes the functionality of the baseband FM demodulator, de-emphasis filtering, and the ability to receive stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMDemodulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s \ ,$$

where τ_s is the filter time constant. The time constant is 50 μs in Europe and 75 μs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s} \ .$$

For an audio sample rate of 44.1 kHz, the de-emphasis filter has the following response.

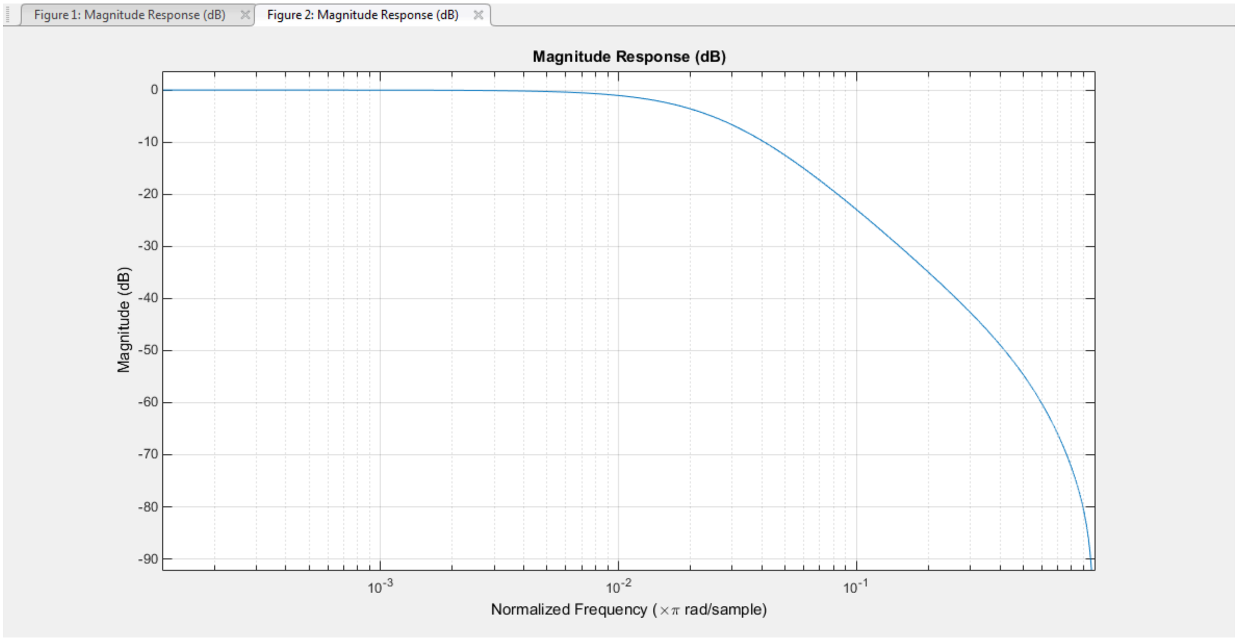
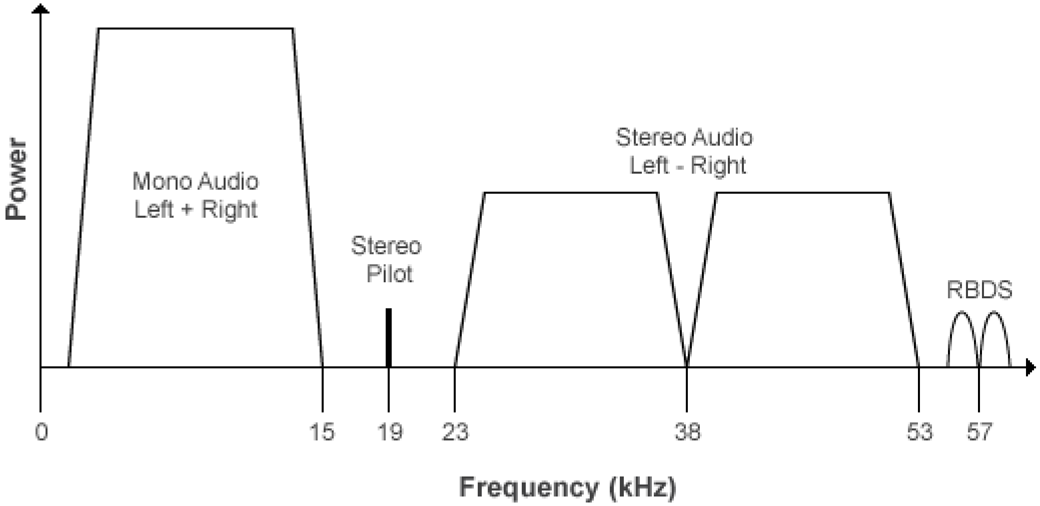


Figure 1: Magnitude Response (dB)
Figure 2: Magnitude Response (dB)
Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^{-3}
10^{-2}
10^{-1}
Normalized Frequency (×π rad/sample)

Stereo and RDS/RBDS FM — Multiplex Signal

The FM broadcast demodulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals.

Here is the spectrum of the multiplex baseband signal, *m(t)*.



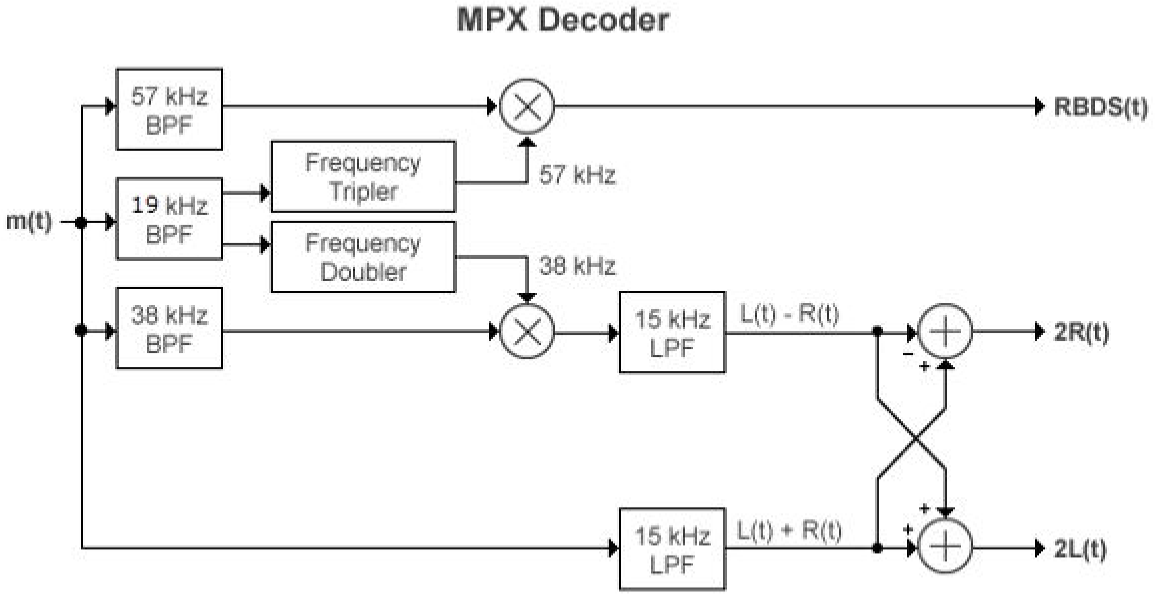
m(t) is given by

$$m(t) = C_0\left[L(t)+R(t)\right] + C_1\cos(2\pi\times 19kHz\times t) + C_0\left[L(t)-R(t)\right]\cos(2\pi\times 38kHz\times t) + C_2 RBDS(t)\cos(2$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

The demodulator applies *m(t)* to three bandpass filters with center frequencies at 19, 38, and 57 kHz, and to a lowpass filter with a 3-dB cutoff frequency of 15 kHz. The 19 kHz bandpass filter extracts the pilot tone from the modulated signal. The recovered pilot tone is doubled and tripled in frequency to produce the 38 kHz and 57 kHz signals, which demodulate the (*L* – *R*) and RDS/RBDS signals, respectively. To generate a scaled version of the left and right channels that produce the stereo sound, the (*L* + *R*) and (*L* – *R*) signals are added and subtracted. The RDS/RBDS signal is recovered by mixing with the 57 kHz signal.

Here is the block diagram of the FM broadcast demodulator.



Limitations

The input length must be an integer multiple of the `AudioDecimationFactor` property. If `RBDS` is set to `true`, the input length in addition must be an integer multiple of `RBDSDecimationFactor`. For more information on these two properties, see the `info` method.

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.FMBroadcastModulator` | `comm.FMDemodulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Blocks
FM Broadcast Demodulator Baseband | FM Broadcast Modulator Baseband

Introduced in R2015a

info

System object: comm.FMBroadcastDemodulator
Package: comm

Filter information about FM broadcast demodulator

Syntax

```
S = info(fmbDemod)
```

Description

`S = info(fmbDemod)` returns a structure, `S`, containing this information for the `comm.FMBroadcastDemodulator` System object, `fmbDemod`:

Field	Description
`AudioDecimationFactor`	Decimation factor of the audio demodulator filter.
`AudioInterpolationFactor`	Interpolation factor of the audio demodulator filter.
`RBDSDecimationFactor`	Decimation factor of the RDS/RBDS demodulator filter.
`RBDSInterpolationFactor`	Interpolation factor of the RDS/RBDS demodulator filter.

Note When `RBDS` is `true`, the demodulator input sequence length must be a multiple of `AudioDecimationFactor` and `RBDSDecimationFactor`.

When `RBDS` is `false`, the demodulator input sequence length must be a multiple of `AudioDecimationFactor`.

Introduced in R2015a

reset

System object: comm.FMBroadcastDemodulator
Package: comm

Reset states of the FM broadcast demodulator object

Syntax

```
reset(fmbDemod)
```

Description

`reset(fmbDemod)` resets the states of the `comm.FMBroadcastDemodulator` object, `fmbDemod`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

Introduced in R2015a

step

System object: comm.FMBroadcastDemodulator
Package: comm

Apply FM broadcast demodulation

Syntax

```
audioSig = step(fmbDemod,X)
[audioSig,rbdsSig] = step(fmbDemod,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`audioSig = step(fmbDemod,X)` demodulates the complex baseband FM signal, `X`, and filters this signal with a de-emphasis filter to produce an audio signal, `audioSig`. If the `Stereo` property is set to `true`, stereo decoding is also performed. The output, `audioSig`, is a real vector with length equal to `(AudioSampleRate/SampleRate) × length(X)`.

`[audioSig,rbdsSig] = step(fmbDemod,X)` also demodulates the baseband RBDS signal, `rbdsSig`. The `step` method outputs the RBDS signal only if the `RBDS` property is set to `true`. The output, `rbdsSig`, is a real vector with length equal to `(RBDSSamplesPerSymbol × 1187.5/SampleRate) × length(X)`.

Note `fmbDemod` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2015a

comm.FMBroadcastModulator System object

Package: comm

Modulate broadcast FM signal

Description

The `comm.FMBroadcastModulator` System object pre-emphasizes an audio signal and modulates it onto a baseband FM signal. If the `Stereo` property is set to `true`, the object modulates the audio input (*L*–*R*) in the 38 kHz band, in addition to modulating it in the baseband (*L*+*R*). If the `RBDS` property is set to `true`, the object modulates a baseband RDS/RBDS signal at 57 kHz. For more details, see “Algorithms” on page 3-619.

To FM modulate an audio signal:

1. Define and set up the `comm.FMBroadcastModulator` object. See “Construction” on page 3-611.
2. Call `step` to apply broadcast FM modulation to an audio signal according to the properties of `comm.FMBroadcastModulator`.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fmbMod = comm.FMBroadcastModulator` creates a modulator System object, `fmbMod`, that frequency modulates an input signal.

`fmbMod = comm.FMBroadcastModulator(demod)` creates a broadcast FM modulator object whose properties are determined by the corresponding broadcast FM demodulator object, `demod`.

`fmbMod = comm.FMBroadcastModulator(Name,Value)` creates a broadcast FM modulator object 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

SampleRate

Output signal sample rate (Hz)

Specify the sample rate of the output signal in Hz as a positive real scalar. The default value is 240e3. This property is nontunable.

FrequencyDeviation

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM modulator in Hz as a positive real scalar. The default value is 75e3. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe. This property is nontunable.

FilterTimeConstant

Filter time constant (s)

Specify the pre-emphasis highpass filter time constant as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe. The default value is 7.5e-05. The property is nontunable.

AudioSampleRate

Sample rate of the input audio signal (Hz)

Specify the audio sample rate as a positive real scalar. The default value is 48000. This property is nontunable.

Stereo

Flag to set stereo operations

Set this property to true if the input is a stereophonic audio signal. Set to false if the input signal is monophonic. The default is false. This property is nontunable.

RBDS

Flag to modulate RDS/RBDS waveform

If RBDS is set to true, the step method accepts the baseband RDS/RBDS waveform as its second input and the object modulates the signal at 57 kHz. The default value is false. This property is nontunable.

RBDSSamplesPerSymbol

Oversampling factor of RDS/RBDS input

Specify the number of samples per RDS/RBDS symbol as a positive integer. The RDS/RBDS sample rate is given by RBDSSamplesPerSymbol × 1187.5 Hz. According to the RDS/RBDS standard, the sample rate of each bit is 1187.5 Hz.

This property applies only when you set RBDS to true.

The default is 10.

Methods

info	Filter information about FM broadcast modulator
reset	Reset states of the FM broadcast modulator object
step	Apply FM broadcast modulation

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate and Demodulate a Streaming Audio Signal

Modulate and demodulate an audio signal with the FM broadcast modulator and demodulator objects. Plot the frequency responses of the input and demodulated signals.

Create an audio file reader System object™ and read the file `guitartune.wav`. Set the `SamplesPerFrame` property to include the entire file.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',44100);
x = audio();
```

Create spectrum analyzer objects to plot the spectra of the modulated and demodulated signals.

```
SAaudio = dsp.SpectrumAnalyzer('SampleRate',44100,'ShowLegend',true, ...
    'Title','Audio Signal', ...
    'ChannelNames',{'Input Signal' 'Demodulated Signal'});
SAfm = dsp.SpectrumAnalyzer('SampleRate',152e3, ...
    'Title','FM Broadcast Signal');
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',200e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate,'SampleRate',200e3);
```

Use the `info` method to determine the audio decimation factor of the filter in the modulator object. The length of the sequence input to the object must be an integer multiple of the object's decimation factor.

```
info(fmbMod)
```

```
ans = struct with fields:
       AudioDecimationFactor: 441
    AudioInterpolationFactor: 2000
        RBDSDecimationFactor: 19
     RBDSInterpolationFactor: 320
```

Use the `info` method to determine the audio decimation factor of the filter in the demodulator object.

```
info(fmbDemod)
```

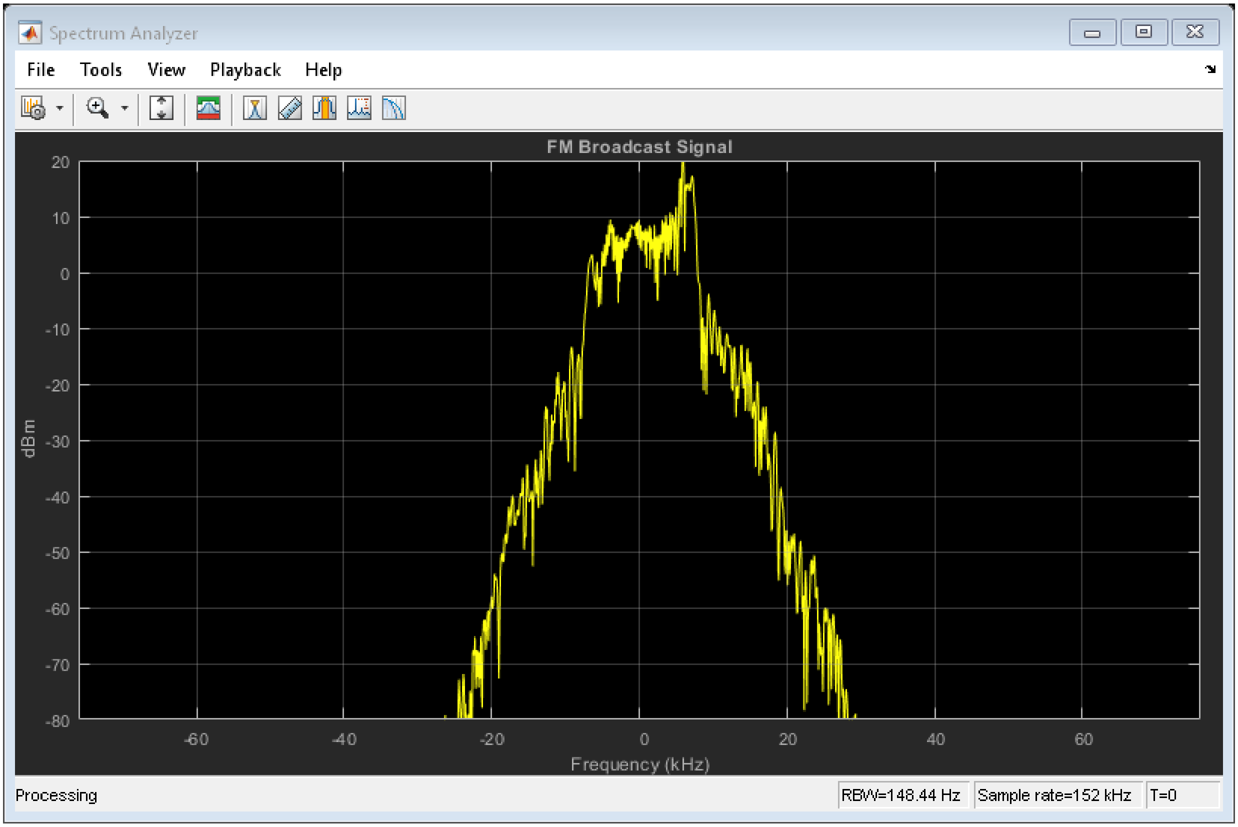
```
ans = struct with fields:
       AudioDecimationFactor: 50
    AudioInterpolationFactor: 57
```

```
      RBDSDecimationFactor: 50
    RBDSInterpolationFactor: 57
```

The audio decimation factor of the modulator is a multiple of the audio frame length of 44100. The audio decimation factor of the demodulator is an integer multiple of the 200000 samples data sequence length of the modulator output.

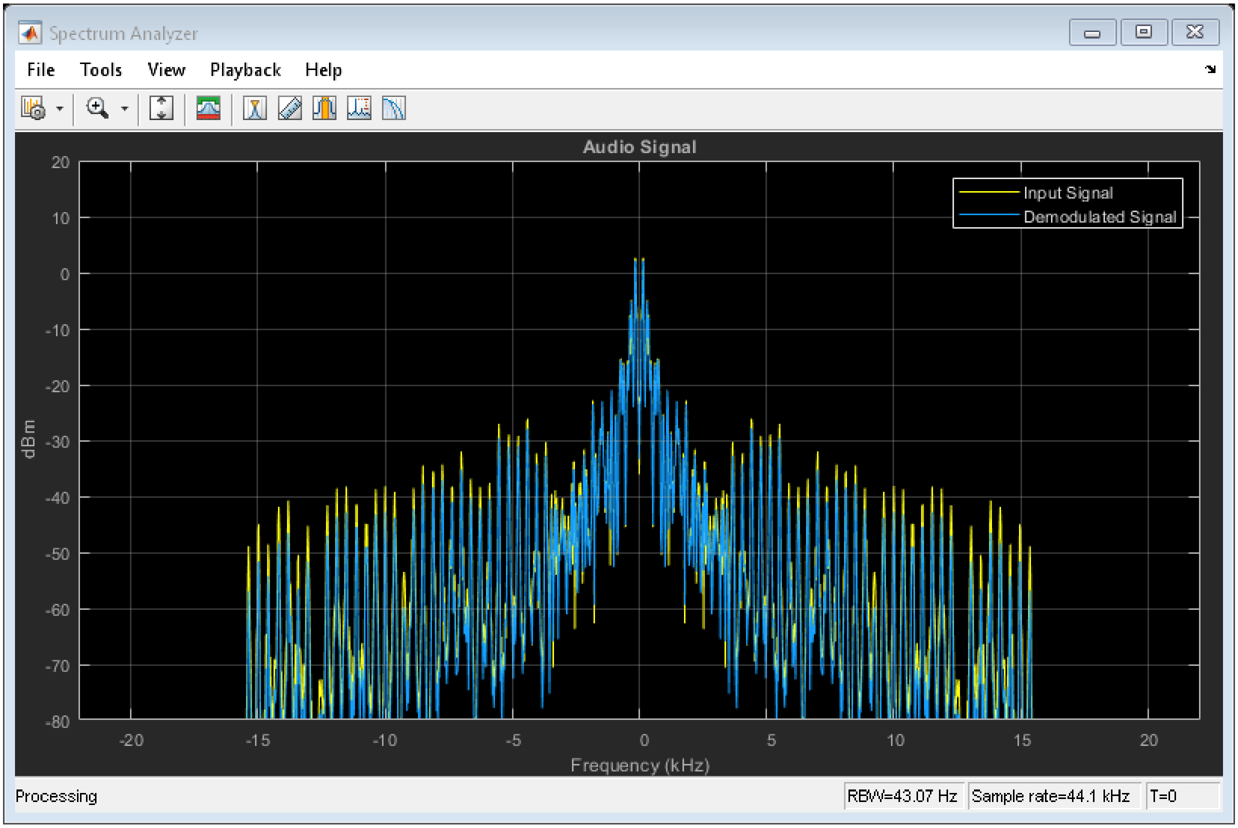
Modulate the audio signal and plot its spectrum.

```
y = fmbMod(x);
SAfm(y)
```



Demodulate y and plot the resultant spectrum. Compare the input signal spectrum with the demodulated signal spectrum. The spectra are similar except that demodulated signal has smaller high frequency components.

```
z = fmbDemod(y);
SAaudio([x z])
```



FM Broadcast a Streaming Audio Signal

Modulate and demodulate a streaming audio signal with the FM broadcast modulator and demodulator objects. Play the audio signal using a default audio device.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an audio file reader System object™ and read the file `guitartune.wav`.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',4410);
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator. Set the `PlaySound` property of the demodulator to `true` to enable audio playback.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3,'PlaySound',true);
```

Read the audio data in frames of length 4410, apply FM broadcast modulation, demodulate the FM signal and playback the audio input.

```
while ~isDone(audio)
    audioData = audio();
    modData = fmbMod(audioData);
    demodData = fmbDemod(modData);
end
```

FM Modulate and Demodulate an RBDS Waveform

Generate a basic RBDS waveform, FM modulate it with an audio signal, and then demodulate it.

Note: This example runs only in R2017a or later.

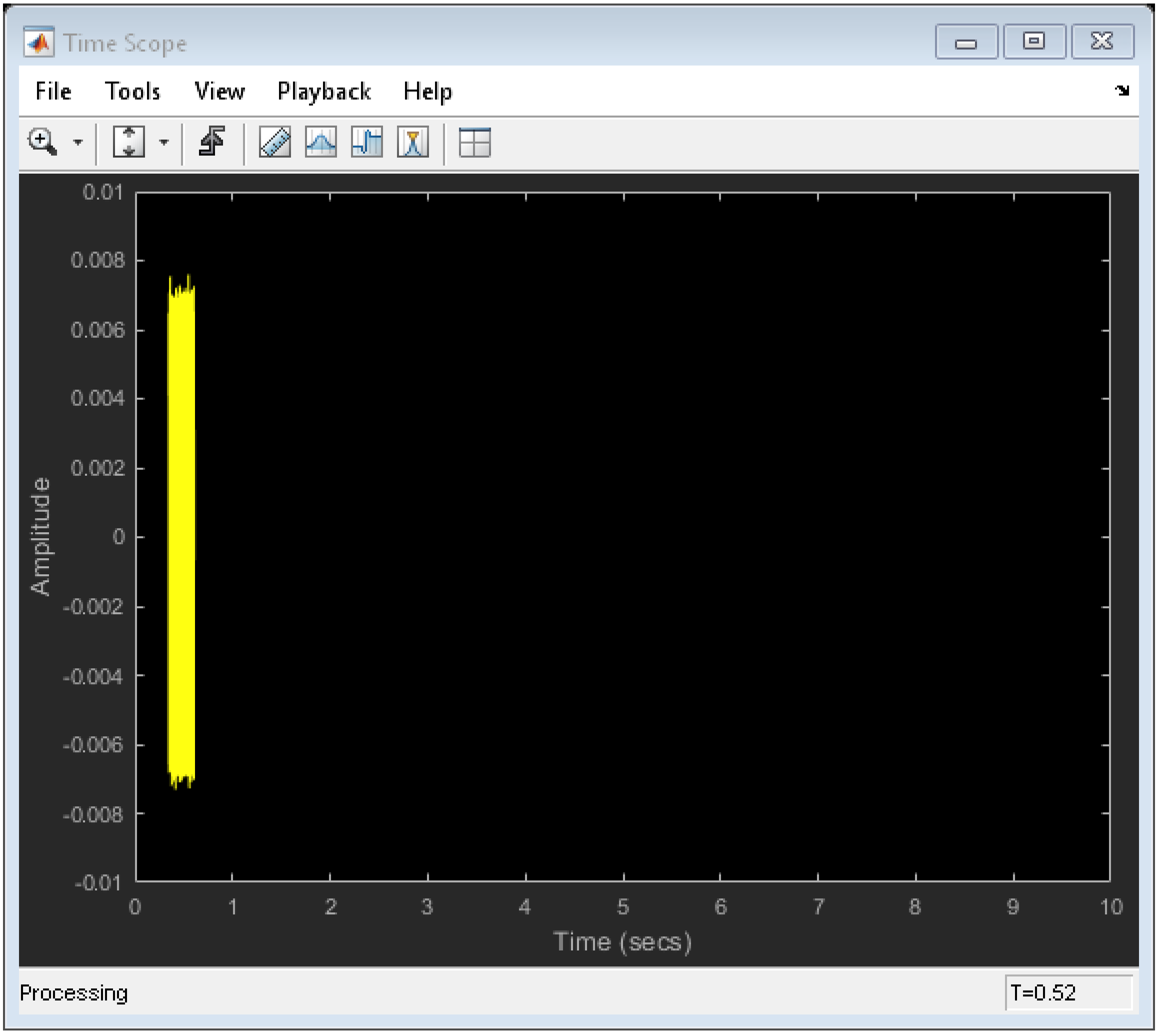
Create a RBDS waveform with 19 groups per frame and 10 samples per symbol. The sample rate of the RBDS waveform is given by 1187.5 x 10. Set the audio sample rate to 1187.5 x 40.

```
groupLen = 104;
sps = 10;
```

```
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRa
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sp

fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = dsp.TimeScope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the current audio input. Generate RBDS information at the same configured rate as audio. FM modulate the stereo audio with RBDS information. Add additive white Gaussian noise. FM demodulate the audio and RBDS waveforms. View the waveforms in a time scope.

```
for idx = 1:7
    input = afr();
    rbdsWave = rbds();
    yFM = fmMod([input input], rbdsWave);
    rcv = awgn(yFM, 40);
    [audioRcv, rbdsRcv] = fmDemod(rcv);
    scope(rbdsRcv);
end
```

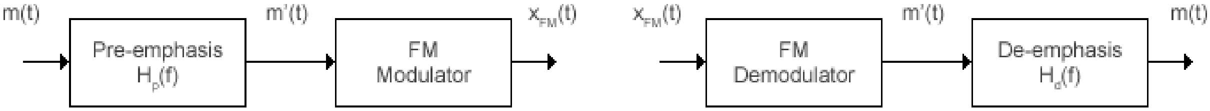


Algorithms

The FM Broadcast modulator includes the functionality of the baseband FM modulator, pre-emphasis filtering, and the ability to transmit stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMModulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f \tau_s \ ,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

$$H_d(f) = \frac{1}{1 + j2\pi f \tau_s} \ .$$

Irrespective of the audio sampling rate, the signal is converted to a 152 kHz output sampling rate. For an audio sample rate of 44.1 kHz, the pre-emphasis filter has the following response.

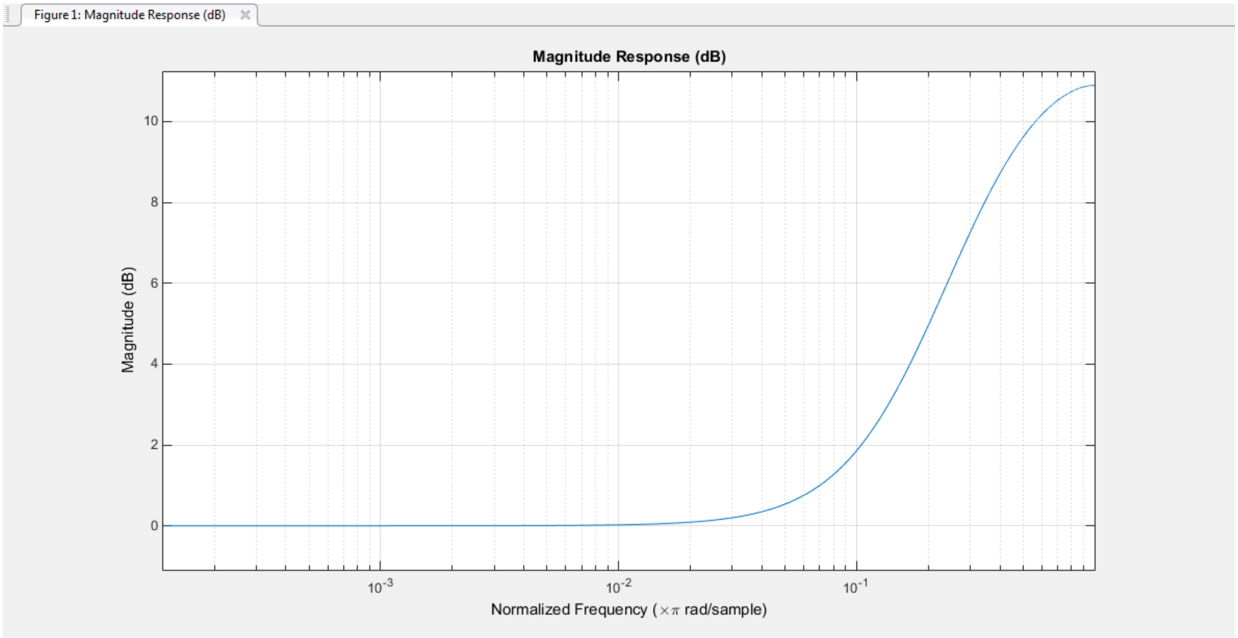
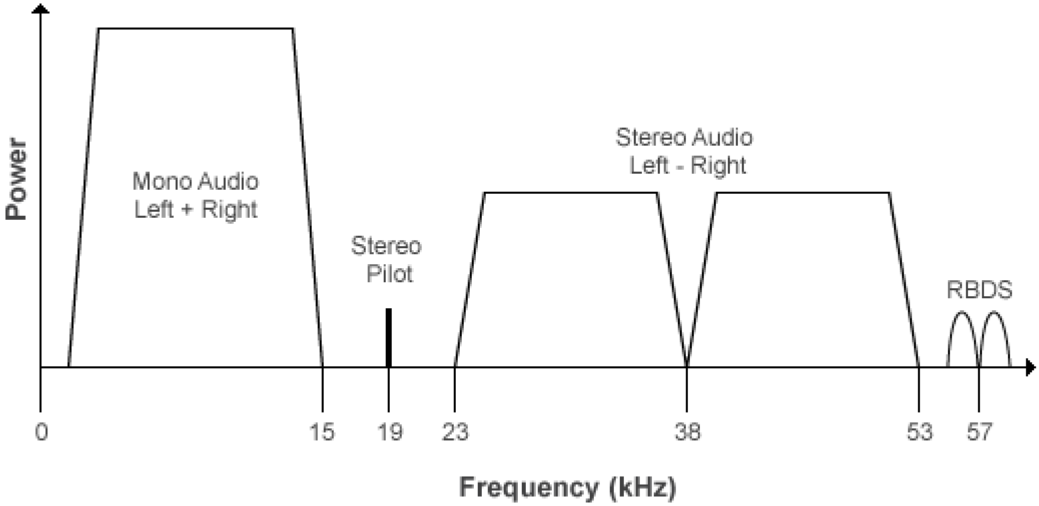


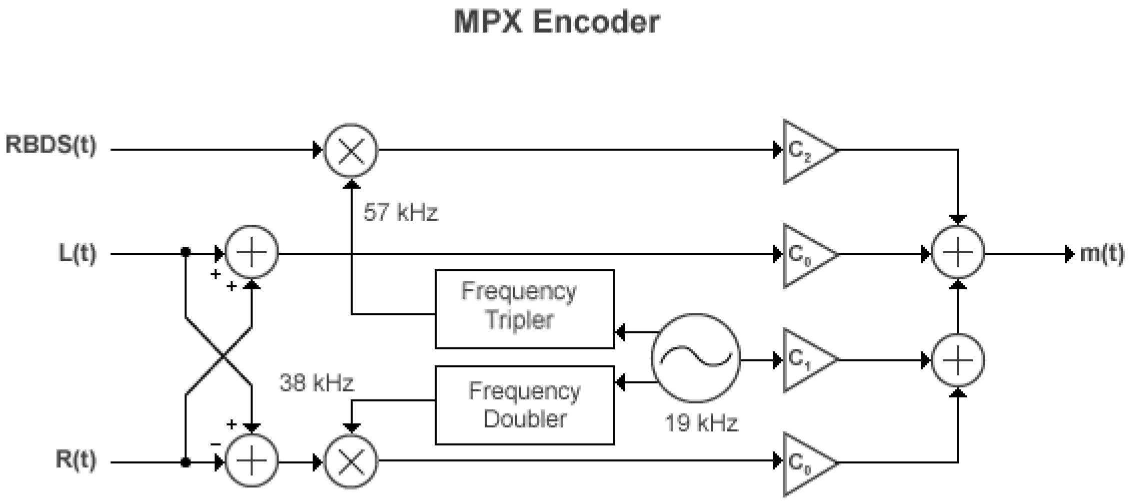
Figure 1: Magnitude Response (dB)
Magnitude Response (dB)
Magnitude (dB)
10
8
6
4
2
0
10^{-3}
10^{-2}
10^{-1}
Normalized Frequency ($\times\pi$ rad/sample)

Stereo and RDS/RBDS FM – Multiplex Signal

The FM broadcast modulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals. Here is the spectrum of the multiplex baseband signal.



Here is the block diagram of the FM broadcast modulator, which is used to generate the multiplex baseband signal. *L(t)* and *R(t)* denote the time-domain waveforms from the left and right channels. *RBDS(t)* denotes the time-domain waveform of the RDS/RBDS signal.



The multiplex message signal, *m(t)* is given by

$$m(t) = C_0\left[L(t) + R(t)\right] + C_1 \cos(2\pi \times 19kHz \times t) + C_0\left[L(t) - R(t)\right]\cos(2\pi \times 38kHz \times t) + C_2 RBDS(t)\cos(2$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

Limitations

- If `RBDS` is `true`, both the audio and RDS/RBDS inputs must satisfy the following equation:

$$\frac{audioLength}{audioSampleRate} = \frac{RBDSLength}{RBDSSampleRate}$$

- The input length of the audio signal must be an integer multiple of the `AudioDecimationFactor` property. The input length of the RDS/RBDS signal must be an integer multiple of the `RBDSDecimationFactor` property. For more information on these two properties, see the `info` method.

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects

`comm.FMBroadcastDemodulator` | `comm.FMDemodulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Blocks

FM Broadcast Demodulator Baseband | FM Broadcast Modulator Baseband

Introduced in R2015a

info

System object: comm.FMBroadcastModulator
Package: comm

Filter information about FM broadcast modulator

Syntax

`S = info(fmbMod)`

Description

`S = info(fmbMod)` returns a structure, `S`, containing this information for the `comm.FMBroadcastModulator` System object, `fmbMod`:

Field	Description
`AudioDecimationFactor`	Decimation factor of the audio modulator filter.
`AudioInterpolationFactor`	Interpolation factor of the audio modulator filter.
`RBDSDecimationFactor`	Decimation factor of the RDS/RBDS modulator filter.
`RBDSInterpolationFactor`	Interpolation factor of the RDS/RBDS modulator filter.

Note The modulator input sequence length for the audio input must be a multiple of `AudioDecimationFactor`.

The modulator input sequence length for the RDS/RBDS input must be a multiple of `RBDSDecimationFactor`.

Introduced in R2015a

reset

System object: comm.FMBroadcastModulator
Package: comm

Reset states of the FM broadcast modulator object

Syntax

```
reset(fmbMod)
```

Description

`reset(fmbMod)` resets the states of the `comm.FMBroadcastModulator` object, `fmbMod`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

Introduced in R2015a

step

System object: comm.FMBroadcastModulator
Package: comm

Apply FM broadcast modulation

Syntax

```
modSig = step(fmbMod,audioSig)
modSig = step(fmbMod,audioSig,rbdsSig)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`modSig = step(fmbMod,audioSig)` pre-emphasizes the audio signal, `audioSig`, and modulates it onto a baseband FM signal. The audio signal can be real or complex with a single-precision or a double-precision data type. If the `Stereo` property of `fmbMod` is set to `true`, stereo encoding is performed after pre-emphasis and the audio signal must have at least two channels. If `Stereo` is `false`, the audio signal must be a column vector. The length of the modulated signal, `modSig`, is `(SampleRate/AudioSampleRate) × length(audioSig)`.

`modSig = step(fmbMod,audioSig,rbdsSig)` also modulates a baseband RBDS signal at 57 kHz. You can pass `rbdsSig` as an input only if you set the `RBDS` property to true. The length of output vector `modSig` is `(SampleRate/AudioSampleRate) × length(audioSig)`.

Note `fmbMod` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2015a

comm.FMDemodulator System object

Package: comm

Demodulate using FM method

Description

The `FMDemodulator` System object demodulates an FM modulated signal.

To FM demodulate a signal:

1. Define and set up the `FMDemodulator` object. See "Construction" on page 3-629.
2. Call `step` to FM demodulate a signal according to the properties of `comm.FMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.FMDemodulator` creates a demodulator System object, `H`, that frequency demodulates an input signal.

`H = comm.FMDemodulator(mod)` creates an FM demodulator object whose properties are determined by the corresponding FM modulator object, `mod`.

`H = comm.FMDemodulator(Name,Value)` creates an FM demodulator object with each specified property `Name` set to the specified `Value`. `Name` must appear inside single quotes. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`FrequencyDeviation`

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM demodulator in Hz as a positive real scalar. The default value is `75e3`. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. This property is nontunable.

`SampleRate`

Sample rate of input signal (Hz)

Specify the sample rate in Hz as a positive real scalar. The default value is `240e3`. The output sample rate is equal to the input sample rate. This property is nontunable.

Methods

reset	Reset states of the FM demodulator object
step	Applies FM baseband demodulation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

FM Modulate and Demodulate a Sinusoidal Signal

Modulate and demodulate a sinusoidal signal. Plot the demodulated signal and compare it to the original signal.

Set the example parameters.

```
fs = 100;                % Sample rate (Hz)
ts = 1/fs;               % Sample period (s)
fd = 25;                 % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5s and frequency 4 Hz.

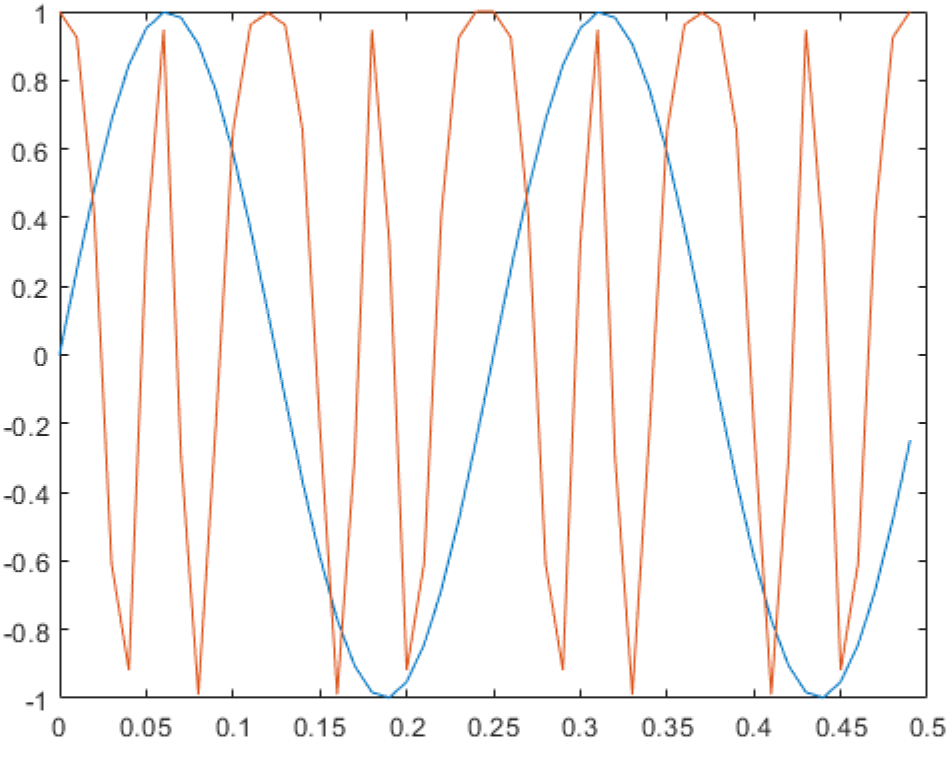
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create an FM modulator System object™.

```
MOD = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = step(MOD,x);
plot(t,[x real(y)])
```

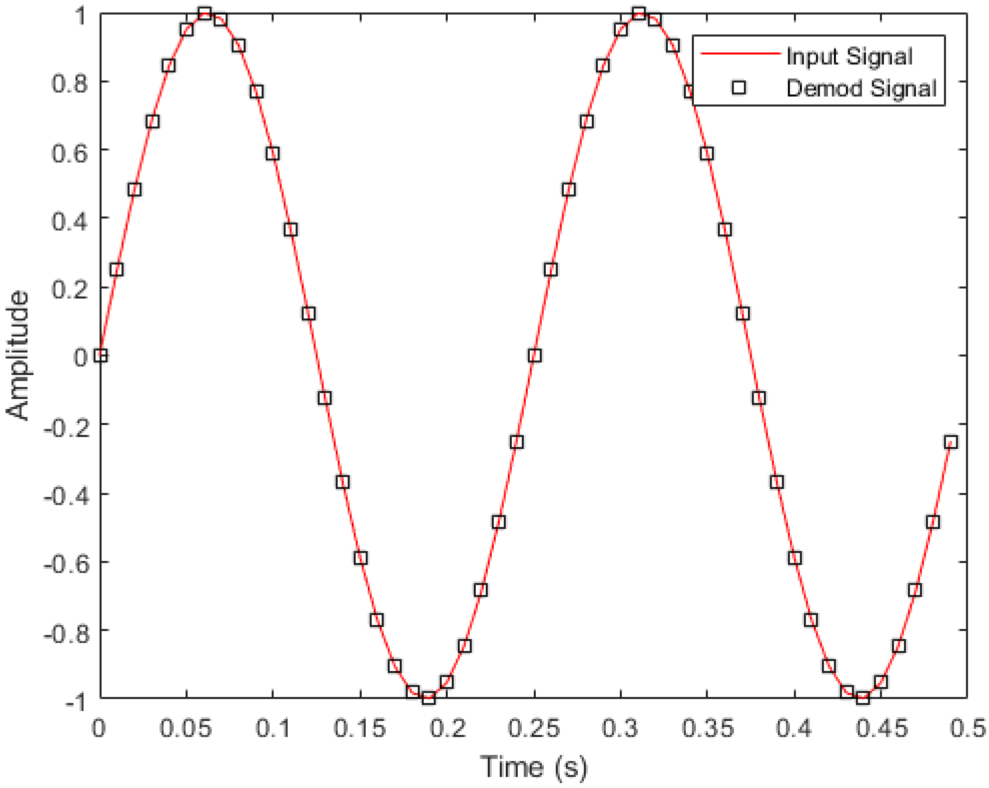


Demodulate the FM modulated signal.

```
DEMOD = comm.FMDemodulator('SampleRate',fs,'FrequencyDeviation',fd);
z = step(DEMOD,y);
```

Plot the input and demodulated signals. The demodulator output signal exactly aligns with the input signal.

```
plot(t,x,'r',t,z,'ks')
legend('Input Signal','Demod Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



Create an FM Demodulator from an FM Modulator

Create an FM demodulator System object™ from an FM modulator object. Modulate and demodulate audio data loaded from a file and compare its spectrum with that of the input data.

Set the example parameters.

```
fd = 50e3;                                % Frequency deviation (Hz)
fs = 300e3;                               % Sample rate (Hz)
```

Create an FM modulator System object.

```
MOD = comm.FMModulator('FrequencyDeviation',fd,'SampleRate',fs);
```

Create a companion demodulator object based on the modulator.

```
DEMOD = comm.FMDemodulator(MOD);
```

Verify that the properties are identical in the two System objects.

```
MOD
```

```
MOD =
  comm.FMModulator with properties:

            SampleRate: 300000
    FrequencyDeviation: 50000
```

```
DEMOD
```

```
DEMOD =
  comm.FMDemodulator with properties:

            SampleRate: 300000
    FrequencyDeviation: 50000
```

Load audio data into structure variable, S.

```
S = load('handel.mat');
data = S.y;
fsamp = S.Fs;
```

Create a spectrum analyzer System object.

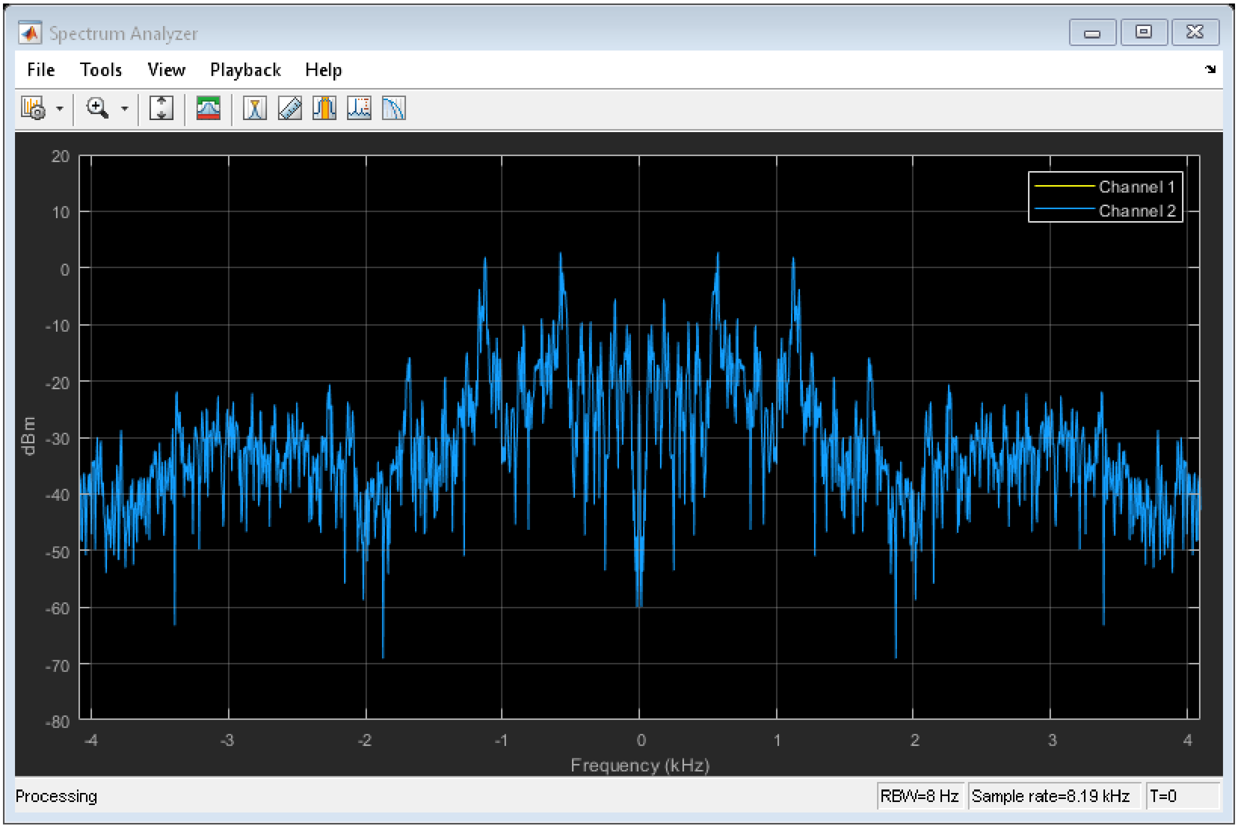
```
SA = dsp.SpectrumAnalyzer('SampleRate',fsamp,'ShowLegend',true);
```

FM modulate and demodulate the audio data.

```
modData = step(MOD,data);
demodData = step(DEMOD,modData);
```

Verify that the spectrum plot of the input data (Channel 1) is aligned with that of the demodulated data (Channel 2).

```
step(SA,[data demodData])
```



FM Modulate and Demodulate an Audio File

Playback an audio file after applying FM modulation and demodulation. The example takes advantage of the characteristics of System objects™ to process the data in streaming mode.

Load the audio file, `guitartune.wav`, using an audio file reader object.

```
AUDIO = dsp.AudioFileReader...
    ('guitartune.wav','SamplesPerFrame',4410);
```

Create an audio device writer object for audio playback.

```
AUDIOPLAYER = audioDeviceWriter;
```

Create modulator and demodulator objects having default properties.

```
MOD = comm.FMModulator;
DEMOD = comm.FMDemodulator;
```

Read audio data, FM modulate, FM demodulate, and playback the demodulated signal, `z`.

```
while ~isDone(AUDIO)
    x = step(AUDIO);                         % Read audio data
    y = step(MOD,x);                         % FM modulate
    z = step(DEMOD,y);                       % FM demodulate
    step(AUDIOPLAYER,z);                     % Playback the demodulated signal
end
```

Algorithms

You can represent a standard frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where A is the carrier amplitude, f_c is the carrier frequency, $x(\tau)$ is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming $|x(t)| \leq 1$.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$y_s(t) = Y(t)e^{-j2\pi f_c t} = \frac{A}{2}\left[e^{j\left(2\pi f_c t+2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t+2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t}$$

$$= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t-j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right].$$

Removing the component at -2f_c from $y_s(t)$ leaves the baseband signal representation, *y(t)*, which is expressed as

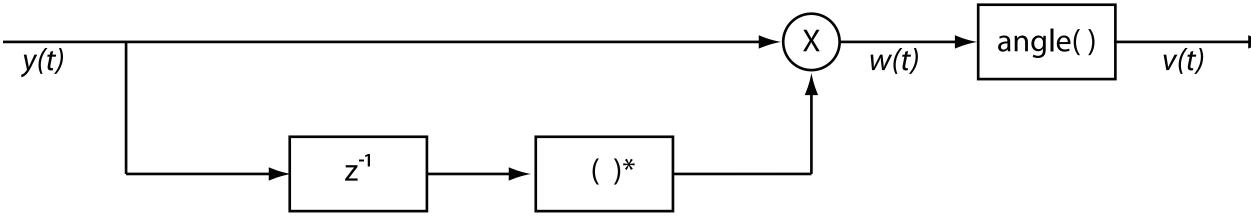
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)} ,$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, *ϕ(t)*.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself.

$$w(t) = \frac{A^2}{4}e^{j\phi(t)}e^{-j\phi(t-T)} = \frac{A^2}{4}e^{j[\phi(t)-\phi(t-T)]} ,$$

where *T* is the sample period. In discrete terms, w_n=*w(nT)*, consequently

$$w_n = \frac{A^2}{4} e^{j[\phi_n - \phi_{n-1}]} ,$$

$$v_n = \phi_n - \phi_{n-1} .$$

The signal v_n is the approximate derivative of ϕ_n such that $v_n \approx x_n$.

Selected Bibliography

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator` | `comm.FMModulator`

Introduced in R2015a

reset

System object: comm.FMDemodulator
Package: comm

Reset states of the FM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FMDemodulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: comm.FMDemodulator
Package: comm

Applies FM baseband demodulation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` frequency demodulates an input signal, `X`, and returns an output signal, `Y`. The input `X` is real or complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` is real and has the same data type and dimensions as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.FMModulator System object

Package: comm

Modulate using FM method

Description

The `FMModulator` System object applies FM modulation to an input signal.

To FM modulate a signal:

1. Define and set up the `FMModulator` object. See "Construction" on page 3-640.
2. Call `step` to apply FM modulation to a signal according to the properties of `comm.FMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.FMModulator` creates a modulator System object, `H`, that frequency modulates an input signal.

`H = comm.FMModulator(demod)` creates an FM modulator object whose properties are determined by the corresponding FM demodulator object, `demod`.

`H = comm.FMModulator(Name,Value)` creates an FM modulator object with each specified property `Name` set to the specified `Value`. `Name` must appear inside single quotes. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

FrequencyDeviation

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM modulator in Hz as a positive real scalar. The default value is 75e3. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. This property is nontunable.

SampleRate

Sample rate of the input signal (Hz)

Specify the sample rate in Hz as a positive real scalar. The default value is 240e3. The output sample rate is equal to the input sample rate. This property is nontunable.

Methods

reset	Reset states of the FM modulator object
step	Applies FM baseband modulation

Common to All System Objects	
release	Allow System object property value changes

Examples

FM Modulate a Sinusoidal Signal

Apply baseband modulation to a sine wave input signal and plot its response.

Set the example parameters.

```
fs = 1e3;                % Sample rate (Hz)
ts = 1/fs;               % Sample period (s)
fd = 50;                 % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5s and frequency 4 Hz.

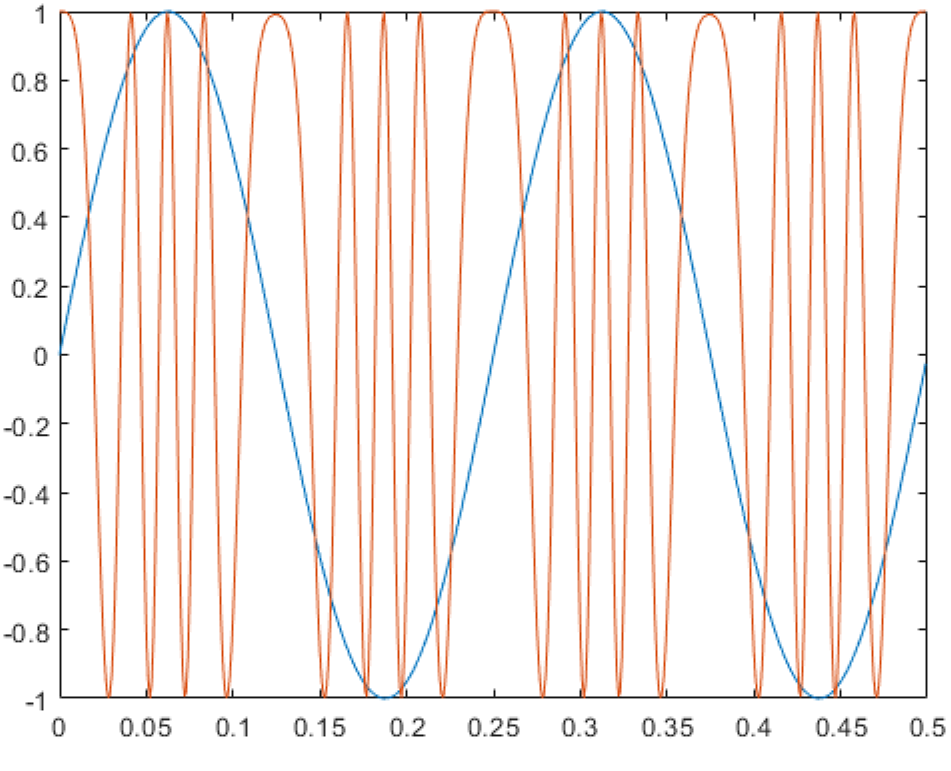
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create an FM modulator System object™.

```
MOD = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = step(MOD,x);
plot(t,[x real(y)])
```



Plot Spectrum of FM Modulated Baseband Signal

Apply FM baseband modulation to a white Gaussian noise source and plot its spectrum.

Set the example parameters.

```
fs = 1e3;              % Sample rate (Hz)
ts = 1/fs;             % Sample period (s)
fd = 10;               % Frequency deviation (Hz)
```

Create a white Gaussian noise source having a duration of 5s.

```
t = (0:ts:5-ts)';
x = wgn(length(t),1,0);
```

Create an FM modulator System object™ and modulate the input signal.

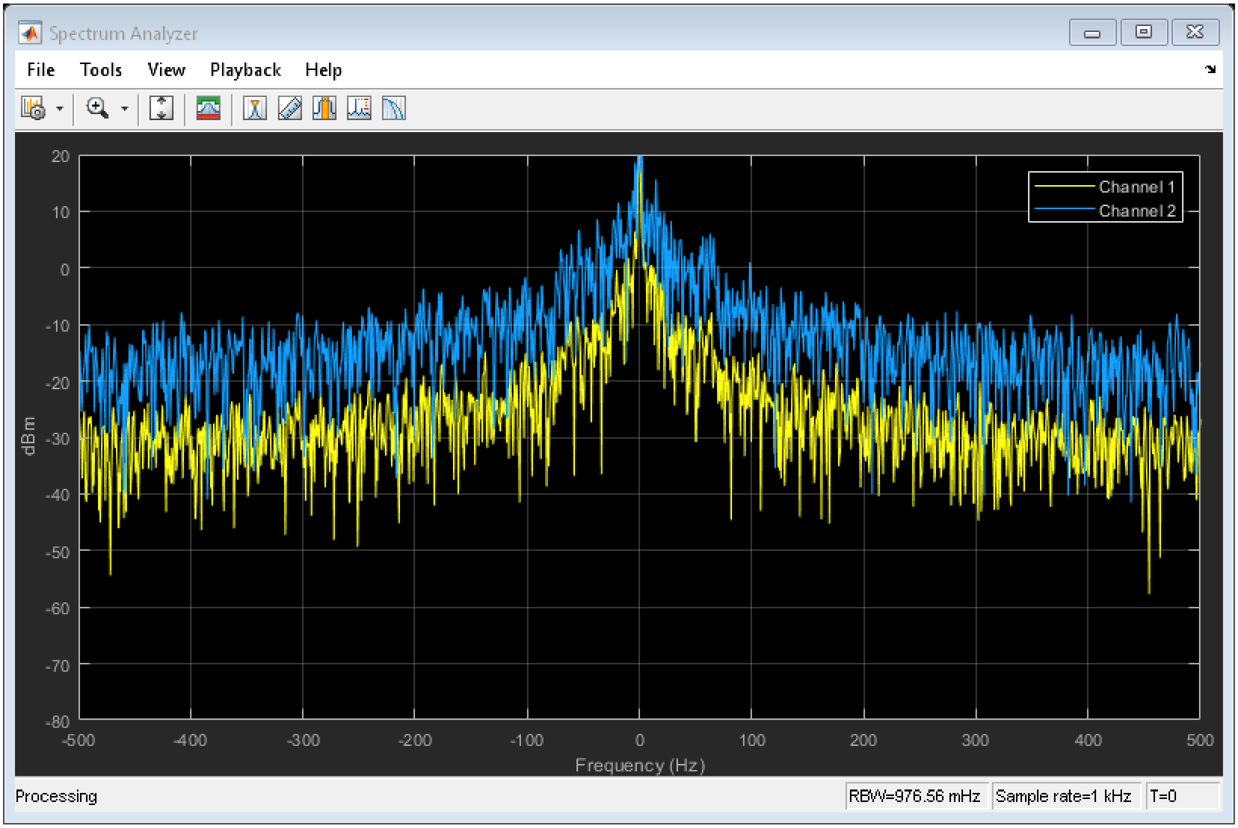
```
MOD1 = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
y = step(MOD1,x);
```

Create another modulator object, `MOD2`, whose frequency deviation is five times larger and apply FM modulation.

```
MOD2 = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',5*fd);
z = step(MOD2,x);
```

Plot the spectra of the two modulated signals. The larger frequency deviation associated with channel 2 results in a noise level that is 10 dB higher.

```
SA = dsp.SpectrumAnalyzer('SampleRate',fs,'ShowLegend',true);
step(SA,[y z])
```



Algorithms

You can represent a standard frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where A is the carrier amplitude, f_c is the carrier frequency, $x(\tau)$ is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming $|x(t)| \leq 1$.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$y_s(t) = Y(t)e^{-j2\pi f_c t} = \frac{A}{2}\left[e^{j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t}$$

$$= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t - j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right].$$

Removing the component at -2f_c from $y_s(t)$ leaves the baseband signal representation, *y(t)*, which is expressed as

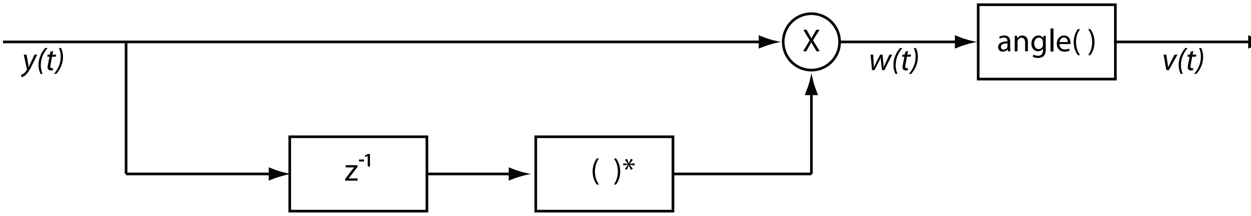
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)} ,$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, $\phi(t)$.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself.

$$w(t) = \frac{A^2}{4}e^{j\phi(t)}e^{-j\phi(t-T)} = \frac{A^2}{4}e^{j[\phi(t)-\phi(t-T)]} ,$$

where *T* is the sample period. In discrete terms, w_n=*w(nT)*, consequently

$$w_n = \frac{A^2}{4} e^{j[\phi_n - \phi_{n-1}]} \, ,$$
$$v_n = \phi_n - \phi_{n-1} \, .$$

The signal v_n is the approximate derivative of ϕ_n such that $v_n \approx x_n$.

Selected Bibliography

[1] Chakrabarti, I. H., and Hatai, I. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator` | `comm.FMDemodulator`

Introduced in R2015a

reset

System object: comm.FMModulator
Package: comm

Reset states of the FM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FMModulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: comm.FMModulator
Package: comm

Applies FM baseband modulation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` frequency modulates an input signal, `X`, and returns a modulated signal, `Y`. The input `X` is real or complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` has the same data type and dimensions as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.FSKDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using frequency shift keying:

1. Define and set up your FSK demodulator object. See "Construction" on page 3-649.
2. Call `step` to demodulate a signal according to the properties of `FSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 . *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` on page 3-0 –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` on page 3-0) 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` on page 3-0 ×`SymbolRate`, which is greater than `ModulationOrder` on page 3-0 × `FrequencySeparation` on page 3-0 . 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` on page 3-0 property to false and the `ModulationOrder` on page 3-0 property to two. When you set the `BitOutput` property to true, the output data requires a type of `logical` | `double`.

Methods

reset	Reset states of M-FSK demodulator object
step	Demodulate using M-ary FSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

FSK Modulation and Demodulation in AWGN

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz.

Set the modulation order and frequency separation parameters.

```
M = 8;
freqSep = 100;
```

Create FSK modulator and demodulator System objects™ with modulation order 8 and 100 Hz frequency separation.

```
fskMod = comm.FSKModulator(M,freqSep);
fskDemod = comm.FSKDemodulator(M,freqSep);
```

Create an additive white Gaussian noise channel, where the noise is specified as a signal-to-noise ratio.

```
ch = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',-2);
```

Create an error rate calculator object.

```
err = comm.ErrorRate;
```

Transmit one hundred 50-symbol frames using 8-FSK in an AWGN channel.

```
for counter = 1:100
    data = randi([0 M-1],50,1);
    modSignal = step(fskMod,data);
    noisySignal = step(ch,modSignal);
    receivedData = step(fskDemod,noisySignal);
    errorStats = step(err,data,receivedData);
end
```

Display the error statistics.

```
es = 'Error rate = %4.2e\nNumber of errors = %d\nNumber of symbols = %d\n';
fprintf(es,errorStats)
```

```
Error rate = 1.40e-02
Number of errors = 70
Number of symbols = 5000
```

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` on page 3-0 property.
- The `SymbolRate` on page 3-0 property replaces the block sample rate capability.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPFSKModulator` | `comm.FSKModulator`

Introduced in R2012a

reset

System object: comm.FSKDemodulator
Package: comm

Reset states of M-FSK demodulator object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `FSKDemodulator` object, `H`.

step

System object: comm.FSKDemodulator
Package: comm

Demodulate using M-ary FSK method

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.FSKModulator System object

Package: comm

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.

To modulate a signal using frequency shift keying:

1. Define and set up your FSK modulator object. See “Construction” on page 3-656.
2. Call `step` to modulate a signal according to the properties of `comm.FSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 –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` on page 3-0) 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` on page 3-0 ×`SymbolRate` on page 3-0 , which is greater than `ModulationOrder` on page 3-0 multiplied by `FrequencySeparation` on page 3-0 .

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` on page 3-0 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` on page 3-0 ×`SymbolRate`, which is greater than `ModulationOrder` on page 3-0 × `FrequencySeparation` on page 3-0 . 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

reset	Reset states of M-FSK modulator object
step	Modulate using M-ary FSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

FSK Modulation and Demodulation in AWGN

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz.

Set the modulation order and frequency separation parameters.

```
M = 8;
freqSep = 100;
```

Create FSK modulator and demodulator System objects™ with modulation order 8 and 100 Hz frequency separation.

```
fskMod = comm.FSKModulator(M,freqSep);
fskDemod = comm.FSKDemodulator(M,freqSep);
```

Create an additive white Gaussian noise channel, where the noise is specified as a signal-to-noise ratio.

```
ch = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',-2);
```

Create an error rate calculator object.

```
err = comm.ErrorRate;
```

Transmit one hundred 50-symbol frames using 8-FSK in an AWGN channel.

```
for counter = 1:100
    data = randi([0 M-1],50,1);
    modSignal = step(fskMod,data);
    noisySignal = step(ch,modSignal);
    receivedData = step(fskDemod,noisySignal);
    errorStats = step(err,data,receivedData);
end
```

Display the error statistics.

```
es = 'Error rate = %4.2e\nNumber of errors = %d\nNumber of symbols = %d\n';
fprintf(es,errorStats)
```

```
Error rate = 1.40e-02
Number of errors = 70
Number of symbols = 5000
```

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` on page 3-0 property.
- The `SymbolRate` on page 3-0 property takes the place of the block sample rate capability.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.FSKDemodulator`

Introduced in R2012a

reset

System object: comm.FSKModulator
Package: comm

Reset states of M-FSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FSKModulator` object, `H`.

step

System object: comm.FSKModulator
Package: comm

Modulate using M-ary FSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Recover symbol timing phase using Gardner's method

Note `comm.GardnerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

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.

To recover the symbol timing phase of the input signal:

1. Define and set up your Gardner timing synchronizer object. See “Construction” on page 3-665.
2. Call `step` to recover symbol timing phase according to the properties of `comm.GardnerTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 , 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` on page 3-0 property to false.

Methods

reset	Reset states of Gardner timing phase synchronizer object
step	Recover symbol timing phase using Gardner's method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover timing phase using the Gardner method.

```
% Initialize data
    L = 16; M = 8; numSymb = 100; snrdB = 30;
    R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;

% Create System objects
    hMod = comm.PSKModulator(M);
    hTxFilter = comm.RaisedCosineTransmitFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'OutputSamplesPerSymbol', L);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
        'NoiseMethod', 'Signal to noise ratio (SNR)', ...
        'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = comm.RaisedCosineReceiveFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'InputSamplesPerSymbol', L, ...
        'DecimationFactor', 1);
    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` on page 3-0 and `ResetCondition` on page 3-0 properties.

See Also

`comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: comm.GardnerTimingSynchronizer
Package: comm

Reset states of Gardner timing phase synchronizer object

Syntax

```
reset(H)
```

Note `comm.GardnerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

`reset(H)` resets the states of the `GardnerTimingSynchronizer` object, `H`.

step

System object: comm.GardnerTimingSynchronizer
Package: comm

Recover symbol timing phase using Gardner's method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Note `comm.GardnerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GeneralQAMDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using quadrature amplitude modulation:

1. Define and set up your QAM demodulator object. See “Construction” on page 3-672.
2. Call `step` to demodulate a signal according to the properties of `comm.GeneralQAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 property to `Hard decision` or `Approximate log-likelihood ratio`. In this case, when you set the `OutputDataType` on page 3-0 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 “Fixed-Point Support for MATLAB System Objects in DSP System Toolbox” (DSP System Toolbox).

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` on page 3-0 to true and `DecisionMethod` on page 3-0 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` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 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` on page 3-0 property to false. This property does not apply when you set the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 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` on page 3-0 property to false, the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 property to false and the `ProductInputDataType` on page 3-0 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` on page 3-0 property to false, the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 property to false and the `ProductOutputDataType` on page 3-0 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` on page 3-0 property to false, the `BitOutput` on page 3-0 property to `true` and the `DecisionMethod` on page 3-0 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` on page 3-0 property to false and the `Accumulator2DataType` on page 3-0 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` on page 3-0 property to false, the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 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` on page 3-0 property to false and the `Accumulator3DataType` on page 3-0 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 on page 3-0 property to false, the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 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 on page 3-0 property to false and the NoiseScalingInputDataType on page 3-0 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 on page 3-0 property to true, the DecisionMethod on page 3-0 property to Approximate log-likelihood ratio, and the VarianceSource on page 3-0 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 on page 3-0 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 on page 3-0 property to false and the OutputDataType on page 3-0 property to Custom.

Methods

step	Demodulate using arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

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', 0.89);
   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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMModulator` | `comm.RectangularQAMDemodulator`

Introduced in R2012a

step

System object: comm.GeneralQAMDemodulator
Package: comm

Demodulate using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GeneralQAMModulator System object

Package: comm

Modulate using arbitrary QAM constellation

Description

The `GeneralQAMModulator` object modulates using quadrature amplitude modulation. The output is a baseband representation of the modulated signal.

To modulate a signal using quadrature amplitude modulation:

1. Define and set up your QAM modulator object. See “Construction” on page 3-684.
2. Call `step` to modulate a signal according to the properties of `comm.GeneralQAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 property to Custom.

Methods

step	Modulate using arbitrary QAM constellation

Common to All System Objects	
release	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMDemodulator` | `comm.RectangularQAMModulator`

Introduced in R2012a

step

System object: comm.GeneralQAMModulator
Package: comm

Modulate using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GeneralQAMTCMDemodulator System object

Package: comm

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.

To demodulate a signal that was modulated using a trellis-coded, general quadrature amplitude modulator:

1. Define and set up your general QAM TCM modulator object. See “Construction” on page 3-688.
2. Call `step` to demodulate a signal according to the properties of `comm.GeneralQAMTCMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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 on page 3-0 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

reset	Reset states of the general QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate and Demodulate Data Using QAM TCM

Modulate and demodulate noisy data using QAM TCM modulation with an arbitrary 4-point constellation. Estimate the resultant BER.

Define a trellis structure with two input symbols and four output symbols using a [171 133] generator polynomial. Define an arbitrary four-point constellation.

```
qamTrellis = poly2trellis(7,[171 133]);
refConst = exp(pi*1i*[1 2 3 6]/4);
```

Create a QAM TCM modulator and demodulator System object™ pair using `qamTrellis` and `refConst`.

```
hMod = comm.GeneralQAMTCMModulator(qamTrellis,'Constellation', refConst);
hDemod = comm.GeneralQAMTCMDemodulator(qamTrellis,'Constellation',refConst);
```

Create an AWGN channel object in which the noise is set by using a signal-to-noise ratio.

```
hAWGN = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)', ...
    'SNR',4);
```

Create an error rate calculator with delay (in bits) equal to the product of `TracebackDepth` and the number of bits per symbol

```
hError = comm.ErrorRate(...
    'ReceiveDelay', hDemod.TracebackDepth*log2(qamTrellis.numInputSymbols));
```

Generate random binary data and apply QAM TCM modulation. Pass the signal through an AWGN channel and demodulate. Collect the error statistics.

```
for counter = 1:10
    % Generate binary data
    data = randi([0 1],500,1);
    % Modulate
    modSignal = step(hMod,data);
    % Pass through an AWGN channel
    noisySignal = step(hAWGN,modSignal);
    % Demodulate
    receivedData = step(hDemod,noisySignal);
    % Calculate the error statistics
    errorStats = step(hError,data,receivedData);
end
```

Display the BER and the number of bit errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 1.16e-02
Number of errors = 58
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMDemodulator` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: comm.GeneralQAMTCMDemodulator
Package: comm

Reset states of the general QAM TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GeneralQAMTCMDemodulator` object, `H`.

step

System object: comm.GeneralQAMTCMDemodulator
Package: comm

Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GeneralQAMTCMModulator System object

Package: comm

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.

To modulate a signal using a trellis-coded, general quadrature amplitude modulator:

1. Define and set up your general QAM TCM modulator object. See “Construction” on page 3-697.
2. Call `step` to modulate a signal according to the properties of `comm.GeneralQAMTCMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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

reset	Reset states of the general QAM TCM modulator object
step	Convolutionally encode binary data and map using arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data using QAM TCM with an Arbitrary Constellation

Modulate data using QAM TCM modulation with an arbitrary 4-point constellation. Display a scatter plot of the modulated data.

Create binary data.

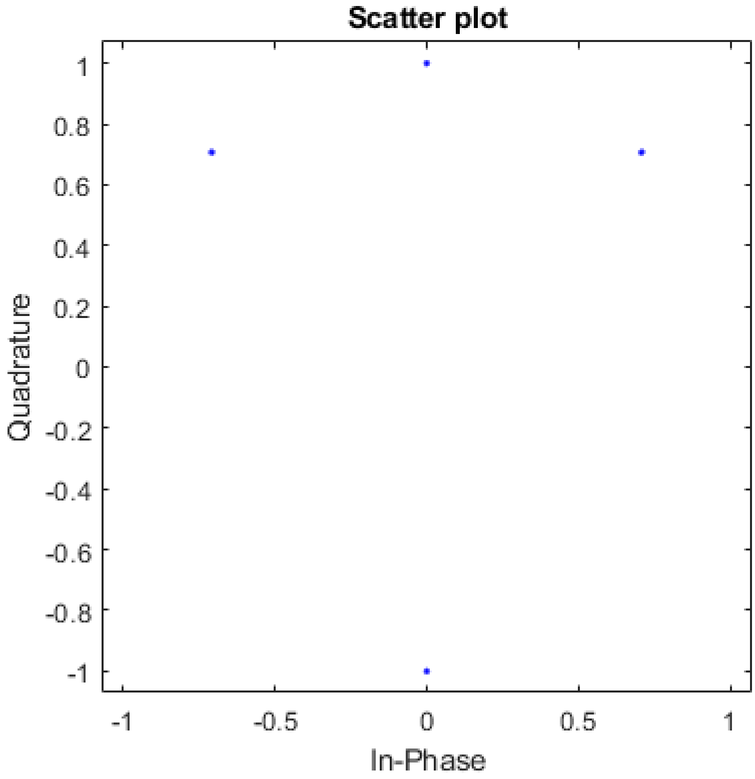
```
data = randi([0 1],1000,1);
```

Use the trellis structure with generating polynomial [171 133] and 4-point arbitrary constellation { $e^{j\pi/4}$, $e^{j\pi/2}$, $e^{j3\pi/4}$, $e^{j3\pi/2}$ } to perform QAM TCM modulation.

```
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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMModulator` | `comm.GeneralQAMTCMDemodulator` | `comm.PSKTCMModulator`

Introduced in R2012a

reset

System object: comm.GeneralQAMTCMModulator
Package: comm

Reset states of the general QAM TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GeneralQAMTCMModulator` object, `H`.

step

System object: comm.GeneralQAMTCMModulator
Package: comm

Convolutionally encode binary data and map using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To demodulate a signal that was modulated using Gaussian minimum shift keying:

1. Define and set up your GMSK demodulator object. See "Construction" on page 3-649.
2. Call `step` to demodulate a signal according to the properties of `GMSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property to `false`, the `step` method outputs a column vector of length equal to *N*/`SamplesPerSymbol` on page 3-0 . *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` on page 3-0 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` on page 3-0 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

reset	Reset states of the GMSK demodulator object
step	Demodulate using GMSK method and the Viterbi algorithm

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Demodulate a GMSK signal with bit inputs and phase offset

```
% Create a GMSK modulator, an AWGN channel, and a GMSK demodulator. Use a phase offset
    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 Viterbi algorith
    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))
```

```
Error rate = 0.000133
Number of errors = 4
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

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. For GMSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.GMSKModulator`

Introduced in R2012a

reset

System object: comm.GMSKDemodulator
Package: comm

Reset states of the GMSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GMSKDemodulator` object, `H`.

step

System object: comm.GMSKDemodulator
Package: comm

Demodulate using GMSK method and the Viterbi algorithm

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GMSKModulator System object

Package: comm

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.

To modulate a signal using Gaussian minimum shift keying:

1. Define and set up your GMSK modulator object. See "Construction" on page 3-714.
2. Call `step` to modulate a signal according to the properties of `comm.GMSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

reset	Reset states of the GMSK modulator object
step	Modulate using GMSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate a GMSK signal with bit inputs and phase offset

```
% Create a GMSK modulator, an AWGN channel, and a GMSK demodulator. Use a phase offset
    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 Viterbi algorit
    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))
```

```
Error rate = 0.000133
Number of errors = 4
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

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. For GMSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.GMSKDemodulator`

Introduced in R2012a

reset

System object: comm.GMSKModulator
Package: comm

Reset states of the GMSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GMSKModulator` object, `H`.

step

System object: comm.GMSKModulator
Package: comm

Modulate using GMSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To recover the symbol timing phase of the input signal:

1. Define and set up your GMSK timing synchronizer object. See “Construction” on page 3-722.
2. Call `step` to recover the symbol timing phase of the input signal according to the properties of `comm.GMSKTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 , 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` on page 3-0 property to `false`.

Methods

reset	Reset states of GMSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover Timing Phase of GMSK Signal

Create GMSK modulator, variable fractional delay, and GMSK timing synchronizer System objects.

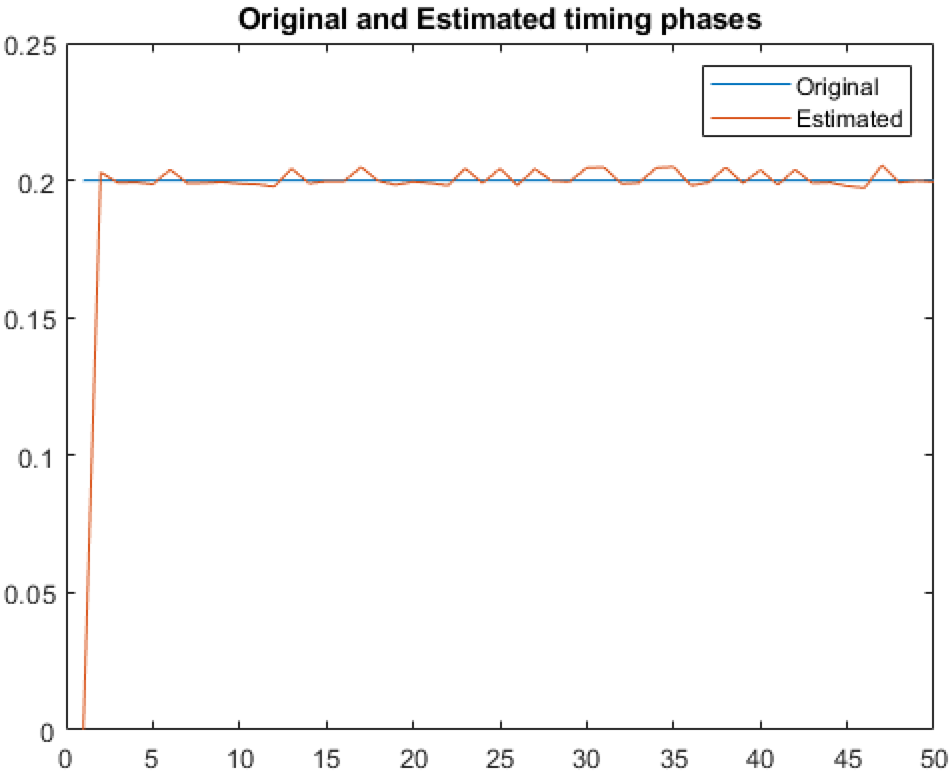
```
gmskMod = comm.GMSKModulator('BitInput', true, ...
    'SamplesPerSymbol', 14);
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
gmskTimingSync = comm.GMSKTimingSynchronizer('SamplesPerSymbol', 14, ...
    'ErrorUpdateGain', 0.05);
```

Main processing loop:

```
phEst = zeros(50,1);
for i = 1:50
    data = randi([0 1],100,1); % Generate data
    modData = gmskMod(data);   % Modulate data

    % Apply timing offset error
    impairedData = varDelay(modData,timingOffset*14);
    % Perform timing phase recovery
    [~,phase] = gmskTimingSync(impairedData);
    phEst(i) = phase(1)/14;
end
```

Plot the results.

```
plot(1:50,[0.2*ones(50,1) phEst])
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` on page 3-0 and `ResetCondition` on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: comm.GMSKTimingSynchronizer
Package: comm

Reset states of GMSK timing phase synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states for the `GMSKTimingSynchronizer` object `H`.

step

System object: comm.GMSKTimingSynchronizer
Package: comm

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.GoldSequence System object

Package: comm

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.

To generate a Gold sequence:

1. Define and set up your Gold sequence object. See “Construction” on page 3-730.
2. Call `step` to generate the Gold sequence according to the properties of `comm.GoldSequence`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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 `'z^6 + z + 1'`. You can specify the polynomial as a character vector. You can also 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. Lastly, 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` on page 3-0 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` on page 3-0 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 `'z^6 + z^5 + z^2 + z + 1'`. You can specify the polynomial as a character vector. You can also 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. Lastly, 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 and `SecondPolynomial` on page 3-0 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 $G(u,v) = \{u, v, (u \texttt{xor} T^{v}), (u \texttt{ xor } T^{2v}), \ldots, (u \texttt{ xor } T^{((N-1)v)})\}$.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` on page 3-0 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` on page 3-0 and `SecondPolynomial` on page 3-0 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` on page 3-0 and `SecondPolynomial` on page 3-0 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` on page 3-0 and `SecondInitialConditions` on page 3-0 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

reset	Reset states of Gold sequence generator object
step	Generate a Gold sequence

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate Gold Sequence Samples

Generate 10 samples of a Gold sequence having period $2^5 - 1$.

```
goldseq = comm.GoldSequence('FirstPolynomial','x^5+x^2+1',...
    'SecondPolynomial','x^5+x^4+x^3+x^2+1',...
    'FirstInitialConditions',[0 0 0 0 1],...
    'SecondInitialConditions',[0 0 0 0 1],...
    'Index',4,'SamplesPerFrame',10);
x = goldseq()
```

```
x = 10×1

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

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.KasamiSequence` | `comm.PNSequence`

reset

System object: comm.GoldSequence
Package: comm

Reset states of Gold sequence generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GoldSequence` object, `H`.

step

System object: comm.GoldSequence
Package: comm

Generate a Gold sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.AWGNChannel System object

Package: comm

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” (Parallel Computing Toolbox).

To add white Gaussian noise to an input signal:

1. Define and set up your additive white Gaussian noise channel object. See “Construction” on page 3-741.
2. Call `step` to add white Gaussian noise to the input signal according to the properties of `comm.gpu.AWGNChannel`. The behavior of `step` is specific to each object in the toolbox.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

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

step	Add white Gaussian noise to input signal

Common to All System Objects	
`release`	Allow System object property value changes

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

GPU AWGN Channel

Specify the modulation order and generate PSK-modulated random data.

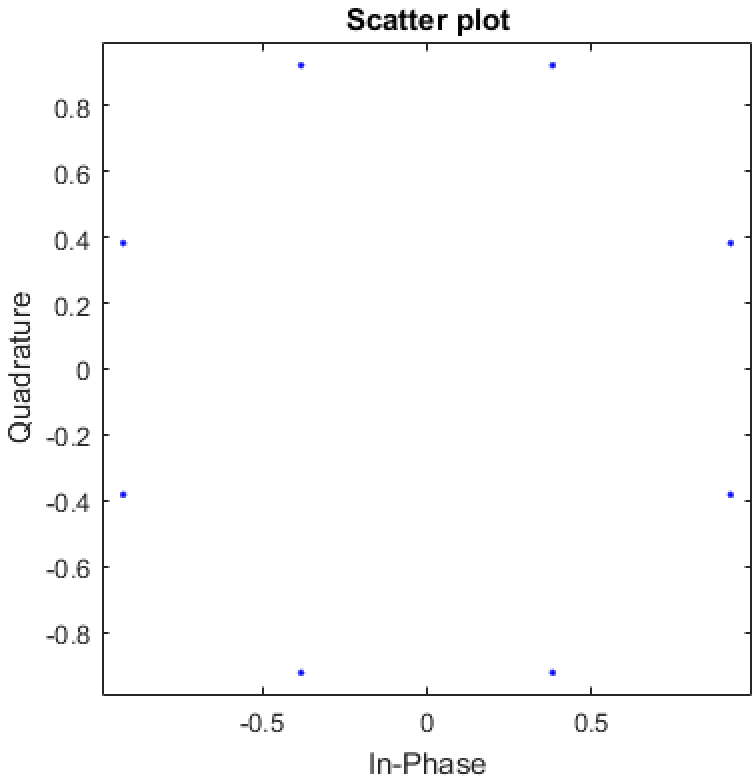
```
M = 8;
modData = pskmod(randi([0 M-1],1000,1),M,pi/M);
```

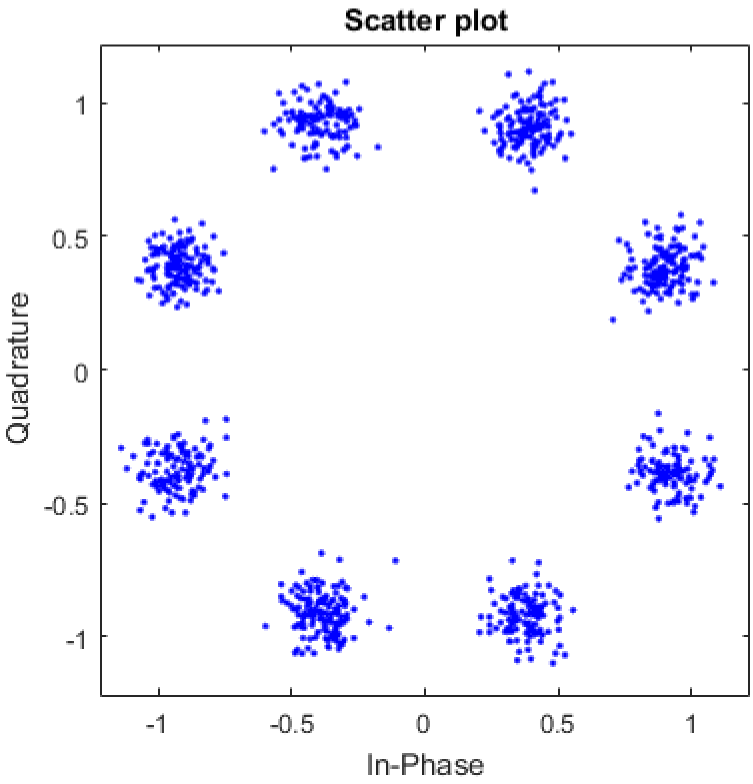
Create an AWGN channel object that uses a GPU. Pass the modulated data through the channel.

```
gpuChannel = comm.gpu.AWGNChannel('EbNo',15,'BitsPerSymbol', ...
                log2(M));
channelOutput = gpuChannel(modData);
```

Visualize the noiseless and noisy data in scatter plots.

```
scatterplot(modData)
scatterplot(channelOutput)
```





See Also

`comm.AWGNChannel`

Introduced in R2012a

step

System object: comm.gpu.AWGNChannel
Package: comm

Add white Gaussian noise to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To deinterleave the input vector:

1. Define and set up your block deinterleaver object. See “Construction” on page 3-750.
2. Call `step` to rearrange the elements of the input vector according to the properties of `comm.gpu.BlockDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Deinterleave input sequence

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

This object uses the same algorithm as the `comm.BlockDeinterleaver` System object. See Algorithms on the `comm.BlockDeinterleaver` help page for details.

Examples

Block Interleaving and Deinterleaving with GPU

Create interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.gpu.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data intData deIntData]
```

```
ans =

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver(permVec);
deinterleaver = comm.gpu.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

See Also

`comm.BlockDeinterleaver` | `comm.gpu.BlockInterleaver`

Introduced in R2012a

step

System object: comm.gpu.BlockDeinterleaver
Package: comm

Deinterleave input sequence

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.BlockInterleaver System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To interleave the input signal:

1. Define and set up your block interleaver object. See “Construction” on page 3-755.
2. Call `step` to reorder the input symbols according to the properties of `comm.gpu.BlockInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Permute input symbols using a permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

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

Block Interleaving and Deinterleaving with GPU

Create interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.gpu.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data intData deIntData]
```

```
ans =

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =
     1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver(permVec);
deinterleaver = comm.gpu.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

See Also

`comm.BlockInterleaver` | `comm.gpu.BlockDeinterleaver`

Introduced in R2012a

step

System object: comm.gpu.BlockInterleaver
Package: comm

Permute input symbols using a permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm.gpu

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" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To convolutionally encode a binary signal:

1. Define and set up your convolutional encoder object. See "Construction" on page 3-760.
2. Call `step` to encode a sequence of binary input vectors to produce a sequence of binary output vectors according to the properties of

comm.gpu.ConvolutionalEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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 on page 3-0 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` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 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` on page 3-0 property to `Continuous` or `Truncated` and the `PuncturePatternSource` on page 3-0 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` on page 3-0 to `Terminated` or `Truncated`.

Methods

reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Common to All System Objects	
`release`	Allow System object property value changes

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.ConvolutionalEncoder` | `comm.gpu.ConvolutionalDeinterleaver` | `comm.gpu.ConvolutionalInterleaver` | `comm.gpu.ViterbiDecoder`

reset

System object: comm.gpu.ConvolutionalEncoder
Package: comm.gpu

Reset states of the convolutional encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalEncoder` object, `H`.

step

System object: comm.gpu.ConvolutionalEncoder
Package: comm.gpu

Convolutionally encode binary data

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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* × *L*, for a positive integer, *L*. The step method sets the length of the output vector, `Y`, to *L* × *N*.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ConvolutionalInterleaver System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To convolutionally interleave binary data:

1. Define and set up your convolutional interleaver object. See “Construction” on page 3-768.
2. Call `step` to convolutionally interleave according to the properties of `comm.gpu.ConvolutionalInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving with GPU

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.gpu.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.gpu.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans =

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
    10    10     4
    11     5     5
    12    12     6
    13     7     7
    14    14     8
    15     9     9
    16    16    10
    17    11    11
    18    18    12
    19    13    13
    20    20    14
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
intrlvDelay =

     6


numSymErrors =
     0
```

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`

Introduced in R2012a

reset

System object: comm.gpu.ConvolutionalInterleaver
Package: comm

Reset states of the convolutional interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalInterleaver` object, `H`.

step

System object: comm.gpu.ConvolutionalInterleaver
Package: comm

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To recover convolutionally interleaved binary data:

1. Define and set up your convolutional deinterleaver object. See “Construction” on page 3-776.

2 Call step to convolutionally deinterleave according to the properties of comm.gpu.ConvolutionalDeinterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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 on page 3-0 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

step	Permute input symbols using shift registers
reset	Reset states of the convolutional deinterleaver object

Common to All System Objects	
release	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving with GPU

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.gpu.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.gpu.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans =

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
    10    10     4
    11     5     5
    12    12     6
    13     7     7
    14    14     8
    15     9     9
    16    16    10
    17    11    11
    18    18    12
    19    13    13
    20    20    14
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
intrlvDelay =

     6
```

```
numSymErrors =

     0
```

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`

Introduced in R2012a

step

System object: comm.gpu.ConvolutionalDeinterleaver
Package: comm

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

reset

System object: comm.gpu.ConvolutionalDeinterleaver
Package: comm

Reset states of the convolutional deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalDeinterleaver` object, `H`.

comm.gpu.LDPCDecoder System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

This object performs LDPC decoding using the belief-passing or message-passing algorithm, implemented as the log-domain sum-product algorithm. For more information, see “Algorithm” on page 3-786. To decode a binary low-density parity-check code:

1. Define and set up your binary low-density parity-check decoder object. See “Construction” on page 3-784.
2. Call `step` to decode a binary low-density parity-check code according to the properties of `comm.gpu.LDPCDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`h = comm.gpu.LDPCDecoder` creates a GPU-based LDPC binary low-density parity-check decoder object, `h`. This object performs LDPC decoding based on the specified parity-check matrix. 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 $N > K > 0$. The last $N-K$ columns in the parity check matrix must be an invertible matrix in GF(2). 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*)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 logical data type. When you set this property to `Soft decision`, the output is log-likelihood ratios of single or double data type.

`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`.

`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.

`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

step	Decode input signal using LDPC decoding scheme

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

This object performs LDPC decoding using the belief-passing or message-passing algorithm, implemented as the log-domain sum-product algorithm. For more information, see the Decoding Algorithm section on the LDPC Decoder block reference page.

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.LDPCDecoder` | `comm.LDPCEncoder`

Introduced in R2012a

step

System object: comm.gpu.LDPCDecoder
Package: comm

Decode input signal using LDPC decoding scheme

Syntax

```
Y = step(H,X)
[Y,NUMITER] = step(H,X)
[Y,PARITY] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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.PSKDemodulator System object

Package: comm

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" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To demodulate a signal that was modulated using phase shift keying:

1. Define and set up your PSK demodulator object. See "Construction" on page 3-791.
2. Call `step` to demodulate the signal according to the properties of `comm.gpu.PSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

constellation	Calculate or plot ideal signal constellation
step	Demodulate using M-ary PSK method

Common to All System Objects	
`release`	Allow System object property value changes

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))
```

GPU PSK Demodulator

Create GPU PSK modulator and demodulator pair.

```
gpuMod = comm.gpu.PSKModulator;
gpuDemod = comm.gpu.PSKDemodulator;
```

Generate random data symbols. Modulate the data.

```
txData = randi([0 7],1000,1);
txSig = gpuMod(txData);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,20);
```

Demodulate the received signal.

```
rxData = gpuDemod(rxSig);
```

Determine the number of symbol errors.

```
numSymErrors = symerr(txData,rxData)
```

```
numSymErrors =

   736
```

See Also

comm.PSKDemodulator | comm.gpu.PSKModulator

Introduced in R2012a

constellation

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)
```

step

Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To modulate a signal using phase shift keying:

1. Define and set up your PSK modulator object. See "Construction" on page 3-802.
2. Call `step` to modulate the signal according to the properties of `comm.gpu.PSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

constellation	Calculate or plot ideal signal constellation
step	Modulate using M-ary PSK method with GPU

Common to All System Objects	
`release`	Allow System object property value changes

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

GPU PSK Modulator

Create binary data for 100, 4-bit symbols

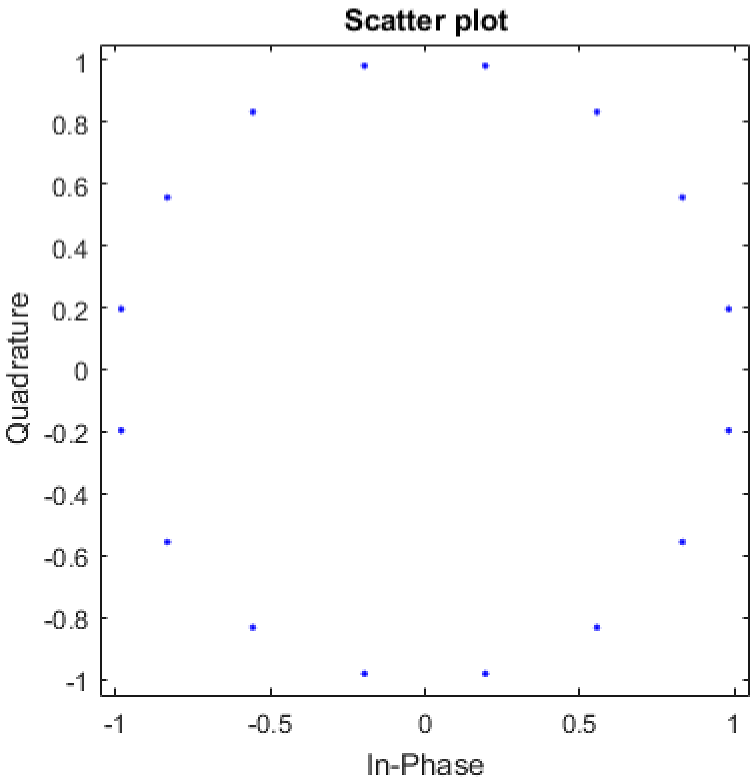
```
data = randi([0 1],400,1);
```

Create a 16-PSK modulator System object with bits as inputs and Gray-coded signal constellation. Change the phase offset to $\pi/16$.

```
gpuMod = comm.gpu.PSKModulator(16,'BitInput',true);
gpuMod.PhaseOffset = pi/16;
```

Modulate and plot the data

```
modData = gpuMod(data);
scatterplot(modData)
```



See Also

`comm.gpu.PSKDemodulator`

Introduced in R2012a

constellation

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)
```

step

Modulate using M-ary PSK method with GPU

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.TurboDecoder System object

Package: comm.gpu

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To decode an input signal using a turbo decoding scheme:

1. Define and set up your turbo decoder object. See “Construction” on page 3-810.
2. Call `step` to decode a binary signal according to the properties of `comm.gpu.TurboDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of the turbo decoder object
step	Decode input signal using parallel concatenated decoding scheme

Common to All System Objects	
release	Allow System object property value changes

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.

```
turboEnc = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

Create a BPSK Modulator System object.

```
bpsk = comm.BPSKModulator;
```

Create an AWGN Channel System object.

```
channel = 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.

```
turboDec = comm.gpu.TurboDecoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
      'NumIterations', 4);
```

Create an Error Rate System object.

```
errorRate = comm.ErrorRate;
```

Run the simulation.

```
for frmIdx = 1:8
 data = randi(s, [0 1], frmLen, 1);
 encodedData = turboEnc(data);
 modSignal = bpsk(encodedData);
 receivedSignal = channel(modSignal);
```

Convert the received signal to log-likelihood ratios for decoding.

```
receivedBits  = turboDec(-2/(noiseVar/2))*real(receivedSignal));
```

Compare original the data to the received data and then calculate the error rate results.

```
errorStats = errorRate(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.TurboDecoder` | `comm.TurboEncoder`

reset

System object: comm.gpu.TurboDecoder
Package: comm.gpu

Reset states of the turbo decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU TurboDecoder` object, `H`.

step

System object: comm.gpu.TurboDecoder
Package: comm.gpu

Decode input signal using parallel concatenated decoding scheme

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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*N_{Tails})/(2*N-1). M represents the input vector length and N_{Tails} is given by log2(`TrellisStructure.numStates`)*N. The output length, `L`, is the same as the length of the interleaver indices.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ViterbiDecoder System object

Package: comm

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” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input. 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. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To decode input symbols and produce binary output symbols:

1. Define and set up your Viterbi decoder object. See “Construction” on page 3-818.
2. Call `step` to decode input symbols according to the properties of `comm.gpu.ViterbiDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

info	Display information about GPU-based Viterbi Decoder object
reset	Reset states of the GPU-based Viterbi Decoder modulator object
step	Decode convolutionally encoded data using Viterbi algorithm

Common to All System Objects	
`release`	Allow System object property value changes

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`

Introduced in R2012a

info

System object: comm.gpu.ViterbiDecoder
Package: comm

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.

reset

System object: comm.gpu.ViterbiDecoder
Package: comm

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`.

step

System object: comm.gpu.ViterbiDecoder
Package: comm

Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To generate a Hadamard code:

1. Define and set up your Hadamard code object. See "Construction" on page 3-827.
2. Call `step` to generate a Hadamard according to the properties of `comm.HadamardCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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` on page 3-0 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`, ... , $N–1$.

`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` on page 3-0 property.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `int8`. The default is `double`.

Methods

reset	Reset states of Hadamard code generator object
step	Generate Hadamard code

Common to All System Objects	
release	Allow System object property value changes

Examples

Hadamard Code Sequence

Generate 10 samples of a Hadamard code sequence having a length of 128.

```
hadamard = comm.HadamardCode('Length',128,'SamplesPerFrame',10)
```

```
hadamard = 
  comm.HadamardCode with properties:

             Length: 128
              Index: 60
    SamplesPerFrame: 10
     OutputDataType: 'double'
```

```
seq = hadamard()
```

```
seq = 10×1

     1
     1
     1
     1
    -1
    -1
    -1
    -1
    -1
    -1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.OVSFCode` | `comm.WalshCode`

Introduced in R2012a

reset

System object: comm.HadamardCode
Package: comm

Reset states of Hadamard code generator object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `HadamardCode` object, `H`.

step

System object: comm.HadamardCode
Package: comm

Generate Hadamard code

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Detect errors in input data using CRC

Description

This HDL-optimized cyclic redundancy code (CRC) detector System object computes a checksum on the input data and compares the result against the input checksum. Instead of frame processing, the `HDLCRCDetector` System object processes streaming data. The object has frame synchronization control signals for both input and output data streams.

To compute and compare checksums:

1. Create the comm.HDLCRCDetector object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
CRCDet = comm.HDLCRCDetector
CRCDet = comm.HDLCRCDetector(Name,Value)
CRCDet = comm.HDLCRCDetector(poly,Name,Value)
```

Description

`CRCDet = comm.HDLCRCDetector` creates an HDL-optimized CRC detector System object, `CRCDet`, that detects errors in the input data according to a specified generator polynomial.

`CRCDet = comm.HDLCRCDetector(Name,Value)`sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
CRCDet = comm.HDLCRCDetector('Polynomial',[1 0 0 0 1 0 0 0 0], ...
'FinalXORValue',[1 1 0 0 0 0 0 0]);
```

specifies a CRC8 polynomial and an 8-bit value to XOR with the final checksum.

`CRCDet = comm.HDLCRCDetector(poly,Name,Value)` creates an HDL-optimized CRC detector System object, `CRCDet`, with the `Polynomial` property set to `poly`, and the other specified property names set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Polynomial` — Generator polynomial

`[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]` (default) | binary vector

Generator polynomial, specified as a binary vector, with coefficients in descending order of powers. The vector length must be equal to the degree of the polynomial plus 1.

`InitialState` — Initial conditions of shift register

`0` (default) | binary scalar | binary vector

Initial conditions of the shift register, specified as a binary, double-precision or single-precision scalar or vector. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

`DirectMethod` — Method of calculating checksum

`false` (default) | `true`

Method of calculating checksum, specified as a logical scalar. When this property is `true`, the object uses the direct algorithm for CRC checksum calculations.

To learn about direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

`ReflectInput` — Input byte order

`false` (default) | `true`

Input byte order, specified as a logical scalar. When this property is `true`, the object flips the input data on a bytewise basis before it enters the shift register.

`ReflectCRCChecksum` — Checksum byte order

`false` (default) | `true`

Checksum byte order, specified as a logical scalar. When this property is `true`, the object flips the output CRC checksum around its center.

`FinalXORValue` — Checksum mask

`0` (default) | binary scalar | binary vector

Checksum mask, specified as a binary, double- or single-precision data type scalar or vector. The object XORs the checksum with this value before appending the checksum to the input data. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
[Y,startOut,endOut,validOut,err] = CRCn(X,startIn,endIn,validIn)
```

Description

`[Y,startOut,endOut,validOut,err] = CRCn(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. If the two checksums are not equal, the output `err` is set to 1 `true`).

Input Arguments

`X` — Input message and appended checksum
binary column vector | scalar integer

Input message and appended checksum, specified as a binary vector or a scalar integer representing several bits. For example, vector input `[0,0,0,1,0,0,1,1]` is equivalent to `uint8` input `19`.

If the input is a vector, the data type can be double or logical. If the input is a scalar, the data type can be unsigned integer or unsigned fixed-point with 0 fractional bits (`fi([],0,N,0)`).

`X` can be part or all of the message to be checked.

The length of `X` must be less than or equal to the CRC length, and the CRC length must be divisible by the length of `X`.

The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical` | `unsigned fi`

`startIn` — Start of input message
logical scalar

Start of the input message, specified as a logical scalar.

`endIn` — End of input message
logical scalar

End of the input message, specified as a logical scalar.

`validIn` — Validity of input data
logical scalar

Validity of input data, specified as a logical scalar. When `validIn` is 1 (`true`), the object computes the CRC checksum for input X.

Output Arguments

Y — Message with checksum removed

binary column vector | scalar integer

Message with checksum removed, returned as a scalar integer or binary column vector with the same width and data type as input X.

`startOut` — Start of input message

logical scalar

Start of the input message, returned as a logical scalar.

`endOut` — End of input message

logical scalar

End of the input message, returned as a logical scalar.

`validOut` — Validity of input data

logical scalar

Validity of input data, returned as a logical scalar. When `validOut` is 1 (`true`), the output data Y is valid.

`err` — Checksum mismatch

logical scalar

Checksum mismatch, returned as a logical scalar. `err` is 1 (`true`) when the input checksum does not match the calculated checksum.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

CRC Encode and Decode for HDL

Encode and decode a signal using the HDL-optimized CRC generator and detector System objects. This example shows how to include each object in a function for HDL code generation.

Create a 32-bit message to be encoded, in two 16-bit columns.

```
msg = randi([0 1],16,2);
```

Run for 12 steps to accommodate the latency of both objects. Assign control signals for all steps. The first two samples are the valid data, and the remainder are processing latency.

```
numSteps = 12;
startIn = logical([1 0 0 0 0 0 0 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0 0 0 0 0 0 0]);
validIn = logical([1 1 0 0 0 0 0 0 0 0 0 0]);
```

Pass random input to the `HDLCRCGenerator` System object™ while it is processing the input message. The random data is not encoded because the input valid signal is `0` for steps 3 to 10.

```
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];
```

Write a function that creates and calls each System object™. You can generate HDL from these functions. The generator and detector objects both have a CRC length of 16 and use the default polynomial.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [dataOut,startOut,endOut,validOut] = HDLCRC16Gen(dataIn,startIn,endIn,validIn
%HDLCRC16Gen
% Generates CRC checksum using the comm.HDLCRCGenerator System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcg16;
  if isempty(crcg16)
    crcg16 = comm.HDLCRCGenerator()
  end
  [dataOut,startOut,endOut,validOut] = crcg16(dataIn,startIn,endIn,validIn);
end
```

```
function [dataOut,startOut,endOut,validOut,err] = HDLCRC16Det(dataIn,startIn,endIn,vali
%HDLCRC16Det
% Checks CRC checksum using the comm.HDLCRCDetector System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcd16;
  if isempty(crcd16)
    crcd16 = comm.HDLCRCDetector()
  end
  [dataOut,startOut,endOut,validOut,err] = crcd16(dataIn,startIn,endIn,validIn);
end
```

Call the CRC generator function. The encoded message is the original message plus a 16 bit checksum.

```
 for i =  1:numSteps
 [dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)] = ...
     HDLCRC16Gen(logical(dataIn(:,i)),startIn(i),endIn(i),validIn(i));
 end
```

```
crcg16 =
  comm.HDLCRCGenerator with properties:

              Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
            InitialState: 0
            DirectMethod: false
            ReflectInput: false
      ReflectCRCChecksum: false
           FinalXORValue: 0
```

Add noise by flipping a bit in the message.

```
dataOutNoise = dataOutGen;
dataOutNoise(2,4) = ~dataOutNoise(2,4);
```

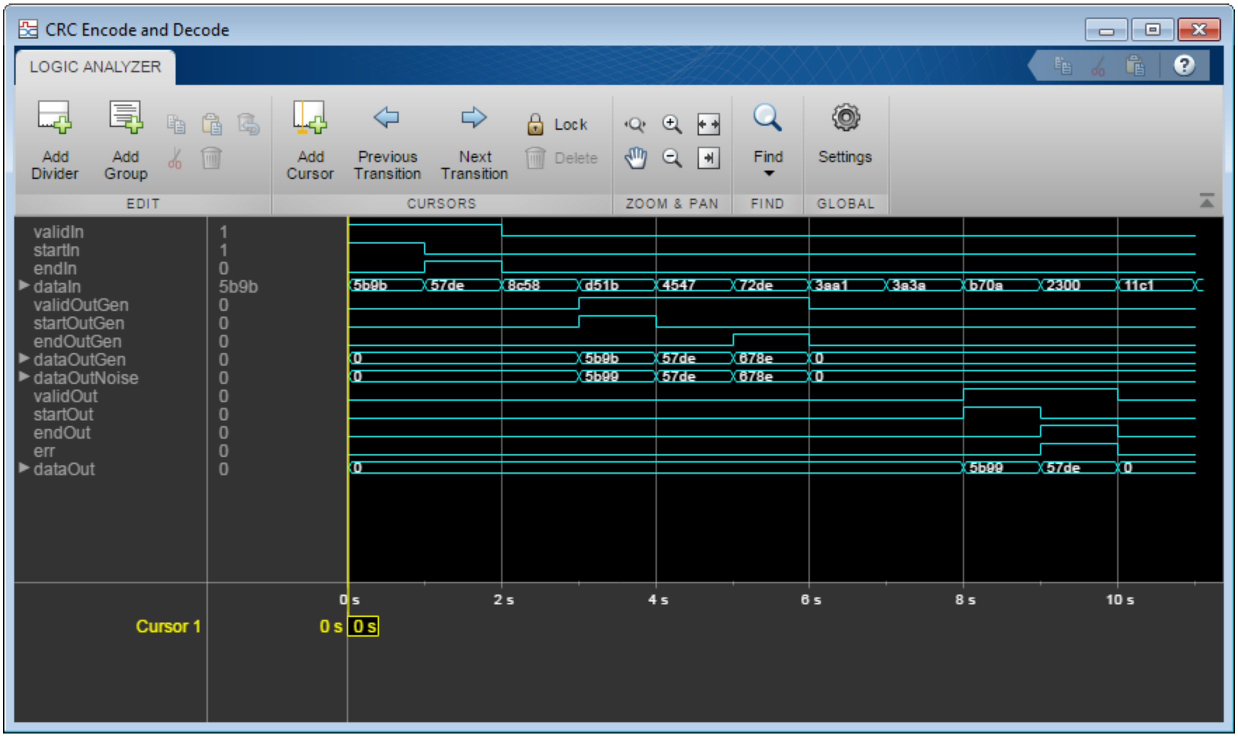
Call the CRC detector function. The output of the detector is the input message with the checksum removed. If the input checksum was not correct, the `err` flag is set with the last word of the output.

```
for i = 1:numSteps
[dataOut(:,i),startOut(i),endOut(i),validOut(i),err(i)] = ...
    HDLCRC16Det(logical(dataOutNoise(:,i)),startOutGen(i),endOutGen(i),validOutGen(i))
end
```

```
crcd16 =

  comm.HDLCRCDetector with properties:

              Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
            InitialState: 0
            DirectMethod: false
            ReflectInput: false
      ReflectCRCChecksum: false
           FinalXORValue: 0
```

Use the **Logic Analyzer** to view the input and output signals.

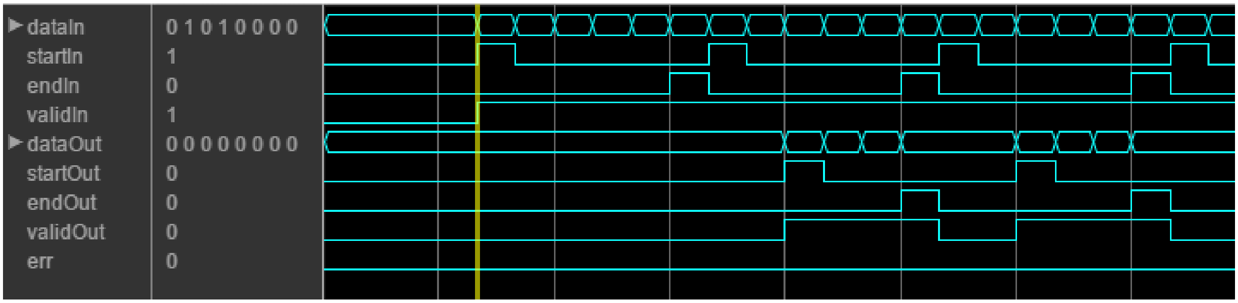
```
channels = {'validIn','startIn','endIn',...
    {'dataIn','Radix','Hexadecimal'},...
    'validOutGen','startOutGen','endOutGen',...
    {'dataOutGen','Radix','Hexadecimal'},...
    {'dataOutNoise','Radix','Hexadecimal'},...
    'validOut','startOut','endOut','err',...
    {'dataOut','Radix','Hexadecimal'}};
la = dsp.LogicAnalyzer('Name','CRC Encode and Decode','NumInputPorts',length(channels),
                      'BackgroundColor','Black','DisplayChannelHeight',8);

 for ii = 1:length(channels)
    if iscell(channels{ii})
        % Display data signals as hexadecimal integers
        c = channels{ii};
        modifyDisplayChannel(la,ii,'Name',c{1},c{2},c{3})
        % Convert binary column vector to integer
        dat2 = uint16(bi2de(eval(c{1})'));
        chanData{ii} = squeeze(dat2);
    else
        modifyDisplayChannel(la,ii,'Name',channels{ii})
        chanData{ii} = squeeze(eval(channels{ii})');
    end
 end
la(chanData{:})
```



Algorithms

Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. The input frames are contiguous, and the output frames show space between them because the detector block removes the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported.

Initial Delay

The `HDLCRCDetector` System object introduces a latency on the output. You can compute the latency as follows, assuming the input data is continuous:

```
initialDelay =  3 * (CRCLength/inputDataWidth) + 2
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

System Objects
`comm.CRCDetector` | `comm.HDLCRCGenerator`

Blocks
General CRC Syndrome Detector HDL Optimized

Introduced in R2012b

comm.HDLCRCGenerator System object

Package: comm

Generate CRC code bits and append to input data

Description

This HDL-optimized cyclic redundancy code (CRC) generator System object generates cyclic redundancy code (CRC) bits. Instead of frame processing, the `HDLCRCGenerator` System object processes streaming data. The object has frame synchronization control signals for both input and output data streams.

To generate cyclic redundancy code bits:

1. Create the comm.HDLCRCGenerator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
CRCGen = comm.HDLCRCGenerator
CRCGen = comm.HDLCRCGenerator(Name,Value)
CRCGen = comm.HDLCRCGenerator(poly,Name,Value)
```

Description

`CRCGen = comm.HDLCRCGenerator` creates an HDL-optimized CRC generator System object, `CRCGen`. This object generates CRC bits according to a specified generator polynomial and appends them to the input data.

`CRCGen = comm.HDLCRCGenerator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
CRCGen = comm.HDLCRCGenerator('Polynomial',[1 0 0 0 1 0 0 0 0], ...
'FinalXORValue',[1 1 0 0 0 0 0 0]);
```

specifies a CRC8 polynomial and an 8-bit value to XOR with the final checksum.

`CRCGen = comm.HDLCRCGenerator(poly,Name,Value)` sets the `Polynomial` property to `poly`, and the other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Polynomial` — Generator polynomial

`[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]` (default) | binary vector

Generator polynomial, specified as a binary vector, with coefficients in descending order of powers. The vector length must be equal to the degree of the polynomial plus 1.

`InitialState` — Initial conditions of shift register

`0` (default) | binary scalar | binary vector

Initial conditions of the shift register, specified as a binary, double-precision or single-precision scalar or vector. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

`DirectMethod` — Method of calculating checksum

`false` (default) | `true`

Method of calculating checksum, specified as a logical scalar. When this property is `true`, the object uses the direct algorithm for CRC checksum calculations.

To learn about direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

`ReflectInput` — Input byte order

`false` (default) | `true`

Input byte order, specified as a logical scalar. When this property is `true`, the object flips the input data on a bytewise basis before it enters the shift register.

`ReflectCRCChecksum` — Checksum byte order

`false` (default) | `true`

Checksum byte order, specified as a logical scalar. When this property is `true`, the object flips the output CRC checksum around its center.

`FinalXORValue` — Checksum mask

`0` (default) | binary scalar | binary vector

Checksum mask, specified as a binary, double- or single-precision data type scalar or vector. The object XORs the checksum with this value before appending the checksum to the input data. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
[Y,startOut,endOut,validOut] = CRCn(X,startIn,endIn, validIn)
```

Description

`[Y,startOut,endOut,validOut] = CRCn(X,startIn,endIn, validIn)` generates CRC checksums for input message X based on control signals and appends the checksums to X.

Input Arguments

X — Input message

binary column vector | scalar integer

Input message, specified as a binary vector or a scalar integer representing several bits. For example, vector input `[0,0,0,1,0,0,1,1]` is equivalent to `uint8` input `19`.

If the input is a vector, the data type can be double or logical. If the input is a scalar, the data type can be unsigned integer or unsigned fixed-point with 0 fractional bits (`fi([], 0,N,0)`).

X can be part or all of the message to be encoded.

The length of X must be less than or equal to the CRC length, and the CRC length must be divisible by the length of X.

The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical` | `unsigned fi`

`startIn` — Start of input message

logical scalar

Start of the input message, specified as a logical scalar.

`endIn` — End of input message

logical scalar

End of the input message, specified as a logical scalar.

`validIn` — Validity of input data

logical scalar

Validity of input data, specified as a logical scalar. When `validIn` is 1 (`true`), the object computes the CRC checksum for input `X`.

Output Arguments

Y — Output message with appended checksum

binary column vector | scalar integer

Output message, consisting of `X` with appended checksum, returned as a scalar integer or binary column vector with the same width and data type as input `X`.

`startOut` — Start of input message

logical scalar

Start of the input message, returned as a logical scalar.

`endOut` — End of input message

logical scalar

End of the input message, returned as a logical scalar.

`validOut` — Validity of input data

logical scalar

Validity of input data, returned as a logical scalar. When `validOut` is 1 (`true`), the output data `Y` is valid.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

CRC Encode and Decode for HDL

Encode and decode a signal using the HDL-optimized CRC generator and detector System objects. This example shows how to include each object in a function for HDL code generation.

Create a 32-bit message to be encoded, in two 16-bit columns.

```
msg = randi([0 1],16,2);
```

Run for 12 steps to accommodate the latency of both objects. Assign control signals for all steps. The first two samples are the valid data, and the remainder are processing latency.

```
numSteps = 12;
startIn = logical([1 0 0 0 0 0 0 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0 0 0 0 0 0 0]);
validIn = logical([1 1 0 0 0 0 0 0 0 0 0 0]);
```

Pass random input to the `HDLCRCGenerator` System object™ while it is processing the input message. The random data is not encoded because the input valid signal is `0` for steps 3 to 10.

```
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];
```

Write a function that creates and calls each System object™. You can generate HDL from these functions. The generator and detector objects both have a CRC length of 16 and use the default polynomial.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [dataOut,startOut,endOut,validOut] = HDLCRC16Gen(dataIn,startIn,endIn,validIn)
%HDLCRC16Gen
% Generates CRC checksum using the comm.HDLCRCGenerator System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.
```

```
  persistent crcg16;
  if isempty(crcg16)
    crcg16 = comm.HDLCRCGenerator()
  end
  [dataOut,startOut,endOut,validOut] = crcg16(dataIn,startIn,endIn,validIn);
end
```

```
function [dataOut,startOut,endOut,validOut,err] = HDLCRC16Det(dataIn,startIn,endIn,vali
%HDLCRC16Det
% Checks CRC checksum using the comm.HDLCRCDetector System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcd16;
  if isempty(crcd16)
    crcd16 = comm.HDLCRCDetector()
  end
  [dataOut,startOut,endOut,validOut,err] = crcd16(dataIn,startIn,endIn,validIn);
end
```

Call the CRC generator function. The encoded message is the original message plus a 16 bit checksum.

```
 for i =  1:numSteps
 [dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)] = ...
     HDLCRC16Gen(logical(dataIn(:,i)),startIn(i),endIn(i),validIn(i));
 end
```

```
crcg16 = 

  comm.HDLCRCGenerator with properties:

            Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
          InitialState: 0
          DirectMethod: false
          ReflectInput: false
    ReflectCRCChecksum: false
```

```
         FinalXORValue: 0
```

Add noise by flipping a bit in the message.

```
dataOutNoise = dataOutGen;
dataOutNoise(2,4) = ~dataOutNoise(2,4);
```

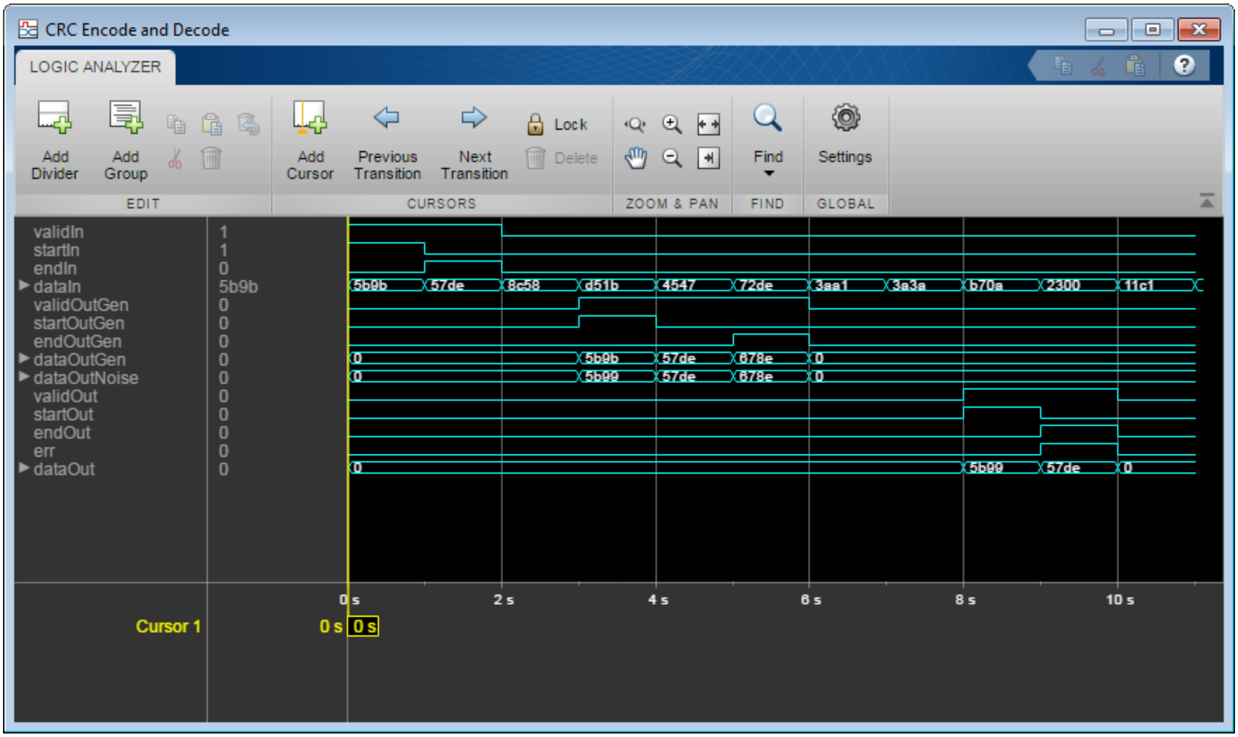
Call the CRC detector function. The output of the detector is the input message with the checksum removed. If the input checksum was not correct, the `err` flag is set with the last word of the output.

```
for i = 1:numSteps
[dataOut(:,i),startOut(i),endOut(i),validOut(i),err(i)] = ...
    HDLCRC16Det(logical(dataOutNoise(:,i)),startOutGen(i),endOutGen(i),validOutGen(i))
end
```

```
crcd16 = 

  comm.HDLCRCDetector with properties:

           Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
         InitialState: 0
         DirectMethod: false
         ReflectInput: false
    ReflectCRCChecksum: false
         FinalXORValue: 0
```

Use the **Logic Analyzer** to view the input and output signals.

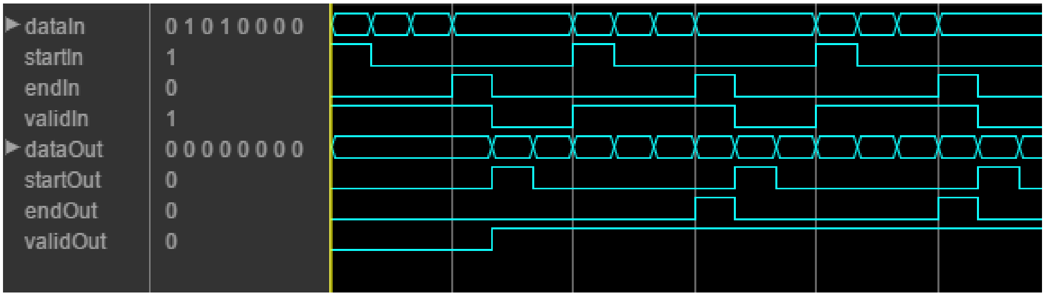
```
channels = {'validIn','startIn','endIn',...
    {'dataIn','Radix','Hexadecimal'},...
    'validOutGen','startOutGen','endOutGen',...
    {'dataOutGen','Radix','Hexadecimal'},...
    {'dataOutNoise','Radix','Hexadecimal'},...
    'validOut','startOut','endOut','err',...
    {'dataOut','Radix','Hexadecimal'}};
la = dsp.LogicAnalyzer('Name','CRC Encode and Decode','NumInputPorts',length(channels)
                      'BackgroundColor','Black','DisplayChannelHeight',8);

 for ii = 1:length(channels)
    if iscell(channels{ii})
        % Display data signals as hexadecimal integers
        c = channels{ii};
        modifyDisplayChannel(la,ii,'Name',c{1},c{2},c{3})
        % Convert binary column vector to integer
        dat2 = uint16(bi2de(eval(c{1})'));
        chanData{ii} = squeeze(dat2);
    else
        modifyDisplayChannel(la,ii,'Name',channels{ii})
        chanData{ii} = squeeze(eval(channels{ii})');
    end
 end
la(chanData{:})
```



Algorithms

Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. There must be enough space between the input frames to insert the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported. The output valid signal matches the input valid pattern.

Initial Delay

The `HDLCRCGenerator`System object introduces a latency on the output. You can compute the latency as follows, assuming the input data is continuous:

```
initialDelay = (CRCLength/inputDataWidth) + 2
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.CRCGenerator` | `comm.HDLCRCDetector`

Blocks
General CRC Generator HDL Optimized

Introduced in R2012a

comm.HDLRSDecoder System object

Package: comm

Decode message using Reed-Solomon decoder

Description

The HDL-optimized `HDLRSDecoder` System object recovers a message vector from a Reed-Solomon (RS) codeword vector. For proper decoding, the code and polynomial property values for this object must match those values in the corresponding encoder.

To recover a message vector from a Reed-Solomon codeword vector:

1. Create the comm.HDLRSDecoder object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Troubleshooting

- Each input frame must contain more than `(N-K)*2` symbols and fewer than or exactly `N` symbols. The object infers a shortened code when the number of valid data samples between `startIn` and `endIn` is less than `N`. A shortened code still requires `N` cycles to perform the Chien search. If the input message is less than `N` symbols, leave a guard interval of at least `N - size` inactive cycles before starting the next frame, where `size`is the message length.
- The decoder can operate on up to four messages at a time. If the object receives the start of a fifth message before completely decoding the first message, the object drops data samples from the first message. To avoid this issue, increase the number of inactive cycles between input messages.
- The generator polynomial is not specified explicitly. However, it is defined by the codeword length, the message length, and the B value for the starting exponent of the roots. To get the value of B from a generator polynomial, use the `genpoly2b` function.

Creation

Syntax

```
RSDec = comm.HDLRSDecoder
RSDec = comm.HDLRSDecoder(Name,Value)
RSDec = comm.HDLRSDecoder(N,K,Name,Value)
```

Description

`RSDec = comm.HDLRSDecoder` creates an HDL-optimized RS decoder System object, `RSDec`, that performs Reed-Solomon decoding.

`RSDec = comm.HDLRSDecoder(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
comm.HDLRSDecoder('BSource','Property','B',2)
```

sets a starting power of 2 for the roots of the primitive polynomial.

`RSDec = comm.HDLRSDecoder(N,K,Name,Value)` sets the `CodewordLength` property to `N`, the `MessageLength` property to `K`, and other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

B — Starting power for roots of primitive polynomial
1 (default) | positive integer

Starting power for roots of the primitive polynomial, specified as a positive integer.

Dependencies

The object uses this value when you set `BSource` to `'Property'`.

`BSource` — Source of starting power for roots of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the starting power for roots of the primitive polynomial, specified as either `'Property'` or `'Auto'`. When you select `'Auto'`, the object uses `B = 1`.

`CodewordLength` — Number of symbols, N, in RS codeword

7 (default) | positive integer

Number of symbols, *N*, in the RS codeword, specified as a positive integer. This value is rounded up to 2^M–1. *M* is the degree of the primitive polynomial, as specified by the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. The difference of `CodewordLength` – `MessageLength` must be an even integer.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

If you set `PrimitivePolynomialSource` to `'Auto'`, then `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16} – 1.

If you set `PrimitivePolynomialSource` to `'Property'`, then `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M– 1. *M* must be in the range 3 ≤ *M* ≤ 16.

`MessageLength` — Message length, K

3 (default) | positive integer

Message length, *K*, specified as a positive integer. The difference of `CodewordLength` – `MessageLength` must be an even integer.

`NumErrorsOutputPort` — Enable number of errors output argument

`false` (default) | `true`

When you set this property to `true`, the object returns the number of corrected errors. The number of corrected errors is not valid when `errOut` is `true`, since there were more errors than could be corrected.

`PrimitivePolynomialSource` — Source of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the primitive polynomial, specified as either `'Property'` or `'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 must specify a polynomial using the `PrimitivePolynomial` property.

`PrimitivePolynomial` — Primitive polynomial

`[1 0 1 1]` (default) | binary row vector

Primitive polynomial, specified as a binary row vector that represents a primitive polynomial over `gf`(2) of degree *M*, in descending order of powers. The polynomial defines the finite field `gf`(2^M) corresponding to the integers that form messages and codewords.

Dependencies

The object uses this value when you set `PrimitivePolynomialSource` to `'Property'`.

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
[Y,startOut,endOut,validOut,errOut] = RSDec(X,startIn,endIn,validIn)
[Y,startOut,endOut,validOut,errOut,numErrors] = RSDec(X,startIn,
endIn,validIn)
```

Description

`[Y,startOut,endOut,validOut,errOut] = RSDec(X,startIn,endIn,validIn)` decodes one encoded message symbol, `X`, and returns the decoded symbol `Y`. The `start` and `end` signals indicate the message frame boundaries. If `errOut` is `1` (`true`), then the object detected uncorrectable errors in the input frame.

`[Y,startOut,endOut,validOut,errOut,numErrors] = RSDec(X,startIn, endIn,validIn)` decodes the input data, and also returns the number of errors detected and corrected. To use this syntax, set the `NumErrorsOutputPort` property to `true`. If `errOut` is `1` (`true`), then the object detected uncorrectable errors in the output frame, and `numErrors` is invalid.

Input Arguments

X — Input message data or parity symbols

integer

Input message data and parity symbols, one symbol at a time, specified as an unsigned integer or `fi()` with any binary point scaling.

`double` type is allowed for simulation but not supported for HDL code generation.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startIn` — Start of input data frame

logical scalar

Start of input data frame, specified as a logical scalar.

Data Types: `logical`

`endIn` — End of input data frame

logical scalar

End of input data frame, specified as a logical scalar.

Data Types: `logical`

`validIn` — Validity of input data

logical scalar

Validity of input data, specified as a logical scalar.

Data Types: `logical`

Output Arguments

Y — Message data symbols

integer

Message data symbols, returned one symbol at a time, as an integer with the same data type as the input message, X.

Data Types: double | uint8 | uint16 | uint32 | fi

startOut — Start of output data frame
logical scalar

Start of output data frame, returned as a logical scalar.

Data Types: logical

endOut — End of output data frame
logical scalar

End of output data frame, returned as a logical scalar.

Data Types: logical

validOut — Validity of output data
logical scalar

Validity of output data, returned as a logical scalar.

Data Types: logical

errOut — Uncorrectable error in output data frame
logical scalar

Uncorrectable error in output data frame, returned as a logical scalar. This signal is 1 (true) when the message frame contains uncorrectable errors. In this case, the output data symbols are corrupted. This value is valid when endOut is 1 (true).

Data Types: logical

numErrors — Number of errors detected and corrected
integer

Number of errors detected and corrected, returned as an integer. This value is valid when endOut is 1 (true) and errOut is 0 (false).

Data Types: uint8

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Reed-Solomon Coding and Error Detection for HDL

Encode and decode a signal using Reed Solomon encoder and decoder System objects. This example shows how to include each object in a function for HDL code generation.

Create a random message to encode. This message is smaller than the codeword length to show how the objects support shortened codes. Pad the message with zeros to accommodate the latency of the decoder, including the Chien search.

```
messageLength = 188;
dataIn = [randi([0,255],1,messageLength,'uint8') zeros(1,1024-messageLength)];
```

Write a function that creates and calls a `HDLRSEncoder` System object™ with an RS(255,239) code. This code is used in the IEEE® 802.16 Broadband Wireless Access standard. `B` is the starting power of the roots of the primitive polynomial. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function  [dataOut,startOut,endOut,validOut] = HDLRSEnc80216(dataIn,startIn,endIn,vali
%HDLRSEnc80216
```

```
% Processes one sample of data using the comm.HDLRSEncoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsEnc80216;
  if isempty(rsEnc80216)
    rsEnc80216 = comm.HDLRSEncoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut] = rsEnc80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to encode the message.

```
for ii = 1:1024
    messageStart = (ii==1);
    messageEnd = (ii==messageLength);
    validIn = (ii<=messageLength);
    [encOut(ii),startOut(ii),endOut(ii),validOut(ii)] = ...
        HDLRSEnc80216(dataIn(ii),messageStart,messageEnd,validIn);
end
```

```
rsEnc80216 =

  comm.HDLRSEncoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
        PuncturePatternSource: 'None'
                      BSource: 'Property'
                            B: 0
```

Inject errors at random locations in the encoded message. Reed-Solomon can correct up to (N – K)/2 errors in each N symbols. So, in this example, the error correction capability is (255 – 239)/2=8 symbols.

```
numErrors = 8;
loc = randperm(messageLength,numErrors);
% encOut is qualified by validOut, use an offset for injecting errors
vi = find(validOut==true,1);
```

```
for i = 1:numErrors
   idx = loc(i)+vi;
   symbol = encOut(idx);
   encOut(idx) = randi([0 255],'uint8');
   fprintf('Symbol(%d): was 0x%x, now 0x%x\n',loc(i),symbol,encOut(idx))
end
```

```
Symbol(147): was 0x1f, now 0x82
Symbol(16): was 0x6b, now 0x82
Symbol(173): was 0x3, now 0xd1
Symbol(144): was 0x66, now 0xcb
Symbol(90): was 0x13, now 0xa4
Symbol(80): was 0x5a, now 0x60
Symbol(82): was 0x95, now 0xcf
Symbol(56): was 0xf5, now 0x88
```

Write a function that creates and calls a `HDLRSDecoder` System object™. This object must have the same code and polynomial as the encoder. You can generate HDL from this function.

```
function  [dataOut,startOut,endOut,validOut,err] = HDLRSDec80216(dataIn,startIn,endIn,v
%HDLRSDec80216
% Processes one sample of data using the comm.HDLRSDecoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsDec80216;
  if isempty(rsDec80216)
    rsDec80216 = comm.HDLRSDecoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut,err] = rsDec80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to detect errors in the encoded message.

```
for ii = 1:1024
 [decOut(ii),decStartOut(ii),decEndOut(ii),decValidOut(ii),decErrOut(ii)] = ...
     HDLRSDec80216(encOut(ii),startOut(ii),endOut(ii),validOut(ii));
end
```

```
rsDec80216 =
  comm.HDLRSDecoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
                      BSource: 'Property'
                            B: 0
          NumErrorsOutputPort: false
```

Select the valid decoder output and compare the decoded symbols to the original message.

```
decOut = decOut(decValidOut==1);
originalMessage = dataIn(1:messageLength);
if all(originalMessage==decOut)
    fprintf('All %d message symbols were correctly decoded.\n',messageLength)
else
   for jj = 1:messageLength
      if dataIn(jj)~=decOut(jj)
        fprintf('Error in decoded symbol(%d). Original 0x%x, Decoded 0x%x.\n',jj,dataIn
      end
   end
end
```

```
All 188 message symbols were correctly decoded.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation, these usage notes and limitations apply:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.HDLRSEncoder` | `comm.RSDecoder`

Blocks
Integer-Output RS Decoder HDL Optimized

Introduced in R2012b

comm.HDLRSEncoder System object

Package: comm

Encode message using Reed-Solomon encoder

Description

The HDL-optimized `HDLRSEncoder` System object creates a Reed-Solomon (RS) code with message and codeword lengths that you specify.

To encode a message using a Reed-Solomon code:

1. Create the comm.HDLRSEncoder object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
RSEnc = comm.HDLRSEncoder
RSEnc = comm.HDLRSEncoder(Name,Value)
RSEnc = comm.HDLRSEncoder(N,K,Name,Value)
```

Description

`RSEnc = comm.HDLRSEncoder` creates an HDL-optimized block encoder System object, `RSEnc`, that performs Reed-Solomon encoding in a streaming fashion for HDL.

`RSEnc = comm.HDLRSEncoder(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
comm.HDLRSEncoder('BSource','Property','B',2)
```

sets a starting power of 2 for the roots of the primitive polynomial.

`RSEnc = comm.HDLRSEncoder(N,K,Name,Value)` sets the `CodewordLength` property to `N`, the `MessageLength` property to `K`, and other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

B — Starting power for roots of primitive polynomial
`1` (default) | positive integer

Starting power for roots of the primitive polynomial, specified as a positive integer.

Dependencies

The object uses this value when you set `BSource` to `'Property'`.

BSource — Source of starting power for roots of primitive polynomial
`'Auto'` (default) | `'Property'`

Source of the starting power for roots of the primitive polynomial, specified as either `'Property'` or `'Auto'`. When you select `'Auto'`, the object uses `B = 1`.

CodewordLength — Number of symbols, N, in RS codeword
`7` (default) | positive integer

Number of symbols, *N*, in the RS codeword, specified as a positive integer. This value is rounded up to 2^M–1. *M* is the degree of the primitive polynomial, as specified by the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. The difference of `CodewordLength` – `MessageLength` must be an even integer.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

If you set `PrimitivePolynomialSource` to `'Auto'`, then `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16} – 1.

If you set `PrimitivePolynomialSource` to `'Property'`, then `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M– 1. M must be in the range $3 \leq M \leq 16$.

MessageLength — Message length, K

3 (default) | positive integer

Message length, K, specified as a positive integer. The difference of `CodewordLength` – `MessageLength` must be an even integer.

PrimitivePolynomialSource — Source of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the primitive polynomial, specified as either `'Property'` or `'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 must specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial — Primitive polynomial

`[1 0 1 1]` (default) | binary row vector

Primitive polynomial, specified as a binary row vector that represents a primitive polynomial over `gf`(2) of degree M, in descending order of powers. The polynomial defines the finite field `gf`(2^M) corresponding to the integers that form messages and codewords.

Dependencies

The object uses this value when you set `PrimitivePolynomialSource` to `'Property'`.

PuncturePatternSource — Source of puncture pattern

`'None'` (default) | `'Property'`

Source of the puncture pattern, specified as `'None'` or `'Property'`. 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` — Pattern used to puncture encoded data
`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Pattern used to puncture the encoded data, specified 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.

Dependencies

This property applies when you set the `PuncturePatternSource` property to `'Property'`.

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
[Y,startOut,endOut,validOut] = RSEnc(X,startIn,endIn,validIn)
```

Description

`[Y,startOut,endOut,validOut] = RSEnc(X,startIn,endIn,validIn)` encodes one input message symbol, `X`, and returns one symbol of encoded data, `Y`. The `start` and `end` signals indicate the message frame boundaries. The object returns associated parity symbols at the end of each message frame.

Input Arguments

X — Input message symbol
integer

Input message data, one symbol at a time, specified as an unsigned integer or `fi()` with any binary point scaling. The word length of each symbol must be `ceil(log`$_2$`(CodewordLength+1))`.

`double` type is allowed for simulation but not supported for HDL code generation.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startIn` — Start of input data frame
logical scalar

Start of input data frame, specified as a logical scalar.

Data Types: `logical`

`endIn` — End of input data frame
logical scalar

End of input data frame, specified as a logical scalar.

Data Types: `logical`

`validIn` — Validity of input data
logical scalar

Validity of input data, specified as a logical scalar.

Data Types: `logical`

Output Arguments

Y — Output message data and parity symbols
integer

Message data and parity symbols, returned one symbol at a time, as an integer with the same data type as the input message, X.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startOut` — Start of output data frame
logical scalar

Start of output data frame, returned as a logical scalar.

Data Types: `logical`

`endOut` — End of output data frame

logical scalar

End of output data frame, returned as a logical scalar.

Data Types: `logical`

`validOut` — Validity of output data

logical scalar

Validity of output data, returned as a logical scalar.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Reed-Solomon Coding and Error Detection for HDL

Encode and decode a signal using Reed Solomon encoder and decoder System objects. This example shows how to include each object in a function for HDL code generation.

Create a random message to encode. This message is smaller than the codeword length to show how the objects support shortened codes. Pad the message with zeros to accommodate the latency of the decoder, including the Chien search.

```
messageLength = 188;
dataIn = [randi([0,255],1,messageLength,'uint8') zeros(1,1024-messageLength)];
```

Write a function that creates and calls a `HDLRSEncoder` System object™ with an RS(255,239) code. This code is used in the IEEE® 802.16 Broadband Wireless Access standard. `B` is the starting power of the roots of the primitive polynomial. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function  [dataOut,startOut,endOut,validOut] = HDLRSEnc80216(dataIn,startIn,endIn,vali
%HDLRSEnc80216
% Processes one sample of data using the comm.HDLRSEncoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsEnc80216;
  if isempty(rsEnc80216)
    rsEnc80216 = comm.HDLRSEncoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut] = rsEnc80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to encode the message.

```
for ii = 1:1024
    messageStart = (ii==1);
    messageEnd = (ii==messageLength);
    validIn = (ii<=messageLength);
    [encOut(ii),startOut(ii),endOut(ii),validOut(ii)] = ...
        HDLRSEnc80216(dataIn(ii),messageStart,messageEnd,validIn);
end
```

```
rsEnc80216 =

  comm.HDLRSEncoder with properties:

             CodewordLength: 255
```

```
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
        PuncturePatternSource: 'None'
                      BSource: 'Property'
                            B: 0
```

Inject errors at random locations in the encoded message. Reed-Solomon can correct up to (*N* – *K*)/2 errors in each *N* symbols. So, in this example, the error correction capability is (255 – 239)/2=8 symbols.

```
numErrors = 8;
loc = randperm(messageLength,numErrors);
% encOut is qualified by validOut, use an offset for injecting errors
vi = find(validOut==true,1);
for i = 1:numErrors
   idx = loc(i)+vi;
   symbol = encOut(idx);
   encOut(idx) = randi([0 255],'uint8');
   fprintf('Symbol(%d): was 0x%x, now 0x%x\n',loc(i),symbol,encOut(idx))
end
```

```
Symbol(147): was 0x1f, now 0x82
Symbol(16): was 0x6b, now 0x82
Symbol(173): was 0x3, now 0xd1
Symbol(144): was 0x66, now 0xcb
Symbol(90): was 0x13, now 0xa4
Symbol(80): was 0x5a, now 0x60
Symbol(82): was 0x95, now 0xcf
Symbol(56): was 0xf5, now 0x88
```

Write a function that creates and calls a `HDLRSDecoder` System object™. This object must have the same code and polynomial as the encoder. You can generate HDL from this function.

```
function  [dataOut,startOut,endOut,validOut,err] = HDLRSDec80216(dataIn,startIn,endIn,v
%HDLRSDec80216
% Processes one sample of data using the comm.HDLRSDecoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsDec80216;
  if isempty(rsDec80216)
```

```
    rsDec80216 = comm.HDLRSDecoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut,err] = rsDec80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to detect errors in the encoded message.

```
for ii = 1:1024
 [decOut(ii),decStartOut(ii),decEndOut(ii),decValidOut(ii),decErrOut(ii)] = ...
     HDLRSDec80216(encOut(ii),startOut(ii),endOut(ii),validOut(ii));
end
```

```
rsDec80216 = 

  comm.HDLRSDecoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
                      BSource: 'Property'
                            B: 0
          NumErrorsOutputPort: false
```

Select the valid decoder output and compare the decoded symbols to the original message.

```
decOut = decOut(decValidOut==1);
originalMessage = dataIn(1:messageLength);
if all(originalMessage==decOut)
    fprintf('All %d message symbols were correctly decoded.\n',messageLength)
else
   for jj = 1:messageLength
      if dataIn(jj)~=decOut(jj)
        fprintf('Error in decoded symbol(%d). Original 0x%x, Decoded 0x%x.\n',jj,dataI
      end
   end
end
```

```
All 188 message symbols were correctly decoded.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.HDLRSDecoder` | `comm.RSEncoder`

Blocks
Integer-Input RS Encoder HDL Optimized

Introduced in R2012b

comm.HelicalDeinterleaver System object

Package: comm

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.

To helically deinterleave input symbols:

1. Define and set up your helical deinterleaver object. See "Construction" on page 3-878.
2. Call `step` to deinterleave input symbols according to the properties of `comm.HelicalDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of the helical deinterleaver object
step	Restore ordering of symbols using a helical array

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Helical Interleaving and Deinterleaving

Create helical interleaver and deinterleaver objects.

```
interleaver = comm.HelicalInterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
deinterleaver = comm.HelicalDeinterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
```

Generate random data. Interleave and then deinterleave the data.

```
[dataIn,dataOut] = deal([]);

for k = 1:10
    data = randi(7,6,1);
    intData = interleaver(data);
    deIntData = deinterleaver(intData);

    dataIn = cat(1,dataIn,data);
    dataOut = cat(1,dataOut,deIntData);
end
```

Determine the delay through the interleaver and deinterleaver pair.

```
intlvDelay = finddelay(dataIn,dataOut)
```

```
intlvDelay = 6
```

After taking the interleaving delay into account, confirm that the original and deinterleaved data are identical.

```
isequal(dataIn(1:end-intlvDelay),dataOut(1+intlvDelay:end))
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HelicalInterleaver` | `comm.MultiplexedDeinterleaver`

Introduced in R2012a

reset

System object: comm.HelicalDeinterleaver
Package: comm

Reset states of the helical deinterleaver object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `HelicalDeinterleaver` object, `H`.

step

System object: comm.HelicalDeinterleaver
Package: comm

Restore ordering of symbols using a helical array

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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*×*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*-1)×*s*, where *s* is the value for the StepSize property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.HelicalInterleaver System object

Package: comm

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.

To helically interleave input symbols:

1. Define and set up your helical interleaver object. See “Construction” on page 3-885.
2. Call `step` to interleave input symbols according to the properties of `comm.HelicalInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of the helical interleaver object
step	Permute input symbols using a helical array

Common to All System Objects	
release	Allow System object property value changes

Examples

Helical Interleaving and Deinterleaving

Create helical interleaver and deinterleaver objects.

```
interleaver = comm.HelicalInterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
deinterleaver = comm.HelicalDeinterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
```

Generate random data. Interleave and then deinterleave the data.

```
[dataIn,dataOut] = deal([]);

for k = 1:10
    data = randi(7,6,1);
    intData = interleaver(data);
    deIntData = deinterleaver(intData);

    dataIn = cat(1,dataIn,data);
    dataOut = cat(1,dataOut,deIntData);
end
```

Determine the delay through the interleaver and deinterleaver pair.

```
intlvDelay = finddelay(dataIn,dataOut)
```

```
intlvDelay = 6
```

After taking the interleaving delay into account, confirm that the original and deinterleaved data are identical.

```
isequal(dataIn(1:end-intlvDelay),dataOut(1+intlvDelay:end))
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HelicalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: comm.HelicalInterleaver
Package: comm

Reset states of the helical interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `HelicalInterleaver` object, `H`.

step

System object: comm.HelicalInterleaver
Package: comm

Permute input symbols using a helical array

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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*×*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*-1)×*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*×*N* symbols from the array by reading the next *N* rows sequentially.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.IntegerToBit System object

Package: comm

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.

To map integers to bits:

1. Define and set up your integer to bit object. See "Construction" on page 3-892.
2. Call `step` to map integers in the input vector to groups of bits in the output vector according to the properties of `comm.IntegerToBit`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

step	Convert vector of integers to vector of bits

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convert random integers to 4-bit words

```
hIntToBit = comm.IntegerToBit(4);
intData = randi([0 2^hIntToBit.BitsPerInteger-1],3,1);
bitData = step(hIntToBit,intData)
```

```
bitData = 12×1

     1
     1
     0
     1
     1
     1
     1
     0
     0
     0
       ⋮
```

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`

Introduced in R2012a

step

System object: comm.IntegerToBit
Package: comm

Convert vector of integers to vector of bits

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 -$(2^{(N\text{-}1)})$ and $(2^{(N\text{-}1)})$-1.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.IntegrateAndDumpFilter System object

Package: comm

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.

To integrate discrete-time signals with periodic resets:

1. Define and set up your integrate and dump filter object. See “Construction” on page 3-898.
2. Call `step` to integrate discrete-time signals according to the properties of `comm.IntegrateAndDumpFilter`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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` on page 3-0 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 “Fixed-Point Support for MATLAB System Objects in DSP System Toolbox” (DSP System Toolbox).

`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` on page 3-0 property. This property applies when you set the `FullPrecisionOverride` on page 3-0 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` on page 3-0 property to false and the `AccumulatorDataType` on page 3-0 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` on page 3-0 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` on page 3-0 property to false and the `OutputDataType` on page 3-0 property to `Custom`.

Methods

step	Integrate discrete-time signal with periodic resets

Common to All System Objects	
release	Allow System object property value changes

Examples

Pass Noisy Pulses Through Integrate and Dump Filter

Create an integrate and dump filter having an integration period of 20 samples.

```
intdump = comm.IntegrateAndDumpFilter(20);
```

Generate binary data.

```
d = randi([0 1],50,1);
```

Upsample the data, and pass it through an AWGN channel.

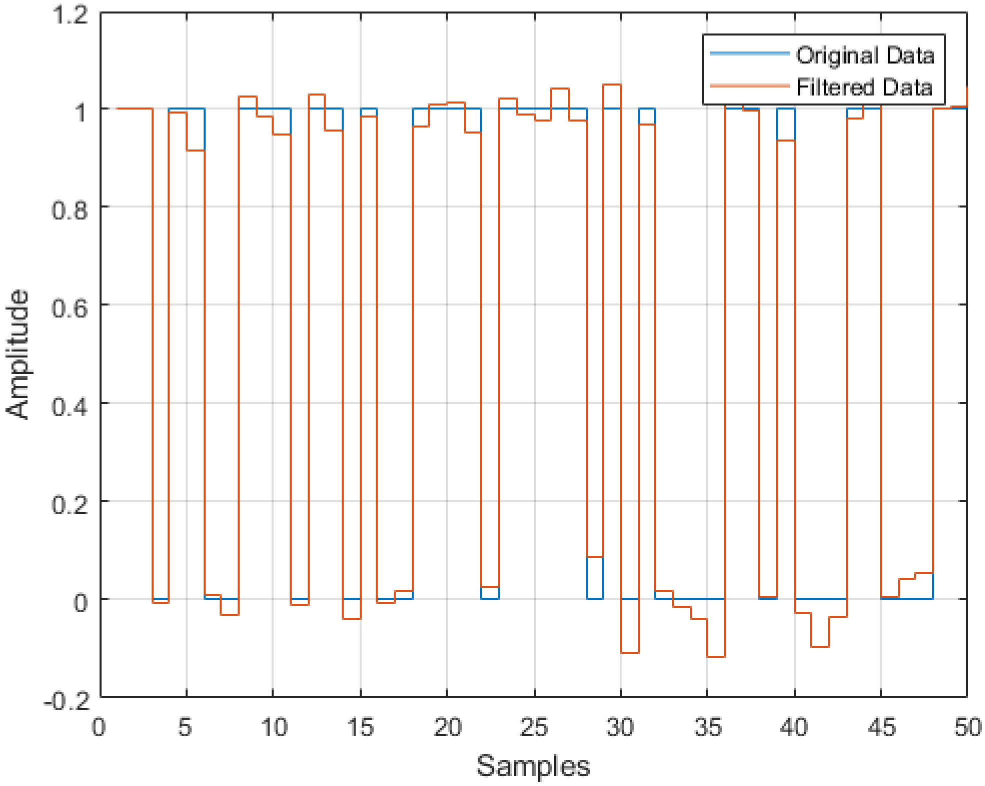
```
x = upsample(d,20);
y = awgn(x,25,'measured');
```

Pass the noisy data through the filter.

```
z = intdump(y);
```

Plot the original and filtered data. The integrate and dump filter removes most of the noise effects.

```
stairs([d z])
legend('Original Data','Filtered Data')
xlabel('Samples')
ylabel('Amplitude')
grid
```



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` on page 3-0 property.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

Introduced in R2012a

step

System object: comm.IntegrateAndDumpFilter
Package: comm

Integrate discrete-time signal with periodic resets

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To generate a Kasami sequence:

1. Define and set up your Kasami sequence object. See "Construction" on page 3-907.
2. Call `step` to generate a Kasami sequence according to the properties of `comm.KasamiSequence`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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 `'z^6 + z + 1'`.

You can specify the generator polynomial as a character vector or 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.

Lastly, 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` on page 3-0 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…$2^{n/2}$–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,…, 2^n–2], and *m* in [–1,…, $2^{n/2}$–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` on page 3-0 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` on page 3-0 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` on page 3-0 property.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `logical`. The default is `double`.

Methods

reset	Reset states of Kasami sequence generator object
step	Generate a Kasami sequence

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Spread BPSK Data with a Kasami Sequence

Spread BPSK data with a Kasami sequence of length 255 by using the Kasami sequence System object.

Generate binary data and apply BPSK modulation.

```
data = randi([0 1],10,1);
modData = pskmod(data,2);
```

Create a Kasami sequence object of length 255 using generator polynomial $x^8 + x^7 + x^4 + 1$.

```
kasamiSequence = comm.KasamiSequence('Polynomial',[8 7 4 0], ...
    'InitialConditions',[0 0 0 0 0 0 0 1],'SamplesPerFrame',255);
```

Generate the Kasami sequence and convert it to bipolar form.

```
kasSeq = kasamiSequence();
kasSeq = 2*kasSeq - 1;
```

Apply a gain of $1/\sqrt{255}$ to ensure that the spreading operation does not increase the overall signal power.

```
kasSeq = kasSeq/sqrt(255);
```

Spread the BPSK data using the Kasami sequence.

```
spreadData = modData*kasSeq';
spreadData = spreadData(:);
```

Verify that the spread data sequence is 255 times longer than the input data sequence.

```
spreadingFactor = length(spreadData)/length(data)
```

```
spreadingFactor = 255
```

Verify that the spreading operation did not increase the signal power.

```
spreadSigPwr = sum(abs(spreadData).^2)/length(data)
```

```
spreadSigPwr = 1.0000
```

Change the generator polynomial of the Kasami sequence generator to $x^8 + x^3 + 1$ after first releasing the object. Use the character representation of the polynomial.

```
release(kasamiSequence)
kasamiSequence.Polynomial = 'x^8 + x^3 + 1';
```

Generate a new sequence and convert it to bipolar form.

```
kasSeq = kasamiSequence();
kasSeq = 2*kasSeq - 1;
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GoldSequence` | `comm.PNSequence`

reset

System object: comm.KasamiSequence
Package: comm

Reset states of Kasami sequence generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `KasamiSequence` object, `H`.

step

System object: comm.KasamiSequence
Package: comm

Generate a Kasami sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Decode binary low-density parity-check code

Description

The `LDPCDecoder` object decodes a binary low-density parity-check code.

This object performs LDPC decoding using the belief-passing or message-passing algorithm, implemented as the log-domain sum-product algorithm. For more information, see "Algorithms" on page 3-919. To decode a binary low-density parity-check code:

1. Define and set up your binary low-density parity-check decoder object. See "Construction" on page 3-916.
2. Call `step` to decode a binary low-density parity-check code according to the properties of `comm.LDPCDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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(P)` creates an LDPC decoder object, *h*, where the input `P` specifies the parity check matrix.

Properties

`ParityCheckMatrix` — Parity-check matrix

`dvbs2ldpc(1/2)` (default) | binary sparse matrix | non-sparse index matrix

Specify the parity-check matrix as a binary valued sparse matrix `P` with dimension (N – K)-by-N, where $N > K > 0$. The last $N-K$ columns in the parity check matrix must be an invertible matrix in GF(2). 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.

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

`'Information part'` (default) | `'Whole codeword'`

Specify the output value format as `'Information part'` or `'Whole codeword'`. 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–K)-by-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

`'Hard decision'` (default) | `'Soft decision'`

Specify the decision method used for decoding as either `'Hard decision'` or `'Soft decision'`. When you set this property to `'Hard decision'`, the output is decoded bits of data type `double` or `logical`. When you set this property to `'Soft decision'`, the output is log-likelihood ratios of data type `double`.

MaximumIterationCount — Maximum number of decoding iterations

`50` (default) | positive integer

Specify the maximum number of iterations the object uses as a positive integer.

IterationTerminationCondition — Condition for iteration termination

`'Maximum iteration count'` (default) | `'Parity check satisfied'`

Specify the condition to stop the decoding iterations as either `'Maximum iteration count'` or `'Parity check satisfied'`. 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.

NumIterationsOutputPort — Output number of iterations performed

`false` (default) | `true`

To output the number of iterations performed, set this property to `true`.

FinalParityChecksOutputPort — Output final parity checks

`false` (default) | `true`

To output the final calculated parity checks, set this property to `true`.

Methods

step	Decode input using LDPC decoding scheme

Common to All System Objects	
`release`	Allow System object property value changes

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 performs LDPC decoding using the belief-passing or message-passing algorithm, implemented as the log-domain sum-product algorithm. For more information, see the Decoding Algorithm section on the LDPC Decoder block reference page.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Using default properties, `comm.LDPCDecoder` does not support code generation. To generate code, specify the `ParityCheckMatrix` property as a non-sparse row-column index matrix.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHDecoder` | `comm.LDPCEncoder` | `comm.gpu.LDPCDecoder`

Introduced in R2012a

step

System object: comm.LDPCDecoder
Package: comm

Decode input using LDPC decoding scheme

Syntax

```
Y = step(H,X)
[Y,NUMITER] = step(H,X)
[Y,PARITY] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.LDPCEncoder System object

Package: comm

Encode binary low-density parity-check code

Description

The `LDPCEncoder` object encodes a binary low-density parity-check code.

To encode a binary low-density parity-check code:

1. Define and set up your binary low-density parity-check encoder object. See "Construction" on page 3-923.
2. Call `step` to encode a binary low-density parity-check code according to the properties of `comm.LDPCEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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('ParityCheckMatrix',Value)` creates an LDPC encoder object, *h*, with the `ParityCheckMatrix` property set to the specified value.

`h = comm.LDPCEncoder(P)` creates an LDPC encoder object, *h*, where the input `P` specifies the parity check matrix.

Properties

`ParityCheckMatrix`

Parity-check matrix

Specify the parity-check matrix as a binary valued sparse matrix `P` with dimension (N – K) by N, where $N > K > 0$. The last $N-K$ columns in the parity check matrix must be an invertible matrix in GF(2). 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

step	Encode input using LDPC coding scheme

Common to All System Objects	
`release`	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHEncoder` | `comm.LDPCDecoder`

Introduced in R2012a

step

System object: comm.LDPCEncoder
Package: comm

Encode input using LDPC coding scheme

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.LTEMIMOChannel System object

Package: comm

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.

To filter an input signal using an LTE MIMO multipath fading channel:

1. Define and set up your LTE MIMO multipath fading channel object. See “Construction” on page 3-928.
2. Call `step` to filter the input signal using an LTE MIMO multipath fading channel according to the properties of `comm.LTEMIMOChannel`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 5Hz` | `EVA 5Hz` | `EVA 70Hz` | `ETU 70Hz` | `ETU 300Hz`, which are supported in the LTE specification Release 10. The default value of this property is `EPA 5Hz`.

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

Excess tap delay [ns]	Relative power [db]
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
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

reset	Reset states of the `LTEMIMOChannel` object
step	Filter input signal through LTE MIMO multipath fading channel

Common to All System Objects	
release	Allow System object property value changes

Examples

Configure MIMO Channel Object Using LTE MIMO Channel 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.

```
pskModulator = comm.PSKModulator;
modData = pskModulator(randi([0 pskModulator.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.

```
lteChan = comm.LTEMIMOChannel(...
    'Profile',              'EVA 5Hz',...
    'AntennaConfiguration', '2x2',...
    'CorrelationLevel',     'Medium',...
    'AntennaSelection',     'Off',...
    'RandomStream',         'mt19937ar with seed',...
    'Seed',                 99,...
    'PathGainsOutputPort',  true);
```

Filter the modulated data using the LTEMIMOChannel System object, lteChan.

```
[LTEChanOut,LTEPathGains] = lteChan(channelInput);
```

Create an equivalent MIMOChannel System object, mimoChannel, using the properties of the LTEMIMOChannel System object, lteChan.

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.

```
mimoChannel = comm.MIMOChannel( ...
    'SampleRate',lteChan.SampleRate, ...
    'PathDelays',lteChan.PathDelays, ...
    'AveragePathGains',lteChan.AveragePathGains, ...
    'NormalizePathGains',lteChan.NormalizePathGains, ...
    'FadingDistribution',lteChan.FadingDistribution, ...
    'MaximumDopplerShift',lteChan.MaximumDopplerShift, ...
    'DopplerSpectrum',lteChan.DopplerSpectrum, ...
    'SpatialCorrelationSpecification', ...
         lteChan.SpatialCorrelationSpecification, ...
    'SpatialCorrelationMatrix',lteChan.SpatialCorrelationMatrix, ...
    'AntennaSelection',lteChan.AntennaSelection, ...
    'NormalizeChannelOutputs',lteChan.NormalizeChannelOutputs, ...
    'RandomStream',lteChan.RandomStream, ...
    'Seed',lteChan.Seed, ...
    'PathGainsOutputPort',lteChan.PathGainsOutputPort);
```

Filter the modulated data using the equivalent `mimoChannel` object.

```
[MIMOChanOut, MIMOPathGains] = mimoChannel(channelInput);
```

Verify that the channel output and the path gain output from the two objects are the same.

```
sameChOutput = isequal(LTEChanOut,MIMOChanOut)
```

```
sameChOutput = logical
   0
```

```
samePathGains = isequal(LTEPathGains,MIMOPathGains)
```

```
samePathGains = logical
   1
```

You can repeat the preceding process with `AntennaConfiguration` set to `4x2` or `4x4` and `CorrelationLevel` set to `Medium` or `High` for `lteChan`.

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	$R_{\mathrm{eNB}} = 1$	$R_{eNB} = \begin{pmatrix} 1 & \alpha \\ \alpha^{*} & 1 \end{pmatrix}$	$R_{eNB} = \left(\begin{matrix} 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{1/9^{*}} & 1 & \alpha^{1/9} \\ \alpha^{4/9^{*}} & \alpha^{1/9^{*}} & 1 \\ \alpha^{*} & \alpha^{4/9^{*}} & \alpha^{1/9^{*}} \end{matrix}\right.$

The following table defines the receiver UE correlation matrix.

	One Antenna	Two Antennas	Four Antennas
UE Correlation	$R_{\mathrm{UE}} = 1$	$R_{UE} = \begin{pmatrix} 1 & \beta \\ \beta^{*} & 1 \end{pmatrix}$	$R_{UE} = \left(\begin{matrix} 1 & \beta^{1/9} & \beta^{4/9} \\ \beta^{1/9^{*}} & 1 & \beta^{1/9} \\ \beta^{4/9^{*}} & \beta^{1/9^{*}} & 1 \\ \beta^{*} & \beta^{4/9^{*}} & \beta^{1/9^{*}} \end{matrix}\right.$

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	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha \\ \alpha^* & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \beta^* \\ \alpha^* \\ \alpha^*\beta^* \end{bmatrix}$
4-by-2	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{\frac{1}{9}} & \alpha^{\frac{4}{9}} & \alpha \\ \alpha^{\frac{1}{9}^*} & 1 & \alpha^{\frac{1}{9}} & \alpha^{\frac{4}{9}} \\ \alpha^{\frac{4}{9}^*} & \alpha^{\frac{1}{9}^*} & 1 & \alpha^{\frac{1}{9}} \\ \alpha^* & \alpha^{\frac{4}{9}^*} & \alpha^{\frac{1}{9}^*} & 1 \end{bmatrix} \otimes$
4-by-4	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{\frac{1}{9}} & \alpha^{\frac{4}{9}} & \alpha \\ \alpha^{\frac{1}{9}^*} & 1 & \alpha^{\frac{1}{9}} & \alpha^{\frac{4}{9}} \\ \alpha^{\frac{4}{9}^*} & \alpha^{\frac{1}{9}^*} & 1 & \alpha^{\frac{1}{9}} \\ \alpha^* & \alpha^{\frac{4}{9}^*} & \alpha^{\frac{1}{9}^*} & 1 \end{bmatrix} \otimes$

Spatial Correlation Correction

Low Correlation		Medium Correlation		High Correlation	
α	β	α	β	α	β
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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.MIMOChannel`

Introduced in R2012a

reset

System object: comm.LTEMIMOChannel
Package: comm

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.

step

System object: comm.LTEMIMOChannel
Package: comm

Filter input signal through LTE MIMO multipath fading channel

Syntax

```
Y = step(H,X)
[Y,PATHGAINS] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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- or single-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` contains complex values with same precision as input signal.

`[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 contains complex values with same precision as input signal.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MatrixDeinterleaver System object

Package: comm

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.

To deinterleave input symbols using a permutation vector:

1. Define and set up your matrix deinterleaver object. See “Construction” on page 3-942.
2. Call `step` to deinterleave the input signal according to the properties of `comm.MatrixDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Deinterleave input symbols using permutation matrix

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Matrix Interleaving and Deinterleaving

Create matrix interleaver and deinterleaver objects.

```
interleaver = comm.MatrixInterleaver('NumRows',2,'NumColumns', 5);
deinterleaver = comm.MatrixDeinterleaver('NumRows',2,'NumColumns', 5);
```

Generate random data, interleave, and then deinterleave the data.

```
data = randi(7,10,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.BlockDeinterleaver` | `comm.MatrixInterleaver`

Introduced in R2012a

step

System object: comm.MatrixDeinterleaver
Package: comm

Deinterleave input symbols using permutation matrix

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MatrixInterleaver System object

Package: comm

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.

To perform block interleaving using a permutation matrix:

1. Define and set up your matrix interleaver object. See "Construction" on page 3-946.
2. Call `step` to interleave the input symbols according to the properties of `comm.MatrixInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Permute input symbols using permutation matrix

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Matrix Interleaving and Deinterleaving

Create matrix interleaver and deinterleaver objects.

```
interleaver = comm.MatrixInterleaver('NumRows',2,'NumColumns', 5);
deinterleaver = comm.MatrixDeinterleaver('NumRows',2,'NumColumns', 5);
```

Generate random data, interleave, and then deinterleave the data.

```
data = randi(7,10,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BlockInterleaver` | `comm.MatrixDeinterleaver`

Introduced in R2012a

step

System object: comm.MatrixInterleaver
Package: comm

Permute input symbols using permutation matrix

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MatrixHelicalScanDeinterleaver System object

Package: comm

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.

To deinterleave the input symbols by filling a matrix with the input symbols helically and then outputting the matrix contents row-by-row:

1. Define and set up your matrix helical scan deinterleaver object. See “Construction” on page 3-950.
2. Call `step` to deinterleave the input signal according to the properties of `comm.MatrixHelicalScanDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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

step	Deinterleave input symbols by filling a matrix along diagonals

Common to All System Objects	
release	Allow System object property value changes

Examples

Matrix Helical Scan Interleaving and Deinterleaving

Create matrix helical scan interleaver and deinterleaver objects.

```
interleaver = comm.MatrixHelicalScanInterleaver('NumRows',4,'NumColumns', 4);
deinterleaver = comm.MatrixHelicalScanDeinterleaver('NumRows',4,'NumColumns',4);
```

Generate random symbols. Pass the data through the interleaver, and then pass that data through the deinterleaver.

```
data = randi(7,16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intData deIntData]
```

```
ans = 16×3

     6     6     6
     7     1     7
     1     2     1
     7     1     7
     5     5     5
     1     7     1
     2     6     2
     4     7     4
     7     7     7
     7     4     7
      ⋮
```

Confirm that the original and deinterleaved sequences are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BlockDeinterleaver` | `comm.MatrixHelicalScanInterleaver`

Introduced in R2012a

step

System object: comm.MatrixHelicalScanDeinterleaver
Package: comm

Deinterleave input symbols by filling a matrix along diagonals

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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 helically. The number of rows and number of columns properties are the dimensions of the matrix that the object uses internally for computations.

To interleave the input signal by filling a matrix row-by-row with the input symbols and then outputting the matrix contents helically:

1. Define and set up your matrix helical scan interleaver object. See "Construction" on page 3-955.
2. Call `step` to interleave the input signal according to the properties of `comm.MatrixHelicalScanInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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

step	Permute input symbols by selecting matrix elements along diagonals

Common to All System Objects	
release	Allow System object property value changes

Examples

Matrix Helical Scan Interleaving and Deinterleaving

Create matrix helical scan interleaver and deinterleaver objects.

```
interleaver = comm.MatrixHelicalScanInterleaver('NumRows',4,'NumColumns', 4);
deinterleaver = comm.MatrixHelicalScanDeinterleaver('NumRows',4,'NumColumns',4);
```

Generate random symbols. Pass the data through the interleaver, and then pass that data through the deinterleaver.

```
data = randi(7,16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intData deIntData]
```

```
ans = 16×3

     6     6     6
     7     1     7
     1     2     1
     7     1     7
     5     5     5
     1     7     1
     2     6     2
     4     7     4
     7     7     7
     7     4     7
      ⋮
```

Confirm that the original and deinterleaved sequences are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BlockInterleaver` | `comm.MatrixHelicalScanDeinterleaver`

Introduced in R2012a

step

System object: comm.MatrixHelicalScanInterleaver
Package: comm

Permute input symbols by selecting matrix elements along diagonals

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MemorylessNonlinearity System object

Package: comm

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.

To apply memoryless nonlinearity to the input signal:

1. Define and set up your memoryless nonlinearity object. See “Construction” on page 3-960.
2. Call `step` to apply memoryless nonlinearity according to the properties of `comm.MemorylessNonlinearity`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to a real scalar value, which is greater than the `PowerLowerLimit` on page 3-0 property and of double- or single-precision data type. This property applies when you set the `Method` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Rapp model`. The default is `1`. This property is tunable.

Methods

step	Apply memoryless nonlinearity to input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Apply Saleh Model Nonlinearity to 16-QAM Signal

Create a 16-QAM modulator having an average power of 10 mW.

```
qamModulator = comm.RectangularQAMModulator('ModulationOrder',16, ...
    'NormalizationMethod','Average power','AveragePower',1e-2);
```

Create Memoryless Nonlinearity System object using the Saleh model.

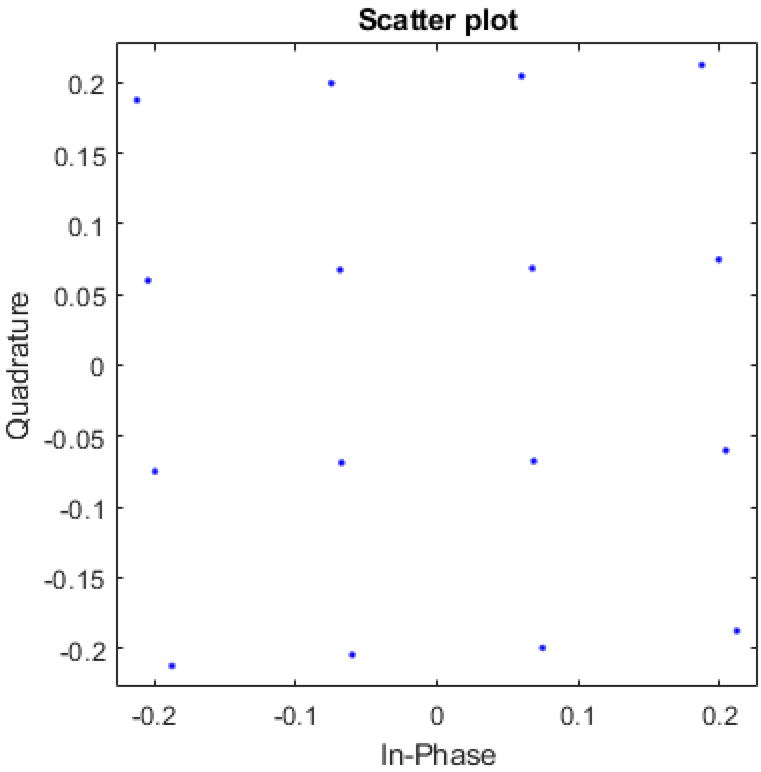
```
saleh = comm.MemorylessNonlinearity('Method','Saleh model');
```

Generate modulated symbols and pass them through the nonlinearity model.

```
modData = qamModulator(randi([0 15],1000,1));
y = saleh(modData);
```

Plot the resultant scatter plot.

```
scatterplot(y)
```



Nonlinear Amplifier Gain Compression

Plot the gain compression of a nonlinear amplifier for a 16-QAM signal.

Specify the modulation order and samples per symbol parameters.

```
M = 16;
sps = 4;
```

To model a nonlinear amplifier, create a memoryless nonlinearity object having a 30 dB third order intercept point. Create a raised cosine transmit filter.

```
amplifier = comm.MemorylessNonlinearity('IIP3',30);

txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.3, ...
    'FilterSpanInSymbols',6, ...
    'OutputSamplesPerSymbol',sps, ...
    'Gain',sqrt(sps));
```

Specify the input power in dBm. Convert the input power to W and initialize the gain vector.

```
pindBm = -5:25;
pin = 10.^((pindBm-30)/10);
gain = zeros(length(pindBm),1);
```

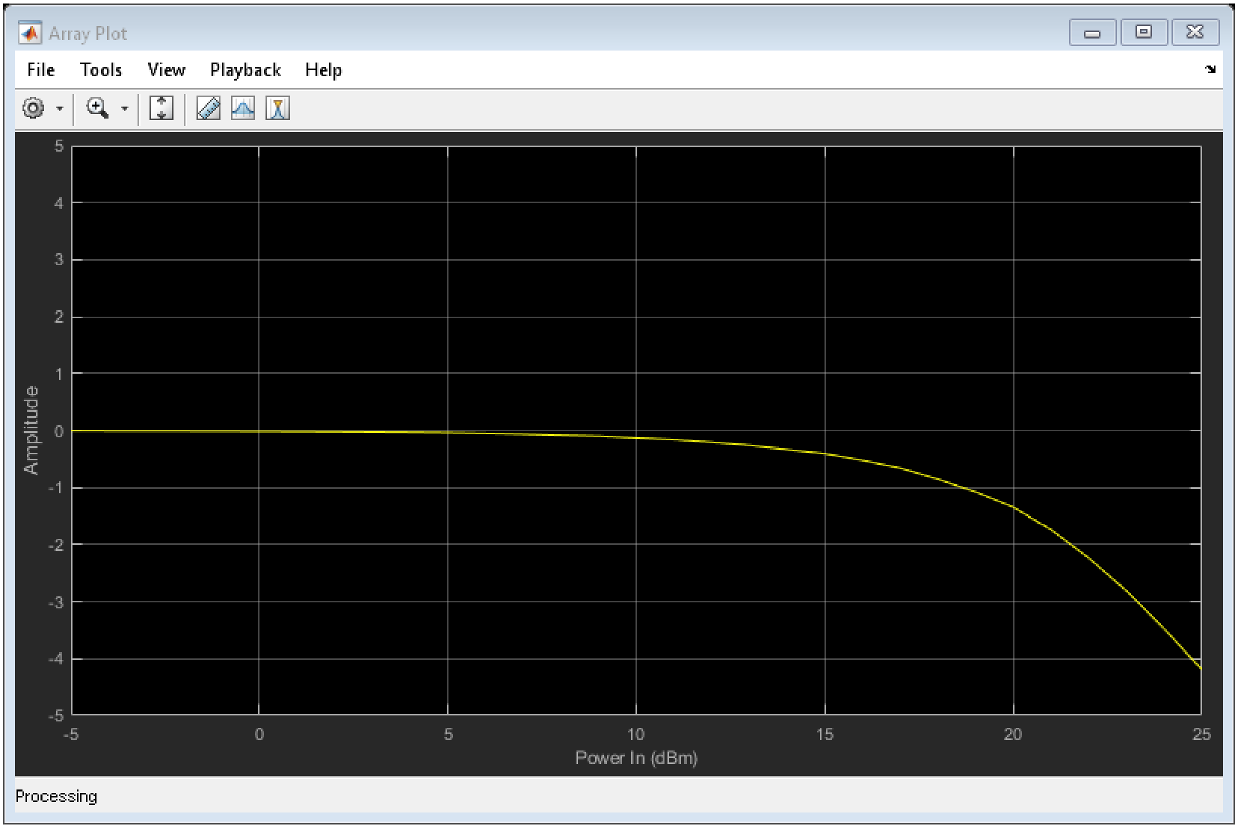
Execute the main processing loop, which includes these steps:

- Generating random data symbols
- Modulating the data and adjusting the average power
- Filtering the modulated signal
- Amplifying the signal
- Measuring the gain

```
for k = 1:length(pin)
    data = randi([0 M-1],1000,1);
    modSig = qammod(data,M,'UnitAveragePower',true)*sqrt(pin(k));
    filtSig = txfilter(modSig);
    ampSig = amplifier(filtSig);
    gain(k) = 10*log10(var(ampSig)/var(filtSig));
end
```

Plot the amplifier gain as a function of the input signal power.

```
arrayplot = dsp.ArrayPlot('PlotType','Line','XLabel','Power In (dBm)', ...
'XOffset',-5,'YLimits',[-5 5]);

arrayplot(gain)
```



The 1 dB gain compression point occurs for an input power of 18.5 dBm. To increase the point at which a 1 dB compression is observed, increase the third order intercept point, `amplifier.IIP3`.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PhaseNoise`

Introduced in R2012a

step

System object: comm.MemorylessNonlinearity
Package: comm

Apply memoryless nonlinearity to input signal

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MER System object

Package: comm

Measure modulation error ratio

Description

The `comm.MER` (modulation error ratio) object measures the signal-to-noise ratio (SNR) in digital modulation applications. You can use MER measurements to determine system performance in communications applications. For example, determining whether a DVB-T system conforms to applicable radio transmission standards requires accurate MER measurements. The block measures all outputs in dB.

To measure modulation error ratio:

1. Define and set up your MER object. See “Construction” on page 3-970.
2. Call `step` to measure the modulation error ratio according to the properties of `comm.MER`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`MER = comm.MER` creates a modulation error ratio (MER) System object, `MER`. This object measures the signal-to-noise ratio (SNR) in digital modulation applications.

`MER = comm.MER(Name,Value)` creates an `MER` object 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`).

Example: `MER = comm.MER('ReferenceSignalSource','Estimated from reference constellation')` creates an object, `MER`, that measures the MER of a received signal by using a reference constellation.

Properties

`ReferenceSignalSource`

Reference signal source

Reference signal source, specified as either `'Input port'` (default) or `'Estimated from reference constellation'`. To provide an explicit reference signal against which the input signal is measured, set this property to `'Input port'`. To measure the MER of the input signal against a reference constellation, set this property to `'Estimated from reference constellation'`.

`ReferenceConstellation`

Reference constellation

Reference constellation, specified as a vector. This property is available when the `ReferenceSignalSource` property is `'Estimated from reference constellation'`.

The default is `[0.7071 - 0.7071i; -0.7071 - 0.7071i; -0.7071 + 0.7071i; 0.7071 + 0.7071i]`, which corresponds to a standard QPSK constellation. You can derive constellation points by using modulation functions or objects. For example, to derive the reference constellation for a 16-QAM signal, you can use `qammod(0:15,16)`.

`MeasurementIntervalSource`

Measurement interval source

Measurement interval source, specified as one of the following: `'Input length'` (default), `'Entire history'`, `'Custom'`, or `'Custom with periodic reset'`. This property affects the RMS and maximum MER outputs only.

- To calculate MER using only the current samples, set this property to `'Input length'`.
- To calculate MER for all samples, set this property to `'Entire history'`.

- To calculate MER over an interval you specify and to use a sliding window, set this property to `'Custom'`.
- To calculate MER over an interval you specify and to reset the object each time the measurement interval is filled, set this property to `'Custom with periodic reset'`.

MeasurementInterval

Measurement interval

Measurement interval over which the MER is calculated, specified in samples as a real positive integer. This property is available when `MeasurementIntervalSource` is `'Custom'` or `'Custom with periodic reset'`. The default is `100`.

AveragingDimensions

Averaging dimensions

Averaging dimensions, specified as a positive integer or row vector of positive integers. This property determines the dimensions over which the averaging is performed. For example, to average across the rows, set this property to `2`. The default is `1`.

The object supports variable-size inputs over the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must remain constant between `step` calls. For example, if the input has size `[4 3 2]` and `Averaging dimensions` is `[1 3]`, the output size is `[1 3 1]`, and the second dimension must remain fixed at `3`.

MinimumMEROutputPort

Minimum MER measurement output port

Minimum MER measurement output port, specified as a logical scalar. To create an output port for minimum MER measurements, set this property to `true`. The default is `false`.

XPercentileMEROutputPort

X-percentile MER measurement output port

X-percentile MER measurement output port, specified as a logical scalar. To create an output port for *X*-percentile MER measurements, set this property to `true`. The *X*-

percentile MER measurements persist until you reset the object. These measurements are calculated by using all of the input frames since the last reset. The default is `false`.

`XPercentileValue`

X-percentile value

X-percentile value above which *X*% of the MER measurements fall, specified as a real scalar from `0` to `100`. This property applies when `XPercentileMEROutputPort` is `true`. The default is `95`.

`SymbolCountOutputPort`

Symbol count output port

Symbol count output port, specified as a logical scalar. To output the number of accumulated symbols used to calculate the *X*-percentile MER measurements, set this property to `true`. This property is available when `XPercentileMEROutputPort` property is `true`. The default is `false`.

Methods

reset	Reset states of MER measurement object
step	Measure modulation error ratio

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Measure MER of Noisy 16-QAM Modulated Signal

Create an MER object which outputs minimum MER, 90-percentile MER, and the number of symbols.

```
mer = comm.MER('MinimumMEROutputPort',true, ...
    'XPercentileMEROutputPort',true,'XPercentileValue',90,...
    'SymbolCountOutputPort',true);
```

Generate random data. Apply 16-QAM modulation having unit average power. Pass the signal through an AWGN channel.

```
data = randi([0 15],1000,1);
refsym = qammod(data,16,'UnitAveragePower',true);
rxsym = awgn(refsym,20);
```

Determine the RMS, minimum, and 90th percentile MER values.

```
[MERdB,MinMER,PercentileMER,NumSym] = mer(refsym,rxsym)
```

```
MERdB = 20.1071

MinMER = 11.4248

PercentileMER = 16.5850

NumSym = 1000
```

Measure MER Using Reference Constellation

Generate random data symbols, and apply 8-PSK modulation.

```
d = randi([0 7],2000,1);
txSig = pskmod(d,8,pi/8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,30);
```

Create an MER object. Measure the MER using the transmitted signal as the reference.

```
mer = comm.MER;
mer1 = mer(txSig,rxSig);
```

Release the MER object. Set the object to use a reference constellation for making MER measurements.

```
release(mer)
mer.ReferenceSignalSource = 'Estimated from reference constellation';
mer.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Measure the MER using only the received signal as an input. Verify that it matches the result obtained with a reference signal.

```
mer2 = mer(rxSig);
[mer1 mer2]
```

```
ans = 1×2

   30.0271   30.0271
```

Measure MER Using Custom Measurement Interval

Measure the MER of a noisy 8-PSK signal using two types of custom measurement intervals. Display the results.

Set the number of frames, M, and the number of subframes per frame, K.

```
M = 2;
K = 5;
```

Set the number of symbols in a subframe. Calculate the corresponding frame length.

```
sfLen = 100;
frmLen = K*sfLen
```

```
frmLen = 500
```

Create an MER object. Configure the object to use a custom measurement interval equal to the frame length.

```
mer1 = comm.MER('MeasurementIntervalSource','Custom', ...
    'MeasurementInterval',frmLen);
```

Configure the object to measure MER using an 8-PSK reference constellation.

```
mer1.ReferenceSignalSource = 'Estimated from reference constellation';
mer1.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Create an MER object, and configure it use a 500-symbol measurement interval with a periodic reset. Configure the object to measure MER using an 8-PSK reference constellation.

```
mer2 = comm.MER('MeasurementIntervalSource','Custom with periodic reset', ...
    'MeasurementInterval',frmLen);
mer2.ReferenceSignalSource = 'Estimated from reference constellation';
mer2.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Initialize the MER and signal-to-noise arrays.

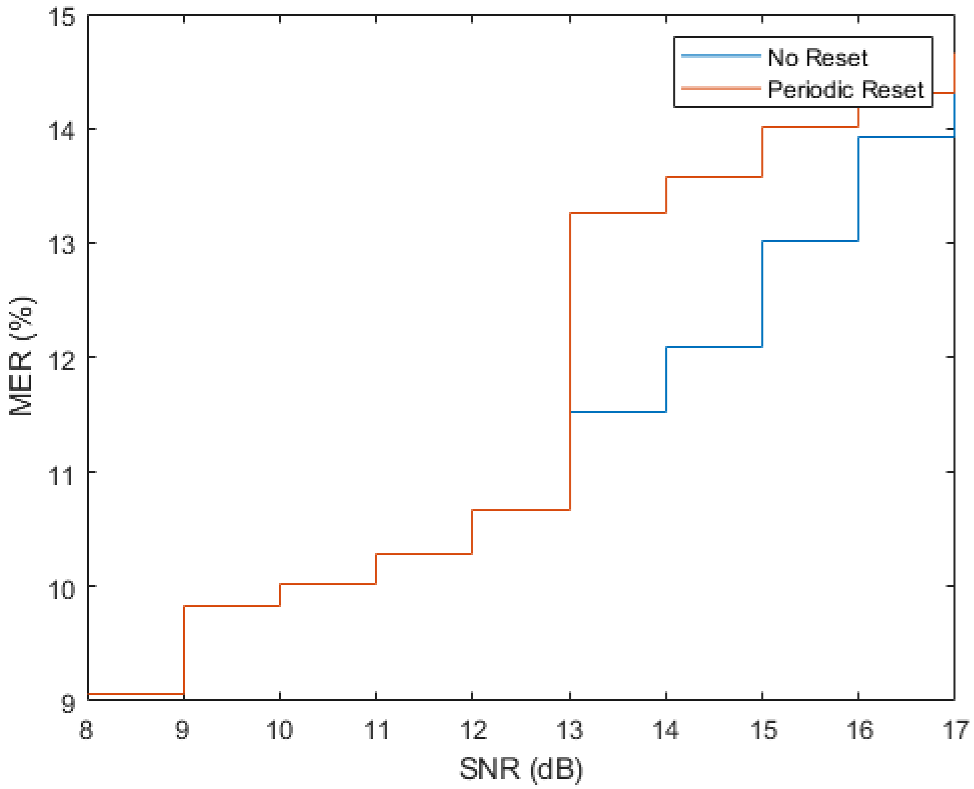
```
merNoReset = zeros(K,M);
merReset = zeros(K,M);
snrdB = zeros(K,M);
```

Measure the MER for a noisy 8-PSK signal using both objects. The SNR is increases by 1 dB from subframe to subframe. For `merNoReset`, the 500 most recent symbols are used to compute the estimate. In this case, a sliding window is used so that an entire data frame is used as the basis for the estimate. For `merReset`, the symbols are cleared each time a new frame is encountered.

```
for m = 1:M
    for k = 1:K
        data = randi([0 7],sfLen,1);
        txSig = pskmod(data,8,pi/8);
        snrdB(k,m) = k+(m-1)*K+7;
        rxSig = awgn(txSig,snrdB(k,m));
        merNoReset(k,m) = mer1(rxSig);
        merReset(k,m) = mer2(rxSig);
    end
end
```

Display the MER measured using the two approaches. The windowing used in the first case provides an averaging across the subframes. In the second case, the MER object resets after the first frame so that the calculated MER values more accurately reflect the current SNR.

```
stairs(snrdB(:),[merNoReset(:) merReset(:)])
xlabel('SNR (dB)')
ylabel('MER (%)')
legend('No Reset','Periodic Reset')
```



Measure MER Across Different Dimensions

Create OFDM modulator and demodulator objects.

```
ofdmmod = comm.OFDMModulator('FFTLength',32,'NumSymbols',4);
ofdmdemod = comm.OFDMDemodulator('FFTLength',32,'NumSymbols',4);
```

Determine the number of subcarriers and symbols in the OFDM signal.

```
ofdmDims = info(ofdmmod);
numSC = ofdmDims.DataInputSize(1)
numSC = 21
```

```
numSym = ofdmDims.DataInputSize(2)
```

```
numSym = 4
```

Generate random symbols and apply QPSK modulation.

```
msg = randi([0 3],numSC,numSym);
modSig = pskmod(msg,4,pi/4);
```

OFDM modulate the QPSK signal. Pass the signal through an AWGN channel. Demodulate the noisy signal.

```
txSig = ofdmmod(modSig);
rxSig = awgn(txSig,10,'measured');
demodSig = ofdmdemod(rxSig);
```

Create an MER object, where the result is averaged over the subcarriers. Measure the MER. There are four entries corresponding to each of the 4 OFDM symbols.

```
mer = comm.MER('AveragingDimensions',1);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 1×4

   11.2338   12.5315   12.8882   12.7015
```

Overwrite the MER object, where the result is averaged over the OFDM symbols. Measure the MER. There are 21 entries corresponding to each of the 21 subcarriers.

```
mer = comm.MER('AveragingDimensions',2);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 21×1

   10.8054
   14.9655
   14.5721
   13.6024
   13.0132
   12.1391
   10.4012
    9.5017
    8.8055
   13.3824
      ⋮
```

Measure the MER and average over both the subcarriers and the OFDM symbols.

```
mer = comm.MER('AveragingDimensions',[1 2]);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 12.2884
```

Algorithms

MER is a measure of the SNR in a modulated signal calculated in dB. The MER over N symbols is

$$MER = 10 * \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}\left(e_k\right)}\right) \text{dB.}$$

The MER for the kth symbol is

$$MER_k = 10 * \log_{10}\left(\frac{\frac{1}{N}\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{e_k}\right) \text{dB.}$$

The minimum MER represents the minimum MER value in a burst, or

$$MER_{\min} = \min_{k\in[1,\ldots,N]}\left\{MER_k\right\},$$

where:

- $e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k = In-phase measurement of the *k*th symbol in the burst
- Q_k = Quadrature phase measurement of the *k*th 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 the *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 fall.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.CCDF` | `comm.EVM`

Introduced in R2012a

reset

System object: comm.MER
Package: comm

Reset states of MER measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MER` object, `H`.

step

System object: comm.MER
Package: comm

Measure modulation error ratio

Syntax

```
MERDB= step(MER,REFSYM,RXSYM)
MERDB = step(MER,RXSYM)
[___,MINMER] = step(___)
[___,XMER] = step(___)
[___,NUMSYM] = step(___)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`MERDB= step(MER,REFSYM,RXSYM)` returns the measured MER, `MERDB`, of the received signal `RXSYM`, based on reference signal `REFSYM`. MER values are measured in dB.

`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. All outputs of the object are of data type `double`. To set the interval over which the MER is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`MERDB = step(MER,RXSYM)` returns the measured MER of received signal `RXSYM` based on a reference signal specified in the `ReceivedConstellation` property.

`[___,MINMER] = step(___)` returns the minimum MER, `MINMER`, given either of the two previous syntaxes.

To return minimum MER, set the `MinimumMEROutputPort` property to `true`. To set the interval over which `MINMER` is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`[ ___ ,XMER] = step( ___ )` returns the *X*-percentile MER, `XMER`.

To return the *X*-percentile MER, set the `XPercentileMEROutputPort` property to `true`. `XMER` is the MER above which *X*% of the measurements fall, where *X* is set by the `XPercentileValue` property. `XMER` is measured using all the input frames since the last reset.

`[ ___ ,NUMSYM] = step( ___ )` returns the number of symbols, `NUMSYM`, used to calculate the *X*-percentile MER.

To return `NUMSYM`, set the `SymbolCountOutputPort` to `true`. `NUMSYM` is measured using all the input frames since the last reset.

Note `MER` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MIMOChannel System object

Package: comm

Filter input signal through MIMO multipath fading channel

Description

A comm.MIMOChannel 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. For processing details, see the Algorithms on page 3-1016 section.

To filter an input signal through a MIMO multipath fading channel:

1. Create the comm.MIMOChannel object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
mimochan = comm.MIMOChannel
mimochan = comm.MIMOChannel(Name,Value)
```

Description

`mimochan = comm.MIMOChannel` creates a multiple-input multiple-output (MIMO) frequency-selective or frequency-flat fading channel System object. This object filters a real or complex input signal through the multipath MIMO channel to obtain the channel-impaired signal.

`mimochan = comm.MIMOChannel(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.MIMOChannel('SampleRate',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`SampleRate` — Input signal sample rate

`1` (default) | positive scalar

Input signal sample rate in hertz, specified as a positive scalar.

Data Types: `double`

`PathDelays` — Discrete path delay

`0` (default) | scalar | row vector

Discrete path delay in seconds, specified as a scalar or row vector.

- 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.

Data Types: `double`

`AveragePathGains` — Average path gains (dB)

`0` (default) | scalar | row vector

Average path gains in decibels, specified as a scalar or row vector. `AveragePathGains` must have the same size as PathDelays.

Data Types: `double`

`NormalizePathGains` — Normalize path gains

`true` (default) | `false`

Normalize path gains, specified as `true` or `false`.

- When you set this property to `true`, the fading processes are normalized so that the total power of the path gains, averaged over time, is `0` dB.
- 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.

Data Types: `logical`

`FadingDistribution` — Fading distribution

`'Rayleigh'` (default) | `'Rician'`

Fading distribution to use for the channel, specified as `'Rayleigh'` or `'Rician'`.

Data Types: `char`

`KFactor` — K-factor of Rician fading channel

`3` (default) | positive scalar | row vector

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of positive-valued elements. N_P equals number of path delays specified by the PathDelays property.

- If you set `KFactor` to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of `KFactor`. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set `KFactor` to 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.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`DirectPathDopplerShift` — Doppler shifts for line-of-sight components (Hz)

`0` (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This property must have the same size as KFactor.

- If you set `DirectPathDopplerShift` to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set `DirectPathDopplerShift` to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of `DirectPathDopplerShift` that correspond to positive elements in the KFactor vector.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`DirectPathInitialPhase` — Initial phases for line-of-sight components (Radians)

`0` (default) | scalar | row vector

Initial phases for the line-of-sight components of the Rician fading channel in radians, specified as a scalar or row vector. This property must have the same size as KFactor.

- If you set `DirectPathInitialPhase` to a scalar, it represents the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set `DirectPathInitialPhase` to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the elements of `DirectPathInitialPhase` that correspond to positive elements in the KFactor vector.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`MaximumDopplerShift` — Maximum Doppler shift for all channel paths (Hz)

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

The Doppler shift applies to all channel paths. When you set this property to `0`, the channel remains static for the entire input. You can use the `reset` object function to generate a new channel realization.

`MaximumDopplerShift` must be smaller than (SampleRate/10)/f_c for each path, where f_c represents the cutoff frequency factor of the path. For more information on the cutoff frequency, see Cutoff Frequency Factor on page 3-1016.

Data Types: `double`

`DopplerSpectrum` — Doppler spectrum shape for all channel paths
`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this property is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.
- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the PathDelays property.

The maximum Doppler shift value necessary to specify the Doppler spectrum/spectra is given by the MaximumDopplerShift property.

Dependencies

This property applies when MaximumDopplerShift is greater than zero.

If you assign the FadingTechnique property to `'Sum of sinusoids'`, you must set `DopplerSpectrum` to `doppler('Jakes')`.

`SpatialCorrelationSpecification` — Spatial correlation specification
`'Separate Tx Rx'` (default) | `'None'` | `'Combined'`

Spatial correlation specification, specified as `'Separate Tx Rx'`, `'None'`, or `'Combined'`.

- Choose `'Spatial Tx Rx'` to separately specify the transmit and receive spatial correlation matrices from which the number of transmit antenna (N_T) and number of receive antennas (N_R) are derived.

- Choose `'None'` to specify the number of transmit and receive antennas.
- Choose `'Combined'` to specify a single correlation matrix for the whole channel, from which the product of N_T and N_R is derived.

Data Types: `char`

`NumTransmitAntennas` — Number of transmit antennas

2 (default) | positive integer

Number of transmit antennas, specified as a positive integer.

Dependencies

This property applies when SpatialCorrelationSpecification is `'None'` or `'Combined'`.

Data Types: `double`

`NumReceiveAntennas` — Number of receive antennas

2 (default) | positive integer

Number of receive antennas, specified as a positive integer.

Dependencies

This property applies when SpatialCorrelationSpecification is `'None'` or `'Combined'`.

Data Types: `double`

`TransmitCorrelationMatrix` — Spatial correlation of transmitter

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the transmitter as an N_T-by-N_T matrix or N_T-by-N_T-by-N_P array. N_T is the number of transmit antennas, and N_P equals the value of the PathDelays property.

- If `PathDelays` is a scalar, the channel is frequency-flat, and `TransmitCorrelationMatrix` is an N_T-by-N_T Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If `PathDelays` is a vector, the channel is frequency selective, and you can specify `TransmitCorrelationMatrix` as a matrix. Each path has the same transmit spatial correlation matrix.
- Alternatively, you can specify `TransmitCorrelationMatrix` as an N_T-by-N_T-by-N_P array, where each path can have its own different transmit spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Separate Tx Rx'`.

Data Types: `double`
Complex Number Support: Yes

`ReceiveCorrelationMatrix` — Spatial correlation of receiver

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the receiver as an N_R-by-N_R matrix or N_R-by-N_R-by-N_P array. N_R is the number of receive antennas, and N_P equals the value of the PathDelays property.

- If PathDelays is a scalar, the channel is frequency flat, and `ReceiveCorrelationMatrix` is an N_R-by-N_R Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If PathDelays is a vector, the channel is frequency selective, and you can specify `ReceiveCorrelationMatrix` as a matrix. Each path has the same receive spatial correlation matrix.
- Alternatively, you can specify `ReceiveCorrelationMatrix` as an N_R-by-N_R-by-N_P array, where each path can have its own different receive spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Separate Tx Rx'`.

Data Types: `double`
Complex Number Support: Yes

`SpatialCorrelationMatrix` — Combined spatial correlation matrix

`[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]` (default) | matrix | 3-D array

Combined spatial correlation matrix, specified as an N_{TR}-by-N_{TR} matrix or N_{TR}-by-N_{TR}-by-N_P array, where $N_{TR} = (N_T \times N_R)$, and N_P equals the value of the PathDelays property.

- If PathDelays is a scalar, the channel is frequency flat, and `SpatialCorrelationMatrix` is an N_{TR}-by-N_{TR} Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.

- If PathDelays is a vector, the channel is frequency selective, and you can specify `SpatialCorrelationMatrix` as a matrix. Each path has the same spatial correlation matrix.
- Alternatively, you can specify `SpatialCorrelationMatrix` as an N_{TR}-by-N_{TR}-by-N_P array, where each path can have its own different combined spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Combined'`.

Data Types: `double`

`AntennaSelection` — Antenna selection scheme

`'Off'` (default) | `'Tx'` | `'Rx'` | `'Tx and Rx'`

Antenna selection scheme, specified as `'Off'`, `'Tx'`, `'Rx'`, or `'Tx and Rx'`.

`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. For more information, see Antenna Selection on page 3-1017.

Data Types: `char`

`NormalizeChannelOutputs` — Normalize channel outputs

`true` (default) | `false`

Normalize channel outputs, specified as `true` or `false`.

- When you set this property to `true`, channel outputs are normalized by the number of receive antennas.
- When you set this property to `false`, channel outputs are not normalized.

Data Types: `logical`

`FadingTechnique` — Channel model fading technique

`'Filtered Gaussian noise'` (default) | `'Sum of sinusoids'`

Channel model fading technique, specified as `'Filtered Gaussian noise'` or `'Sum of sinusoids'`.

Data Types: `char`

NumSinusoids — Number of sinusoids used

48 (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

This property applies when FadingTechnique is 'Sum of sinusoids'.

Data Types: double

InitialTimeSource — Source to control start time of fading process

'Property' (default) | 'Input port'

Source to control the start time of the fading process, specified as 'Property' or 'Input port'.

- 'Property' -- Use the InitialTime property to set the initial time offset.
- 'Input port' -- Specify the start time of the fading process by using the initialtime input to the object. The input value can change in consecutive calls to the object.

Dependencies

This property applies when FadingTechnique is 'Sum of sinusoids'.

InitialTime — Initial time offset

0 (default) | nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

When InitialTime is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Dependencies

This property applies when the FadingTechnique property is set to 'Sum of sinusoids' and the InitialTimeSource property is set to 'Property'.

Data Types: double

RandomStream — Source of random number stream

'Global stream' (default) | 'mt19937ar with seed'

Source of the random number stream, specified as `'Global stream'` or `'mt19937ar with seed'`.

- `'Global stream'` -- The current global random number stream is used for normally distributed random number generation. In this case, the `reset` object function resets the filters only.
- `'mt19937ar with seed'` -- The mt19937ar algorithm is used for normally distributed random number generation. In this case, the `reset` object function resets the filters and also reinitializes the random number stream to the value of the Seed property.

Data Types: `char`

Seed — Initial seed of mt19937ar random number stream

`73` (default) | nonnegative integer

Initial seed of the mt19937ar random number stream, specified as a nonnegative integer. When the `reset` object function is called, the mt19937ar random number stream is reinitialized to the `Seed` value.

Dependencies

This property applies when you set the RandomStream property to `'mt19937ar with seed'`.

Data Types: `double`

`PathGainsOutputPort` — Option to output path gains

`false` (default) | `true`

Option to output path gains, specified as `false` or `true`. Set this property to `true` to output the channel path gains of the underlying fading process.

Data Types: `logical`

`Visualization` — Channel visualization

`'Off'` (default) | `'Impulse response'` | `'Frequency response'` | `'Impulse and frequency responses'` | `'Doppler spectrum'`

Channel visualization preference, specified as `'Off'`, `'Impulse response'`, `'Frequency response'`, `'Impulse and frequency responses'`, or `'Doppler spectrum'`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

Dependencies

Visualization applies only when the FadingTechnique property is set to `'Filtered Gaussian noise'`.

`AntennaPairsToDisplay` — Transmit-receive antenna pair to display

`[1 1]` (default) | row vector

Transmit-receive antenna pair to display, specified as a 1-by-2 vector, where the first element corresponds to the desired transmit antenna and the second element corresponds to the desired receive antenna. At this time, only a single pair can be displayed.

Dependencies

This property applies when Visualization is not `Off`.

`PathsForDopplerDisplay` — Path for which the Doppler spectrum is displayed

`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_{P}, where N_{P} equals the value of the PathDelays property.

Dependencies

This property applies when Visualization is set to `'Doppler spectrum'`.

`SamplesToDisplay` — Percentage of samples to display

`25%` (default) | `10%` | `50%` | `100%`

Percentage of samples to display, specified as `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This property applies when Visualization is `'Impulse response'`, `'Frequency response'`, or `'Impulse and frequency responses'`.

Usage

Syntax

```
outsignal = mimochan(insignal)
outsignal = mimochan(insignal,seltx)
outsignal = mimochan(insignal,selrx)
outsignal = mimochan(insignal,seltx,selrx)
outsignal = mimochan( ___ ,initialtime)
[outsignal,pathgains] = mimochan( ___ )
```

Description

`outsignal = mimochan(insignal)` filters the input signal through the MIMO fading channel specified by `mimochan` and returns the result in `outsignal`.

`outsignal = mimochan(insignal,seltx)` turns on the transmit antennas selected by `seltx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Tx'`.

For example, to select the first and third transmit antenna index as active:

```
mimochan = comm.MIMOChannel('AntennaSelection','Tx');
seltx = [1 0 1];
...
outsignal = mimochan(insignal,seltx);
```

`outsignal = mimochan(insignal,selrx)` turns on receive antennas, selected by `selrx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Rx'`.

For example, to select the second receive antenna index as active:

```
mimochan = comm.MIMOChannel('AntennaSelection','Rx');
selrx = [0 1];
...
outsignal = mimochan(insignal,selrx);
```

`outsignal = mimochan(insignal,seltx,selrx)` turns on transmit and receive antennas, selected by `seltx` and `selrx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Tx and Rx'`.

For example:

```
mimochan = comm.MIMOChannel('AntennaSelection','Tx and Rx');
seltx = [1 1];
selrx = [0 1];
...
outsignal = mimochan(insignal,selrx);
```

`outsignal = mimochan(___,initialtime)` specifies a start time for the fading process.

This syntax applies when you set the FadingTechnique property of the object to `'Sum of sinusoids'` and the InitialTimeSource property of the object to `'Input port'`. The syntax supports input options from prior syntaxes.

`[outsignal,pathgains] = mimochan(___)` also returns the MIMO channel path gains for antenna selection schemes. The syntax supports input options from prior syntaxes.

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`insignal` — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, an N_S element column vector, an N_S-by-N_T matrix, or an N_S-by-N_{ST} matrix.

- N_S is the number of samples.

- N_T is the number of transmit antennas. N_T is determined by the TransmitCorrelationMatrix or NumTransmitAntennas property values of the object.
- N_{ST} is the number of selected transmit antennas, as determined by the number of elements set to `1` in the vector provided to the seltx input.

The number of transmit antennas is determined by the TransmitCorrelationMatrix or NumTransmitAntennas property values of the object.

Data Types: `double` | `single`
Complex Number Support: Yes

`seltx` — Select active transmit antennas
binary vector

Select active transmit antennas, specified as a 1-by-N_T binary vector. N_T represents the number of transmit antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Data Types: `double`

`selrx` — Select active receive antennas
binary vector

Select active receive antennas, specified as a 1-by-N_R binary vector. N_R represents the number of receive antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Data Types: `double`

`initialtime` — Initial time offset
`0` (default) | nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

The initial time offset must be greater than the last frame end time. When `initialtime` is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Data Types: `double`

Output Arguments

`outsignal` — Output signal
matrix

Output data signal, returned as an N_S-by-N_R or N_S-by-N_{SR} matrix.

- N_S is the number of samples.
- N_R is the number of receive antennas. N_R is determined by the ReceiveCorrelationMatrix or NumReceiveAntennas property values of the object.
- N_{SR} is the number of selected receive antennas, as determined by the number of elements set to `1` in the vector provided to the selrx input.

`pathgains` — Output path gains
4-D array

Output path gains, returned as an N_S-by-N_P-by-N_T-by-N_R array with `NaN` values for the unselected transmit-receive antenna pairs.

- N_S is the number of samples.
- N_P equals the value of the PathDelays property.
- N_T is the number of transmit antennas.
- N_R is the number of receive antennas.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to comm.MIMOChannel

info	Characteristic information about the fading channel object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Note

- If you set the RandomStream property of the object to 'Global stream', the reset object function resets the filters only.
- If you set RandomStream to 'mt19937ar with seed', the reset object function resets the filters and also reinitializes the random number stream to the value of the Seed property.

Examples

Pass QPSK Data Through 4-by-2 MIMO Channel

Create a 4-by-2 MIMO channel by using the MIMO channel System object. Pass modulated and spatially encoded data through the channel.

Generate QPSK-modulated data.

```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Create an orthogonal space-time block encoder to encode the modulated data into four spatially separated streams. Encode the data.

```
ostbc = comm.OSTBCEncoder('NumTransmitAntennas',4,'SymbolRate',1/2);
txSig = ostbc(modData);
```

Create a MIMO channel object, using name-value pairs to set the properties. The channel consists of two paths with a maximum Doppler shift of 5 Hz. Set the SpatialCorrelationSpecification property to 'None', which requires that you specify the number of transmit and receive antennas. Set the number of transmit antennas to 4 and the number of receive antennas to 2.

```
mimochannel = comm.MIMOChannel(...
    'SampleRate',1000, ...
    'PathDelays',[0 2e-3], ...
    'AveragePathGains',[0 -5], ...
    'MaximumDopplerShift',5, ...
    'SpatialCorrelationSpecification','None', ...
    'NumTransmitAntennas',4, ...
    'NumReceiveAntennas',2);
```

Pass the modulated and encoded data through the MIMO channel.

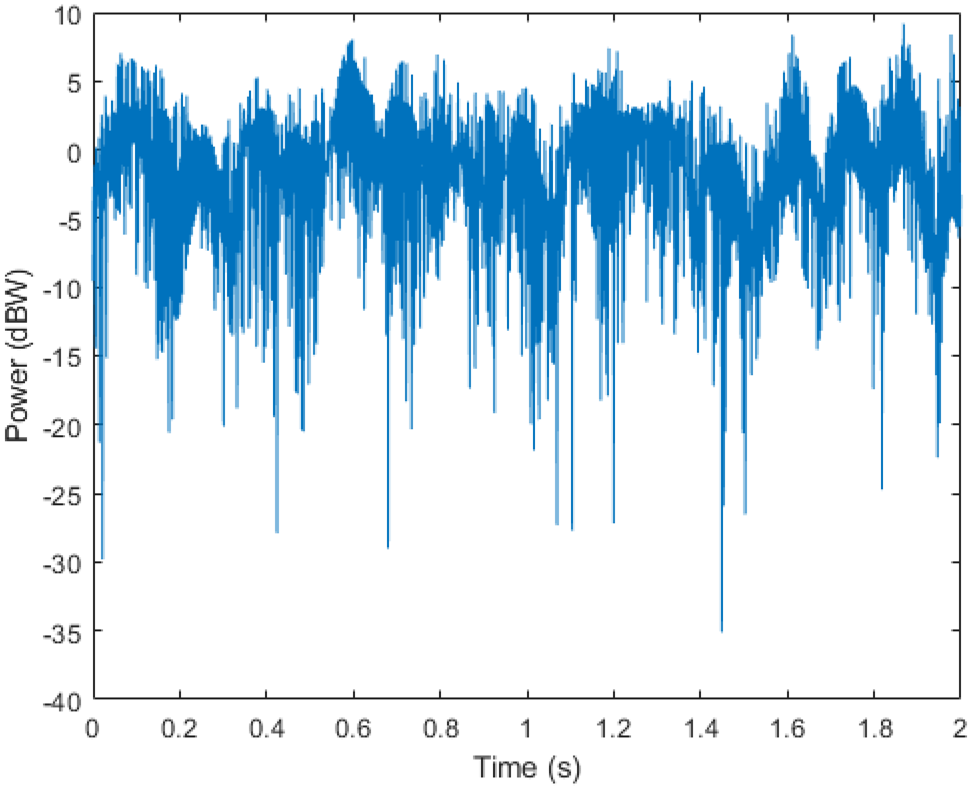
```
rxSig = mimochannel(txSig);
```

Create a time vector, `t`, to use for plotting the power of the received signal.

```
ts = 1/mimochannel.SampleRate;
t = (0:ts:(size(txSig,1)-1)*ts)';
```

Calculate and plot the power of the signal received by antenna 1.

```
pwrdB = 20*log10(abs(rxSig(:,1)));

plot(t,pwrdB)
xlabel('Time (s)')
ylabel('Power (dBW)')
```



Examine Spatial Correlation Characteristics of 2-by-2 Rayleigh Fading Channel

Without specifying antenna selection, filter PSK-modulated data through a 2-by-2 Rayleigh fading channel and examine the spatial correlation characteristics of the channel realization. Use the `release` object function to unlock the object to set the `AntennaSelection` property to `'Tx and Rx'` and then confirm the unselected transmit-receive antenna pairs.

Examine Spatial Correlation Characteristics Without Specifying Antenna Selection

Create a PSK modulator System object™ to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
modData = pskModulator(randi([0 pskModulator.ModulationOrder-1],1e5,1));
```

Split the modulated data into two spatial streams.

```
channelInput = reshape(modData,[2 5e4]).';
```

Create a 2-by-2 MIMO channel System object with two discrete paths. Each path has different transmit and receive correlation matrices, specified by the `TransmitCorrelationMatrix` and `ReceiveCorrelationMatrix` properties.

```
mimoChan = comm.MIMOChannel('SampleRate',1000, 'PathDelays',[0 1e-3], ...
    'AveragePathGains',[3 5], 'NormalizePathGains',false, '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, 'PathGainsOutputPort',true);
```

Filter the modulated data using the MIMO channel object.

```
[~,pathGains] = mimoChan(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 by using the `corrcoef` function.

```
disp('Tx spatial correlation, first path, first Rx:');
```

```
Tx spatial correlation, first path, first Rx:
```

```
disp(corrcoef(squeeze(pathGains(:,1,:,1))));
```

```
   1.0000 + 0.0000i   0.0357 - 0.0253i
   0.0357 + 0.0253i   1.0000 + 0.0000i
```

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:');
```

```
Tx spatial correlation, second path, second Rx:
```

```
disp(corrcoef(squeeze(pathGains(:,2,:,2))));
```

```
   1.0000 + 0.0000i   0.0863 + 0.0009i
   0.0863 - 0.0009i   1.0000 + 0.0000i
```

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:');
```

```
Rx spatial correlation, first path, second Tx:
```

```
disp(corrcoef(squeeze(pathGains(:,1,2,:))));
```

```
   1.0000 + 0.0000i   0.2236 + 0.0550i
   0.2236 - 0.0550i   1.0000 + 0.0000i
```

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:');
```

```
Rx spatial correlation, second path, first Tx:
```

```
disp(corrcoef(squeeze(pathGains(:,2,1,:))));
```

```
   1.0000 + 0.0000i  -0.0088 - 0.0489i
  -0.0088 + 0.0489i   1.0000 + 0.0000i
```

Examine Spatial Correlation Characteristics Specifying Antenna Selection

Enable transmit and receive antenna selection for the mimoChan object. The input frame size is shortened to 100.

```
release(mimoChan);
mimoChan.AntennaSelection = 'Tx and Rx';
modData = pskModulator(randi([0 pskModulator.ModulationOrder-1],100,1));
```

Select the first transmit antenna and second receive antenna.

```
[channelOutput,pathGains] = mimoChan(modData,[1 0],[0 1]);
```

Confirm that the path gains that 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:');
```

```
Return 1 if the path gains for the second transmit antenna are NaN:
```

```
disp(isequal(isnan(squeeze(pathGains(:,:,2,:))), ones(100,2,2)));
```

```
   1
```

```
disp('Return 1 if the path gains for the first receive antenna are NaN:');
```

```
Return 1 if the path gains for the first receive antenna are NaN:
```

```
disp(isequal(isnan(squeeze(pathGains(:,:,:,1))), ones(100,2,2)));
```

```
   1
```

Display Impulse and Frequency Responses of Frequency Selective Channel

Create a frequency selective MIMO channel and display its impulse and frequency responses.

Set the sample rate to 10 MHz and specify path delays and gains using the extended vehicular A (EVA) channel parameters. Set the maximum Doppler shift to 70 Hz.

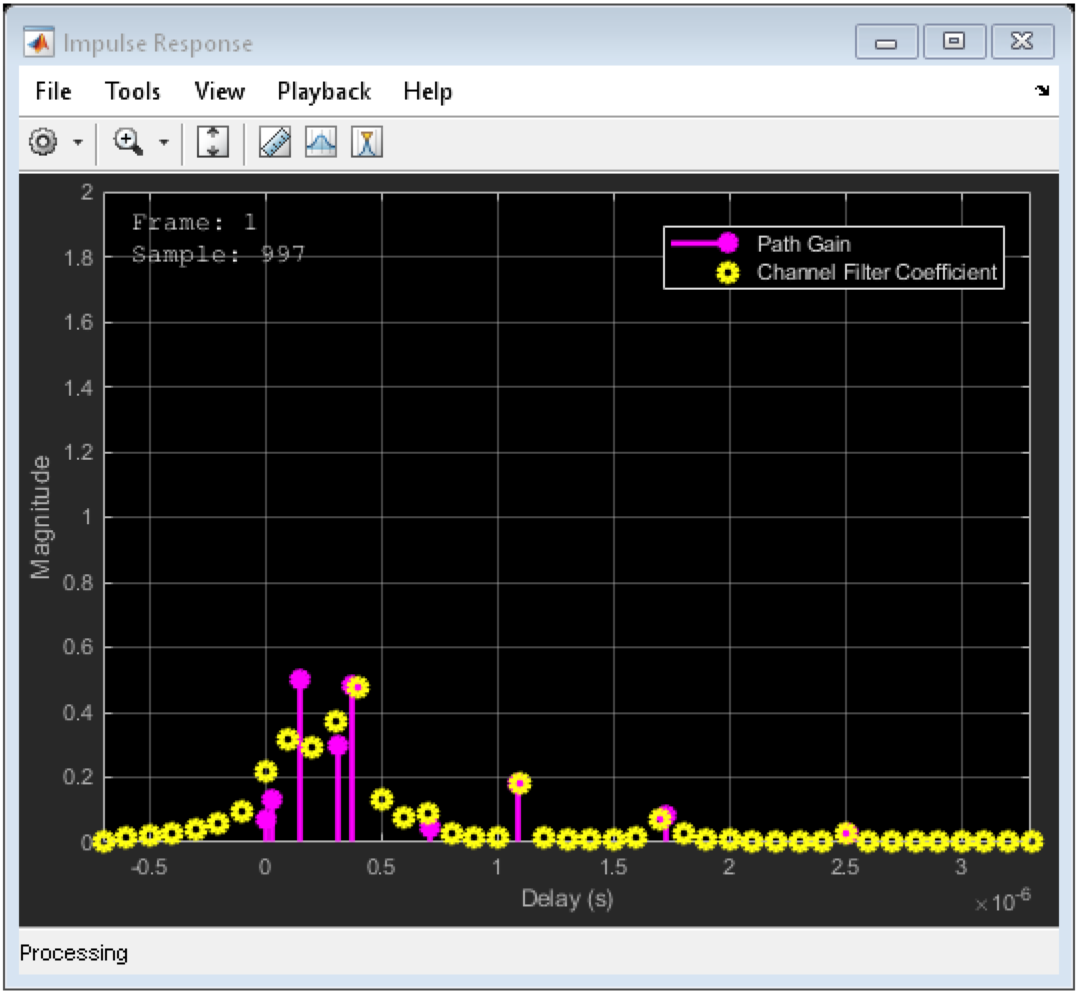
```
fs = 10e6;                                                % Hz
pathDelays = [0 30 150 310 370 710 1090 1730 2510]*1e-9;    % sec
avgPathGains = [0 -1.5 -1.4 -3.6 -0.6 -9.1 -7 -12 -16.9];   % dB
fD = 70;                                                  % Hz
```

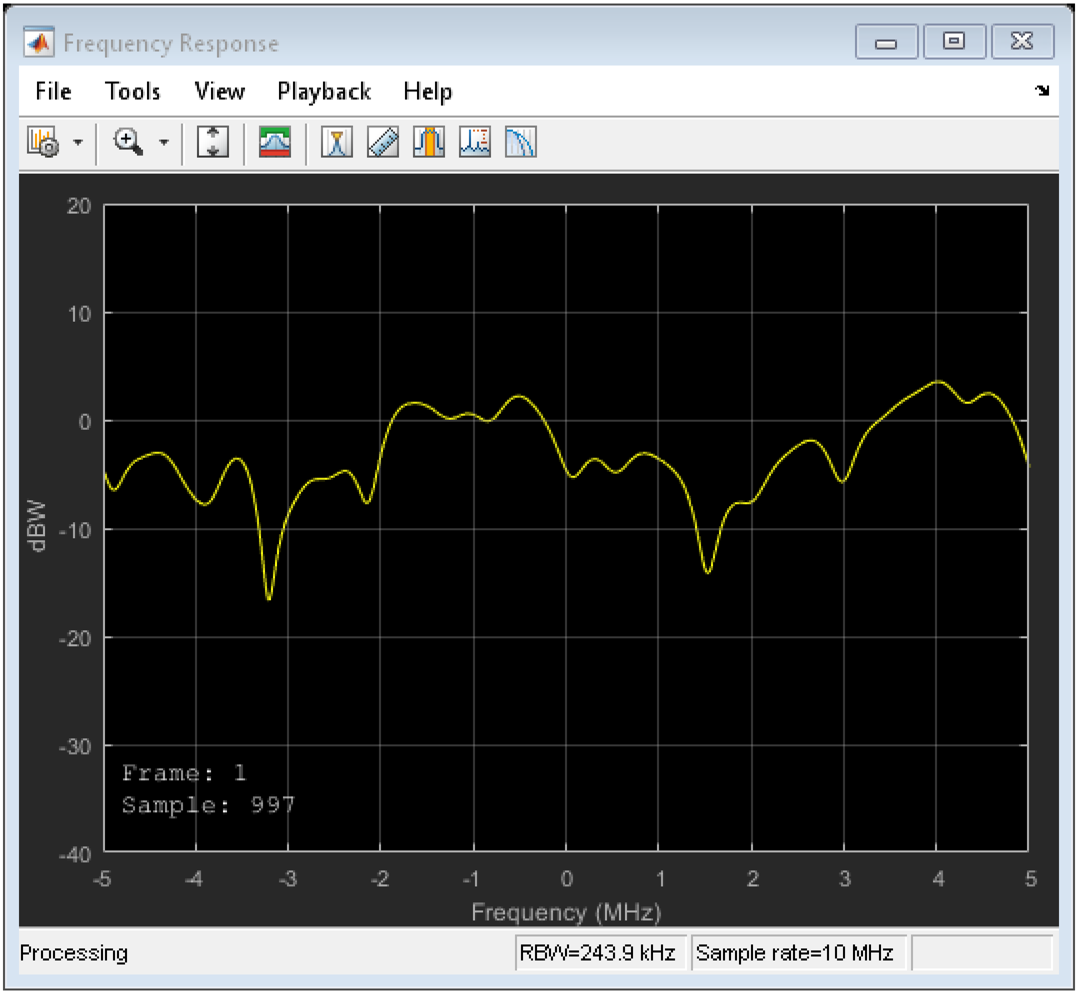
Create a 2x2 MIMO channel System object with the previously defined parameters and set the `Visualization` property to `Impulse and frequency responses` using name-value pairs. By default, the antenna pair corresponding to transmit antenna 1 and receive antenna 1 will be displayed.

```
mimoChan = comm.MIMOChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the MIMO channel. The impulse response plot allows you to easily identify the individual paths and their corresponding filter coefficients. The frequency selective nature of the EVA channel is shown by the frequency response plot.

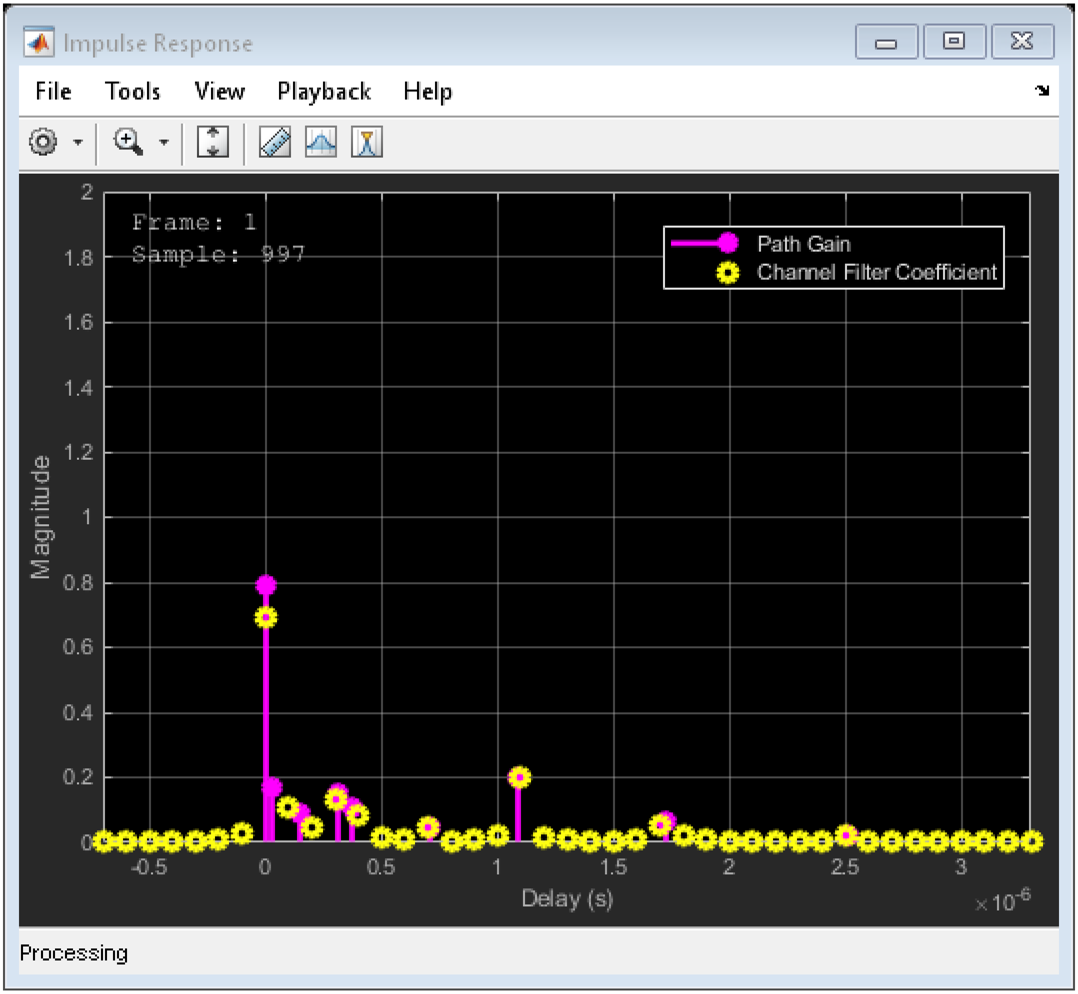
```
x = randi([0 1],1000,2);
y = mimoChan(x);
```

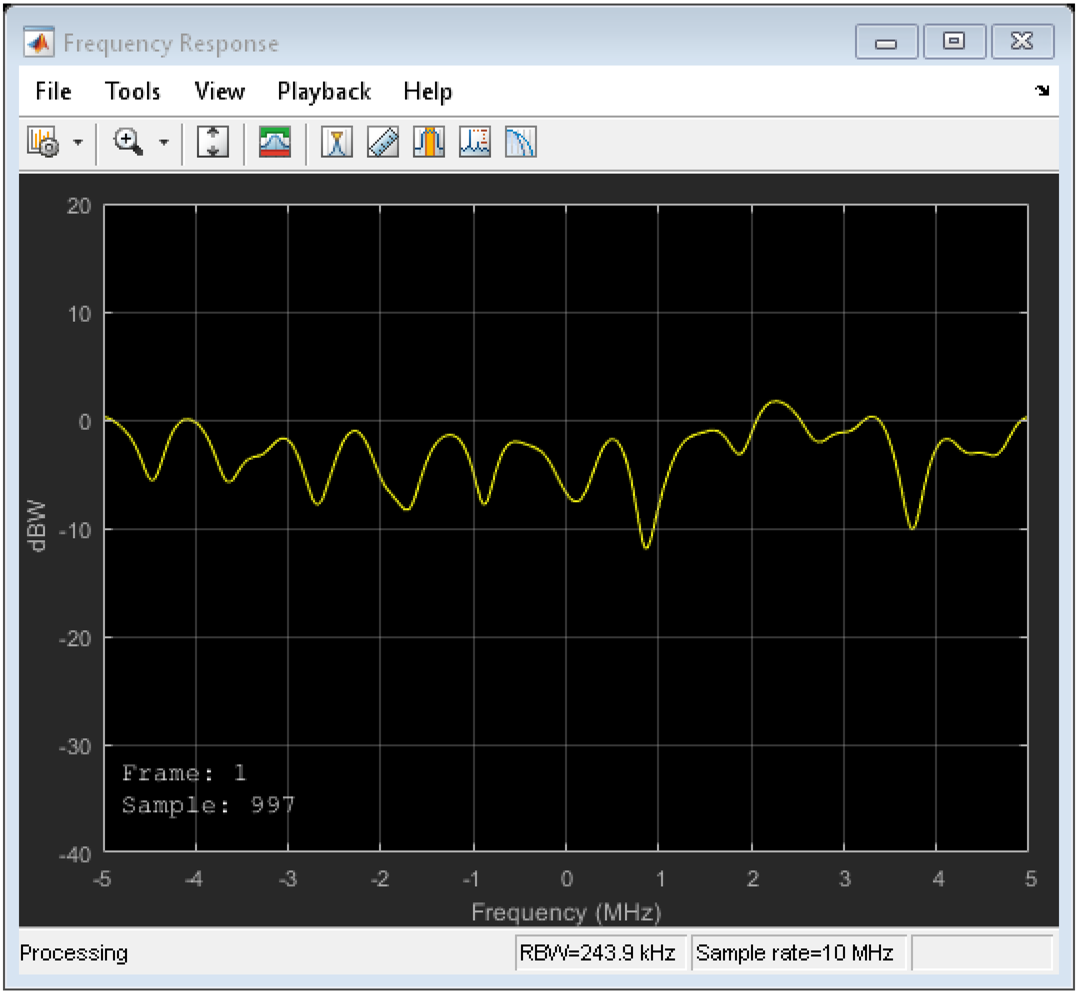

Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 997
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-6
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
-0.5
0
0.5
1
1.5
2
2.5
3
Processing



Release `mimoChan` and set the `AntennaPairsToDisplay` property to [2 1] to view the antenna pair corresponding to transmit antenna 2 and receive antenna 1. It is necessary to release the object as the property is non-tunable.

```
release(mimoChan)
mimoChan.AntennaPairsToDisplay = [2 1];
y = mimoChan(x);
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 997
Path Gain
Channel Filter Coefficient
Magnitude
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
-0.5
0
0.5
1
1.5
2
2.5
3
Delay (s)
$\times 10^{-6}$
Processing



```
Frame: 1
Sample: 997
```

Processing

RBW=243.9 kHz

Sample rate=10 MHz

Display Doppler for 2x2 MIMO Channel

Create and visualize the Doppler spectra of a MIMO channel having two paths.

Construct a cell array of Doppler structures to be used in creating the channel. The Doppler spectrum of the first path is set to have a bell shape while the second path is set to be flat.

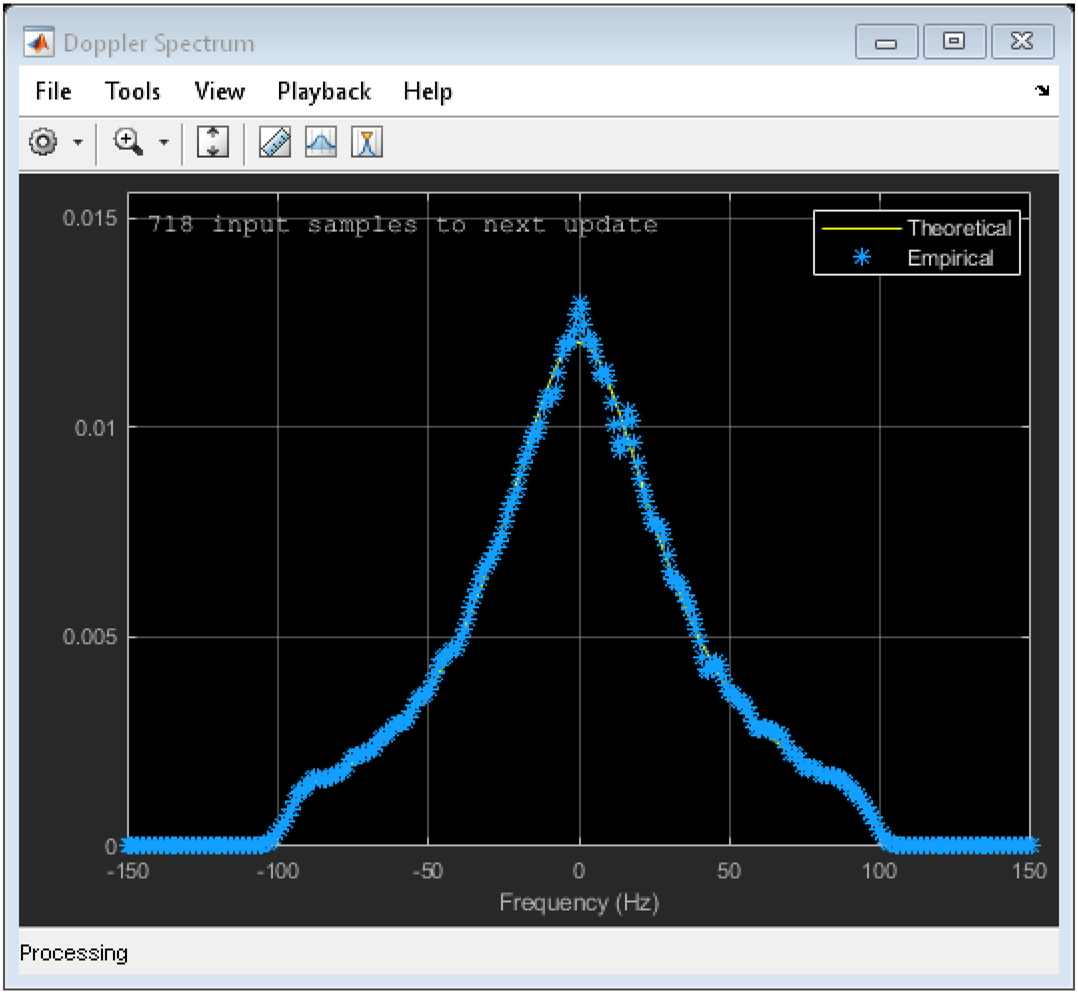
```
dp{1} = doppler('Bell');
dp{2} = doppler('Flat');
```

Create a default 2x2 MIMO channel with two paths and a 100 Hz maximum Doppler shift using name-value pairs. Set the `Visualization` property to `Doppler spectrum` and set `PathsForDopplerDisplay` to 1. The Doppler spectrum of the first path will be displayed.

```
mimoChan = comm.MIMOChannel('SampleRate',1000, ...
    'PathDelays',[0 0.002], ...
    'AveragePathGains',[0 -3], ...
    'MaximumDopplerShift',100, ...
    'DopplerSpectrum',dp, ...
    'Visualization','Doppler spectrum', ...
    'PathsForDopplerDisplay',1);
```

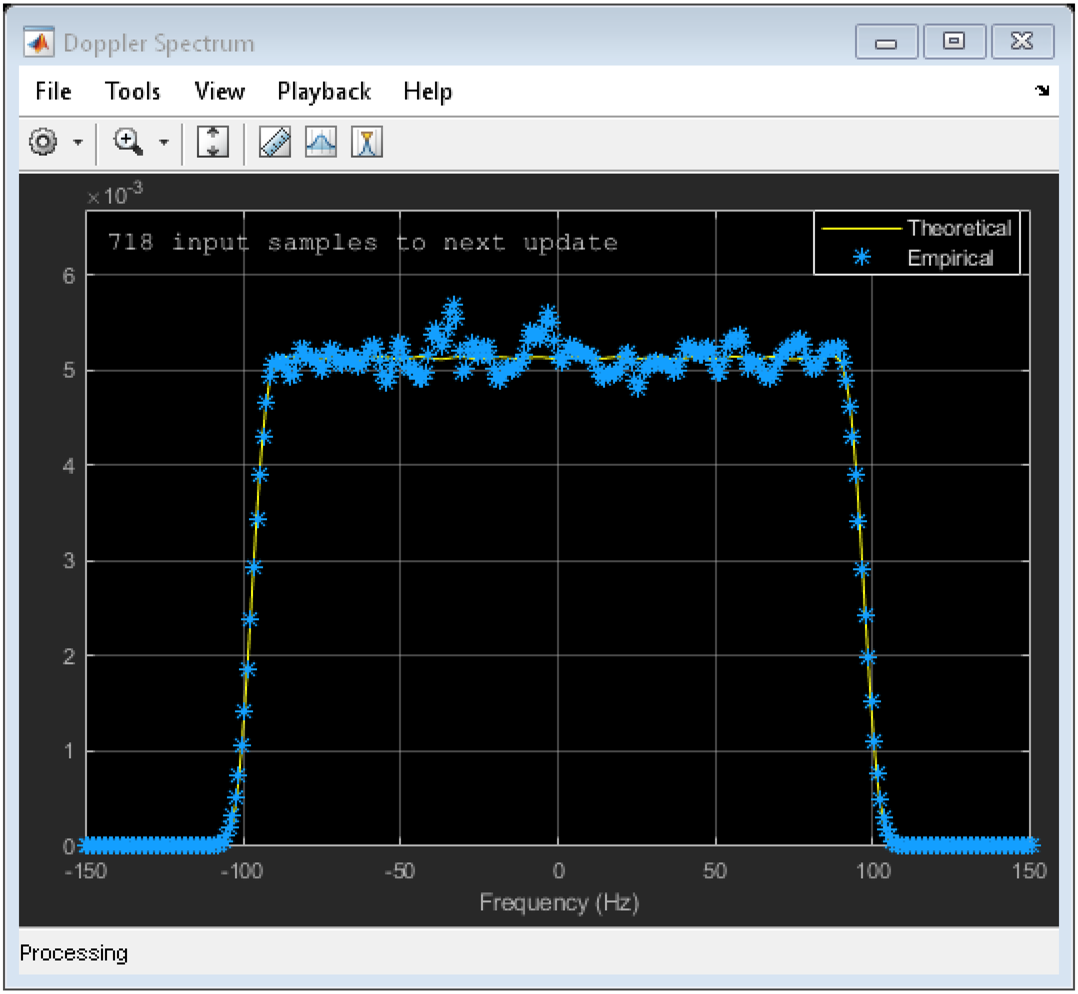
Pass random data through the MIMO channel to generate the Doppler spectrum of the first path. Since the Doppler spectrum plot only updates when its buffer is filled, the `mimoChan` function is invoked multiple times to improve the accuracy of the estimate. Observe that the spectrum has a bell shape and that its minimum and maximum frequencies fall within the limits set by `MaximumDopplerShift`.

```
for k = 1:25
    x = randi([0 1],10000,2);
    y = mimoChan(x);
end
```



Release `mimoChan` and set the `PathsForDopplerDisplay` property to 2. It is necessary to release the object as the property is non-tunable. Call the function multiple times to display the Doppler spectrum of the second path. Observe that the spectrum is flat.

```
release(mimoChan)
mimoChan.PathsForDopplerDisplay = 2;
for k = 1:25
    x = randi([0 1],10000,2);
    y = mimoChan(x);
end
```



Model MIMO Channel Using Sum-of-Sinusoids Technique

Create a MIMO channel object and pass data through it using the sum-of-sinusoids technique. The example demonstrates how the channel state is maintained in cases in which data is discontinuously transmitted.

Define the overall simulation time and three time segments for which data will be transmitted. In this case, the channel is simulated for 1 s with a 1000 Hz sampling rate. One 1000-sample, continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at time 0.1 s, 0.4 s, and 0.7 s.

```
t0 = 0:0.001:0.999;   % Transmission 0
t1 = 0.1:0.001:0.199; % Transmission 1
t2 = 0.4:0.001:0.499; % Transmission 2
t3 = 0.7:0.001:0.799; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],1000,2);  % 1000 samples
d1 = randi([0 1],100,2);   % 100 samples
d2 = randi([0 1],100,2);   % 100 samples
d3 = randi([0 1],100,2);   % 100 samples
```

Create a flat fading 2x2 MIMO channel System object with the `Sum of sinusoids` fading technique. So that results can be repeated, specify a seed using a name-value pair. As the `InitialTime` property is not specified, the fading channel will be simulated from time 0. Enable the path gains output port.

```
mimoChan1 = comm.MIMOChannel('SampleRate',1000, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the MIMO channel System object. Set the `InitialTimeSource` property to `Input port` so that the fading channel offset time can be specified as an input argument to the `mimoChan` function.

```
mimoChan2 = clone(mimoChan1);
mimoChan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first channel object, mimoChan1. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = mimoChan1(d0);
```

Pass random data through the second channel object, mimoChan2, where the initial time offsets are provided as input arguments.

```
[~,pg1] = mimoChan2(d1,0.1);
[~,pg2] = mimoChan2(d2,0.4);
[~,pg3] = mimoChan2(d3,0.7);
```

Compare the number of samples processed by the two channels using the info method. You can see that 1000 samples were processed by mimoChan1 while only 300 were processed by mimoChan2.

```
G = info(mimoChan1);
H = info(mimoChan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

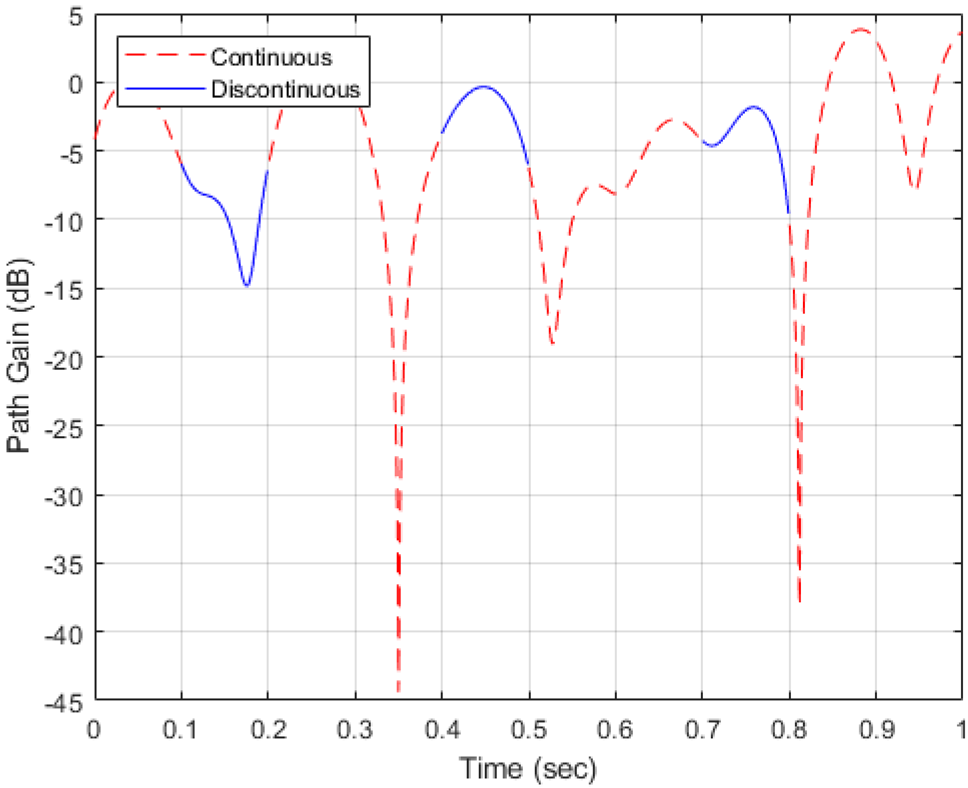
```
ans = 1×2

        1000         300
```

Convert the path gains into decibels for the path corresponding to the first transmit and first receive antenna.

```
pathGain0 = 20*log10(abs(pg0(:,1,1,1)));
pathGain1 = 20*log10(abs(pg1(:,1,1,1)));
pathGain2 = 20*log10(abs(pg2(:,1,1,1)));
pathGain3 = 20*log10(abs(pg3(:,1,1,1)));
```

Plot the path gains for the continuous and discontinuous cases. Observe that the gains for the three segments perfectly match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is ideally suited to the simulation of packetized data as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
```



Calculate Execution Time Advantage Using Sum of Sinusoids

Demonstrate the advantage of using the sum of sinusoids fading technique when simulating a channel with burst data.

Set the simulation parameters such that the sampling rate is 100 kHz, the total simulation time is 100 seconds, and the duty cycle for the burst data is 25%.

```
fs = 1e5;            % Hz
tsim = 100;          % seconds
dutyCycle = 0.25;
```

Create a flat fading 2x2 MIMO channel object using the default `Filtered Gaussian noise` technique.

```
fgn = comm.MIMOChannel('SampleRate',fs);
```

Create a similar MIMO channel object using the `Sum of sinusoids` technique where the fading process start times are given as an input argument.

```
sos = comm.MIMOChannel('SampleRate',fs, ...
    'FadingTechnique','Sum of sinusoids', ...
    'NumSinusoids',48, ...
    'InitialTimeSource','Input port');
```

Run a continuous sequence of random bits through the filtered Gaussian noise MIMO channel object. Use the tic/toc stopwatch timer functions to measure the execution time of the function call.

```
tic
y = fgn(randi([0 1],fs*tsim,2));
tFGN = toc;
```

To transmit a data burst each second, pass random bits through the sum of sinusoids MIMO channel object by calling the `sos` function inside of a for loop. Use the tic/toc stopwatch timer to measure the execution time.

```
tic
for k = 1:tsim
    z = sos(randi([0 1],fs*dutyCycle,2),0.5+(k-1));
end
tSOS = toc;
```

Compare the ratio of the sum of sinusoids execution time to the filtered Gaussian noise execution time. The ratio is less than one, which indicates that the sum of sinusoids technique is faster.

```
tSOS/tFGN
```

```
ans = 0.2917
```

Algorithms

The fading processing per link is described in Methodology for Simulating Multipath Fading Channels and assumes the same parameters for all (N_T × 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 is:

$$R_H = E[R_t \otimes R_r]$$

- The ⊗ symbol represents the Kronecker product.
- R_t represents the correlation matrix at the transmit side: $R_t = E\left[H^H H\right]$, of size N_T-by-N_T.
- R_r represents the correlation matrix at the receive side: $R_r = E\left[HH^H\right]$, of size N_R-by-N_R.

You can obtain a realization of the MIMO channel matrix as:

$$H = R_r^{\frac{1}{2}} A R_t^{\frac{1}{2}}$$

A is an N_R-by-N_T matrix of independent identically distributed complex Gaussian variables with zero mean and unit variance.

Cutoff Frequency Factor

The following information explains how the cutoff frequency factor, f_c, is determined for different Doppler spectrum types:

- For any Doppler spectrum type other than Gaussian and BiGaussian, f_c equals `1`.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation•sqrt(2•log(2))`.
- For a `doppler('BiGaussian')` spectrum type:

  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)•sqrt(2•log(2))`.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)•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:

- All random path gains are always generated and keep evolving for each link, whether or not a given link is selected. The path gain values output for the non-selected links are populated with `NaN`.
- The spatial correlation only applies to the selected transmit and/or receive antennas, and the correlation coefficients are the corresponding entries in the transmit, receive, or combined correlation matrices. In other words, the spatial correlation matrix for the selected transmit or receive antennas is a submatrix of the transmit, receive, or combined spatial correlation matrix property value.
- The input filtering through the path gains is applied to the selected links only.
- Whenever a link associated with a specific transmitter transitions from a selected state to a non-selected state, its channel filter is reset. For example, if the antenna selection property is set to `Tx` and the selected transmit antenna is changed from `2` to `1`, the channel filter corresponding to antenna `2` will be reset.
- Channel output normalization happens over the number of selected receive antennas.

References

[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.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

System Objects

`comm.AWGNChannel` | `comm.LTEMIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Blocks

MIMO Channel

Topics

Channel Visualization

Introduced in R2012a

info

Package: comm

Characteristic information about the fading channel object

Syntax

```
infostruct = info(obj)
```

Description

`infostruct = info(obj)` returns a structure containing characteristic information for the System object.

Examples

Get comm.MIMOChannel Info

Use the `info` object function to get information from a `comm.MIMOChannel` object.

Create a MIMO channel object and some data to pass through the channel.

```
mimo = comm.MIMOChannel('SampleRate',1000);
data = randi([0 1],600,2);
```

Check the MIMO channel object information

```
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 0
```

Pass data through the channel and check the object information again.

```
mimo(data);
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 600
```

Release the object so you can update attributes. Add a two second path delay. Recheck the object information.

```
release(mimo)
mimo.PathDelays = 2;
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 2000
    ChannelFilterCoefficients: [1x2001 double]
          NumSamplesProcessed: 0
```

Model MIMO Channel Using Sum-of-Sinusoids Technique

Create a MIMO channel object and pass data through it using the sum-of-sinusoids technique. The example demonstrates how the channel state is maintained in cases in which data is discontinuously transmitted.

Define the overall simulation time and three time segments for which data will be transmitted. In this case, the channel is simulated for 1 s with a 1000 Hz sampling rate. One 1000-sample, continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at time 0.1 s, 0.4 s, and 0.7 s.

```
t0 = 0:0.001:0.999;   % Transmission 0
t1 = 0.1:0.001:0.199; % Transmission 1
t2 = 0.4:0.001:0.499; % Transmission 2
t3 = 0.7:0.001:0.799; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],1000,2);  % 1000 samples
d1 = randi([0 1],100,2);   % 100 samples
d2 = randi([0 1],100,2);   % 100 samples
d3 = randi([0 1],100,2);   % 100 samples
```

Create a flat fading 2x2 MIMO channel System object with the `Sum of sinusoids` fading technique. So that results can be repeated, specify a seed using a name-value pair. As the `InitialTime` property is not specified, the fading channel will be simulated from time 0. Enable the path gains output port.

```
mimoChan1 = comm.MIMOChannel('SampleRate',1000, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the MIMO channel System object. Set the `InitialTimeSource` property to `Input port` so that the fading channel offset time can be specified as an input argument to the `mimoChan` function.

```
mimoChan2 = clone(mimoChan1);
mimoChan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first channel object, `mimoChan1`. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = mimoChan1(d0);
```

Pass random data through the second channel object, `mimoChan2`, where the initial time offsets are provided as input arguments.

```
[~,pg1] = mimoChan2(d1,0.1);
[~,pg2] = mimoChan2(d2,0.4);
[~,pg3] = mimoChan2(d3,0.7);
```

Compare the number of samples processed by the two channels using the `info` method. You can see that 1000 samples were processed by `mimoChan1` while only 300 were processed by `mimoChan2`.

```
G = info(mimoChan1);
H = info(mimoChan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

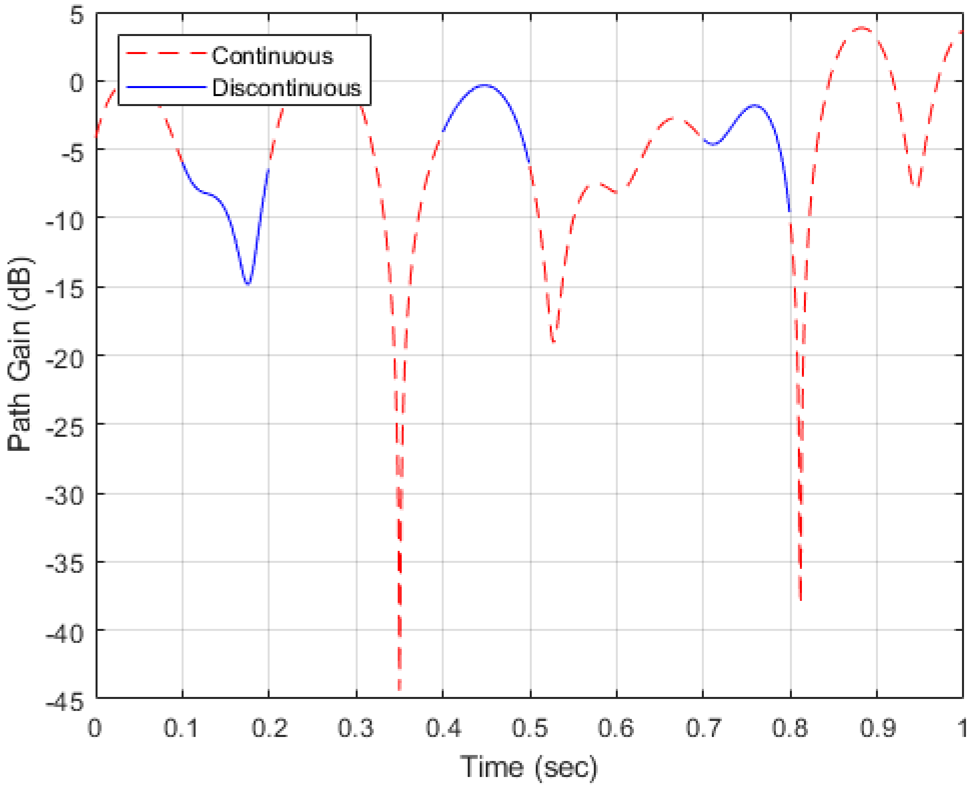
```
ans = 1×2

        1000         300
```

Convert the path gains into decibels for the path corresponding to the first transmit and first receive antenna.

```
pathGain0 = 20*log10(abs(pg0(:,1,1,1)));
pathGain1 = 20*log10(abs(pg1(:,1,1,1)));
pathGain2 = 20*log10(abs(pg2(:,1,1,1)));
pathGain3 = 20*log10(abs(pg3(:,1,1,1)));
```

Plot the path gains for the continuous and discontinuous cases. Observe that the gains for the three segments perfectly match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is ideally suited to the simulation of packetized data as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
```



Input Arguments

obj — System object to get information from

System object

System object to get information from.

Output Arguments

`infostruct` — Structure containing object information

struct

Structure containing these fileds with information about the System object.

`ChannelFilterDelay` — Channel filter delay

positive integer

Channel filter delay in samples, returned as a positive integer.

`ChannelFilterCoefficients` — Channel filter coefficients

matrix

Channel filter coefficients, returned as a matrix. The coefficient matrix is used to convert path gains to channel filter tap gains for each sample and each pair of transmit and receive antennas.

`NumSamplesProcessed` — Number of samples processed by the channel object

positive integer

Number of samples processed by the channel object since the last reset, returned as a positive integer.

`LastFrameTime` — Last frame ending time

positive scalar

Last frame ending time in seconds, returned as a positive scalar. Use this value to confirm the simulation time.

Dependencies

This property applies when FadingTechnique is `'Sum of sinusoids'` and InitialTimeSource is `'Input port'`.

See Also

System Objects

`comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2012a

comm.MLSEEqualizer System object

Package: comm

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.

To equalize a linearly modulated signal and output the maximum likelihood sequence estimate:

1. Define and set up your maximum likelihood sequence estimate equalizer object. See "Construction" on page 3-1027.
2. Call `step` to equalize a linearly modulated signal and output the maximum likelihood sequence estimate according to the properties of `comm.MLSEEqualizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 property. This property applies when you set the ChannelSource on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Truncated` and the `PreambleSource` on page 3-0 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` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Truncated` and the `PostambleSource` on page 3-0 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

reset	Reset states of MLSEE object
step	Equalize using maximum likelihood sequence estimation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

MLSE Equalize QPSK Signal Through Dispersive Channel

This example shows how to use an MLSE equalizer to remove the effects of a frequency-selective channel.

Specify static channel coefficients.

```
chCoeffs = [.986; .845; .237; .12345+.31i];
```

Create an MLSE equalizer object. Create an error rate calculator object.

```
mlse = comm.MLSEEqualizer('TracebackDepth',10,...
    'Channel',chCoeffs,'Constellation',pskmod(0:3,4,pi/4));
errorRate = comm.ErrorRate;
```

The main processing loop includes these steps:

- Data generation
- QPSK modulation

- Channel filtering
- Signal equalization
- QPSK demodulation
- Error computation

```
for n = 1:50
    data= randi([0 3],100,1);
    modSignal = pskmod(data,4,pi/4,'gray');

    % Introduce channel distortion.
    chanOutput = filter(chCoeffs,1,modSignal);

    % Equalize the channel output and demodulate.
    eqSignal = mlse(chanOutput);
    demodData = pskdemod(eqSignal,4,pi/4,'gray');

    % Compute BER.
    errorStats = errorRate(data,demodData);
end
```

Display the bit error rate and the number of errors.

```
ber = errorStats(1)
```

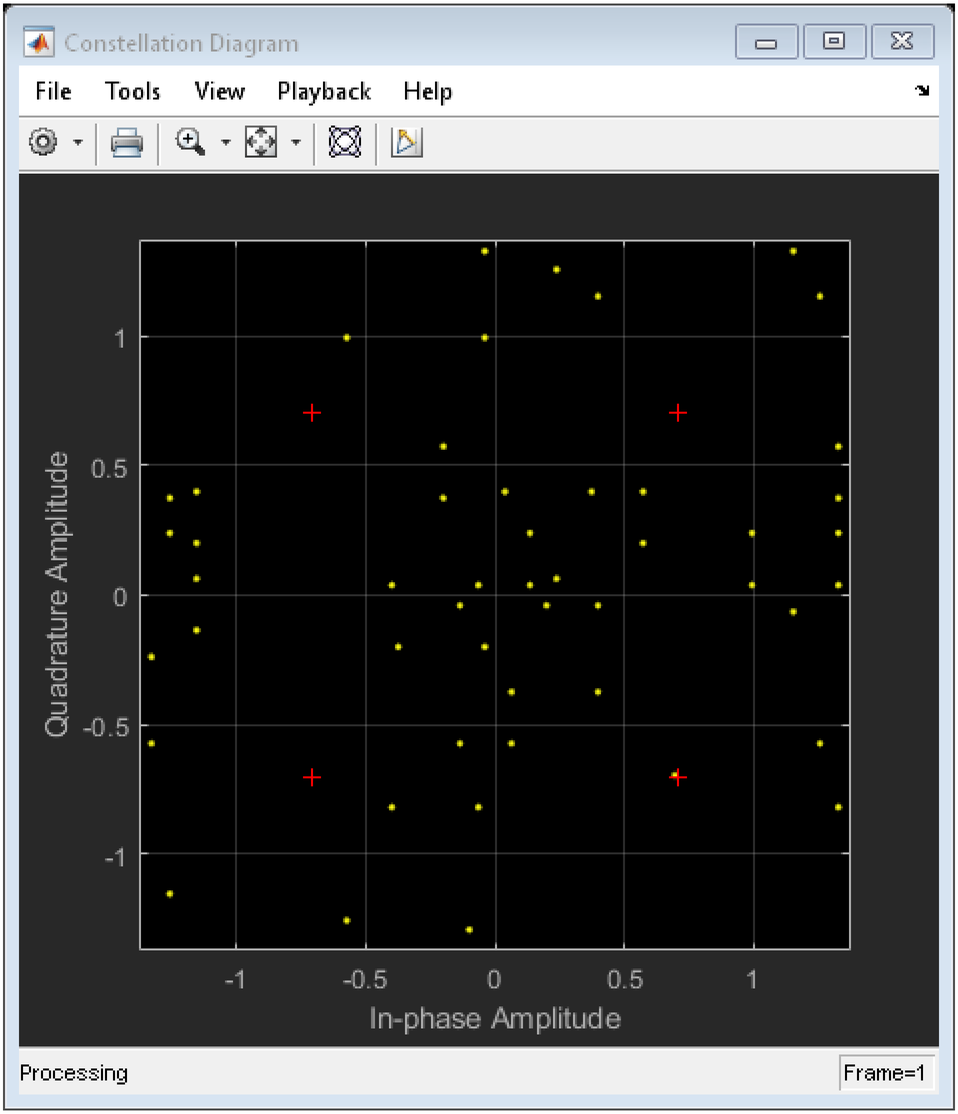
```
ber = 0
```

```
numErrors = errorStats(2)
```

```
numErrors = 0
```

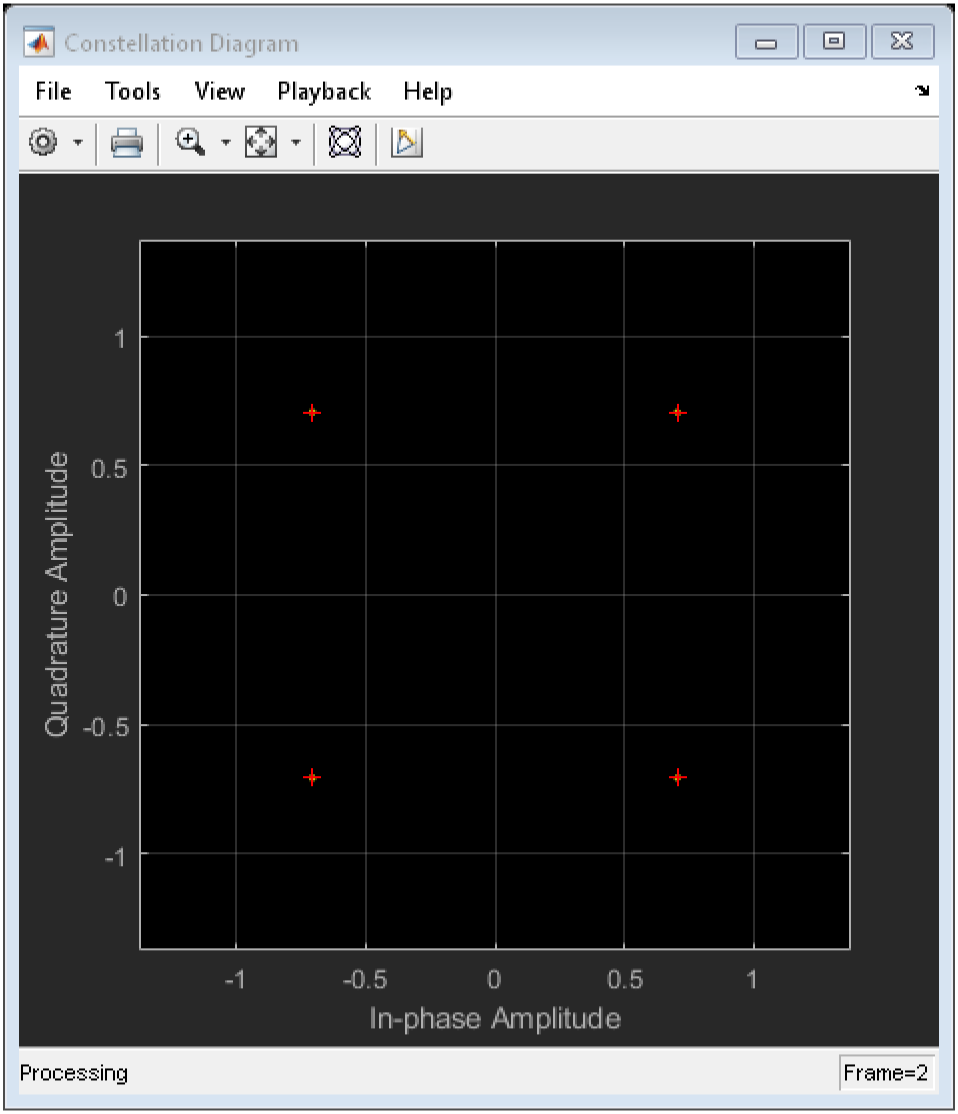
Plot the signal constellation prior to equalization.

```
constDiagram = comm.ConstellationDiagram;
constDiagram(chanOutput)
```



Plot the signal constellation after equalization.

```
constDiagram(eqSignal)
```



The equalized symbols align perfectly with the QPSK reference constellation.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: comm.MLSEEqualizer
Package: comm

Reset states of MLSEE object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MLSEEqualizer` object, `H`.

step

System object: comm.MLSEEqualizer
Package: comm

Equalize using maximum likelihood sequence estimation

Syntax

```
Y = step(H,X)
Y = step(H,X,CHANNEL)
Y = step(H,X,RESET)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MSKDemodulator System object

Package: comm

Demodulate using MSK method and the Viterbi algorithm

Description

The comm.MSKDemodulator object demodulates a signal that was modulated using the minimum shift keying method. The object expects the input signal to be a baseband representation of a coherent modulated signal with no precoding. The initial phase offset property sets the initial phase of the modulated waveform.

To demodulate a signal that was modulated using minimum shift keying:

1. Define and set up your MSK demodulator object. See “Construction” on page 3-1039.
2. Call `step` to demodulate the signal according to the properties of `comm.MSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 . *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` on page 3-0 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 on page 3-0 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

reset	Reset states of the MSK demodulator object
step	Demodulate using MSK method and the Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate an MSK signal with bit inputs and phase offset

```
% Create an MSK modulator, an AWGN channel, and an MSK demodulator.  Use a
% 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 Viterbi algorit
    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))
```

```
Error rate = 0.000000
Number of errors = 0
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

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. For MSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.MSKModulator`

Introduced in R2012a

reset

System object: comm.MSKDemodulator
Package: comm

Reset states of the MSK demodulator object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `MSKDemodulator` object, `H`.

step

System object: comm.MSKDemodulator
Package: comm

Demodulate using MSK method and the Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To modulate a signal using minimum shift keying:

1. Define and set up your MSK modulator object. See "Construction" on page 3-1047.
2. Call step to modulate the signal according to the properties of comm.MSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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 on page 3-0 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

reset	Reset states of the MSK modulator object
step	Modulate using MSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate an MSK signal with bit inputs and phase offset

```
% Create an MSK modulator, an AWGN channel, and an MSK demodulator.  Use a
% 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 Viterbi algorit
    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))
```

```
Error rate = 0.000000
Number of errors = 0
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

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. For MSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.MSKDemodulator`

Introduced in R2012a

reset

System object: comm.MSKModulator
Package: comm

Reset states of the MSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MSKModulator` object, `H`.

step

System object: comm.MSKModulator
Package: comm

Modulate using MSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MSKTimingSynchronizer System object

Package: comm

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.

To recover the symbol timing phase of the input signal:

1. Define and set up your MSK timing synchronizer object. See “Construction” on page 3-1055.
2. Call `step` to recover the symbol timing phase of the input signal according to the properties of `comm.MSKTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 , 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` on page 3-0 property to `false`.

Methods

reset	Reset states of MSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover Timing Phase of MSK Signal

Create MSK modulator, variable fractional delay, and MSK timing synchronizer System objects.

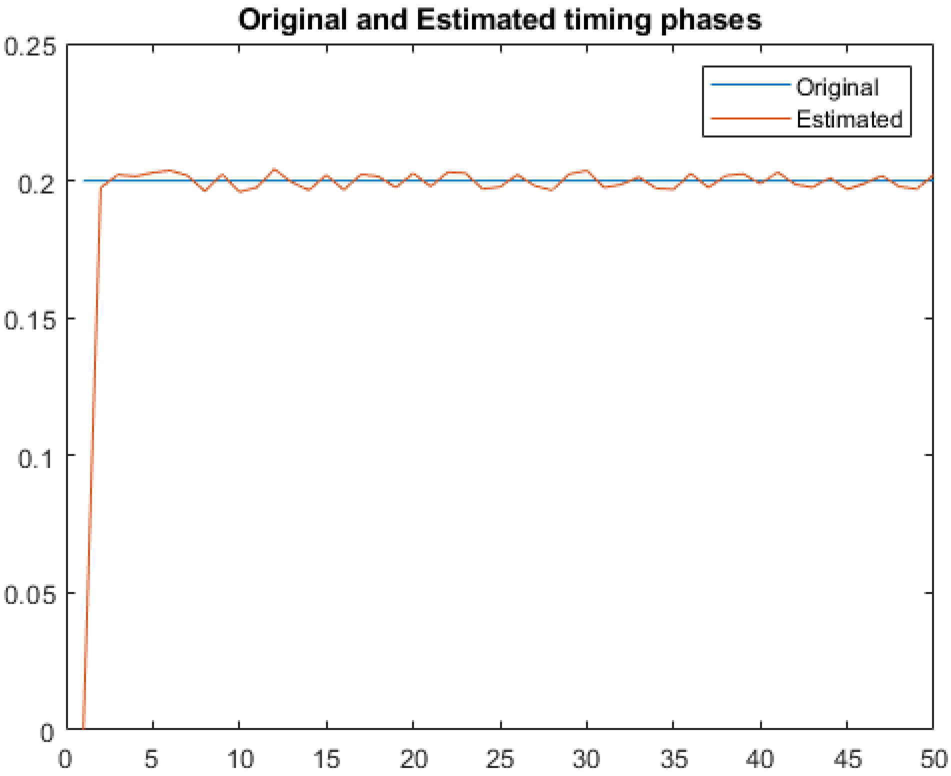
```
mskMod = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',14);
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
mskTimingSync = comm.MSKTimingSynchronizer('SamplesPerSymbol',14,'ErrorUpdateGain', 0.0
```

Main processing loop.

```
phEst = zeros(50,1);
for i = 1:50
    data = randi([0 1],100,1);    % Generate data
    modData = mskMod(data);       % Modulate data

    % Apply timing offset error.
    impairedData = varDelay(modData,timingOffset*14);
    % Perform timing phase recovery.
    [~,phase] = mskTimingSync(impairedData);
    phEst(i) = phase(1)/14;
end
```

Plot the results.

```
plot(1:50,[0.2*ones(50,1) phEst]);
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` on page 3-0 and `ResetCondition` on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: comm.MSKTimingSynchronizer
Package: comm

Reset states of MSK timing phase synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of `MSKTimingSynchronizer` object, `H`.

step

System object: comm.MSKTimingSynchronizer
Package: comm

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MuellerMullerTimingSynchronizer System object

Package: comm

Recover symbol timing phase using Mueller-Muller method

Note `comm.MuellerMullerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

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.

To recover the symbol timing phase of the input signal:

1. Define and set up your Mueller-Muller timing synchronizer object. See “Construction” on page 3-1063.
2. Call `step` to recover the symbol timing phase of the input signal according to the properties of `comm.MuellerMullerTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 , 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` on page 3-0 property to `false`.

Methods

reset	Reset states of Mueller-Muller timing phase synchronizer
step	Recover symbol timing phase using Mueller-Muller method

Common to All System Objects	
`release`	Allow System object property value changes

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;

% Create System objects
    hMod = comm.DPSKModulator(M, 'PhaseRotation', 0);
    hTxFilter = comm.RaisedCosineTransmitFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'OutputSamplesPerSymbol', L);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
        'NoiseMethod', 'Signal to noise ratio (SNR)', ...
        'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = comm.RaisedCosineReceiveFilter(...
        'RolloffFactor', rollOff, ...
        'FilterSpanInSymbols', 2*filtDelay, ...
        'InputSamplesPerSymbol', L, ...
        'DecimationFactor', 1);
    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` on page 3-0 and `ResetCondition` on page 3-0 properties.

See Also

`comm.GMSKTimingSynchronizer` | `comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: comm.MuellerMullerTimingSynchronizer
Package: comm

Reset states of Mueller-Muller timing phase synchronizer

Syntax

```
reset(H)
```

Note `comm.MuellerMullerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

`reset(H)` resets the states of the `MuellerMullerTimingSynchronizer` object, `H`.

step

System object: comm.MuellerMullerTimingSynchronizer
Package: comm

Recover symbol timing phase using Mueller-Muller method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Note `comm.MuellerMullerTimingSynchronizer` has been removed. Use `comm.SymbolSynchronizer` instead.

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To deinterleave the input symbols:

1. Define and set up your multiplexed deinterleaver object. See “Construction” on page 3-1070.
2. Call `step` to restore the original ordering of the input sequence according to the properties of `comm.MultiplexedDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*th shift register.

Methods

reset	Reset states of the multiplexed deinterleaver object
step	Deinterleave input symbols using a set of shift registers with specified delays

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Multiplexed Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.MultiplexedInterleaver('Delay',[1; 0; 2; 1]);
deinterleaver = comm.MultiplexedDeinterleaver('Delay',[1; 0; 2; 1]);
```

Generate a random data sequence. Pass the data sequence through the interleaver and deinterleaver.

```
[dataIn,dataOut] = deal([]);            % Initialize data arrays

for k = 1:50
    data = randi([0 7],20,1);           % Generate data sequence
    intData = interleaver(data);        % Interleave sequence
    deIntData = deinterleaver(intData); % Deinterleave sequence

    dataIn = cat(1,dataIn,data);        % Save original data
    dataOut = cat(1,dataOut,deIntData); % Save deinterleaved data
end
```

Determine the delay through the interleaver and deinterleaver.

```
intlvrDelay = finddelay(dataIn,dataOut)
```

```
intlvrDelay = 8
```

After accounting for the delay, confirm that the original and deinterleaved sequences are identical.

```
isequal(dataIn(1:end-intlvrDelay),dataOut(intlvrDelay+1:end))
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: comm.MultiplexedDeinterleaver
Package: comm

Reset states of the multiplexed deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MultiplexedDeinterleaver` object, `H`.

step

System object: comm.MultiplexedDeinterleaver
Package: comm

Deinterleave input symbols using a set of shift registers with specified delays

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.MultiplexedInterleaver System object

Package: comm

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.

To permute the symbols in the input signal:

1. Define and set up your multiplexed interleaver object. See “Construction” on page 3-1077.
2. Call `step` to interleave the input signal according to the properties of `comm.MultiplexedInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*-th shift register.

Methods

reset	Reset states of the multiplexed interleaver object
step	Permute input symbols using a set of shift registers with specified delays

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Multiplexed Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.MultiplexedInterleaver('Delay',[1; 0; 2; 1]);
deinterleaver = comm.MultiplexedDeinterleaver('Delay',[1; 0; 2; 1]);
```

Generate a random data sequence. Pass the data sequence through the interleaver and deinterleaver.

```
[dataIn,dataOut] = deal([]);            % Initialize data arrays

for k = 1:50
    data = randi([0 7],20,1);           % Generate data sequence
    intData = interleaver(data);        % Interleave sequence
    deIntData = deinterleaver(intData); % Deinterleave sequence

    dataIn = cat(1,dataIn,data);        % Save original data
    dataOut = cat(1,dataOut,deIntData); % Save deinterleaved data
end
```

Determine the delay through the interleaver and deinterleaver.

```
intlvrDelay = finddelay(dataIn,dataOut)
```

```
intlvrDelay = 8
```

After accounting for the delay, confirm that the original and deinterleaved sequences are identical.

```
isequal(dataIn(1:end-intlvrDelay),dataOut(intlvrDelay+1:end))
```

```
ans = logical
   1
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalInterleaver` | `comm.MultiplexedDeinterleaver`

Introduced in R2012a

reset

System object: comm.MultiplexedInterleaver
Package: comm

Reset states of the multiplexed interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MultiplexedInterleaver` object, `H`.

step

System object: comm.MultiplexedInterleaver
Package: comm

Permute input symbols using a set of shift registers with specified delays

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Demodulation using OQPSK method

Description

The comm.OQPSKDemodulator object applies pulse shape filtering to the input waveform and demodulates it using the offset quadrature phase shift keying (OQPSK) method. The input is a baseband representation of the modulated signal.

To demodulate a signal that is OQPSK modulated:

1. Create the comm.OQPSKDemodulator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
oqpskdemod = comm.OQPSKDemodulator
oqpskdemod = comm.OQPSKDemodulator(mod)
oqpskdemod = comm.OQPSKDemodulator(Name,Value)
oqpskdemod = comm.OQPSKDemodulator(phase,Name,Value)
```

Description

`oqpskdemod = comm.OQPSKDemodulator` creates a demodulator System object. This object can jointly match-filter and decimate a waveform, and demodulate it using the offset quadrature phase shift keying (OQPSK) method.

`oqpskdemod = comm.OQPSKDemodulator(mod)` creates a demodulator System object with symmetric configuration to the OQPSK modulator object, `mod`.

`oqpskdemod = comm.OQPSKDemodulator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.OQPSKDemodulator('BitOutput',true)`

`oqpskdemod = comm.OQPSKDemodulator(phase,Name,Value)` sets the PhaseOffset property of the created object to `phase` and sets any other specified `Name`, `Value` pairs.

Example: `comm.OQPSKDemodulator(0.5*pi,'SamplesPerSymbol',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`PhaseOffset` — Phase of zeroth point of constellation from π/4

`0` (default) | scalar

Phase of zeroth point of constellation shifted from π/4 radians, specified as a scalar.

Data Types: `double`

`BitOutput` — Option to output data as bits

`false` (default) | `true`

Option to output data as bits, specified as `false` or `true`.

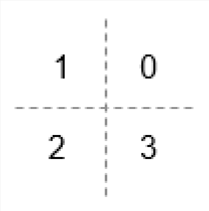
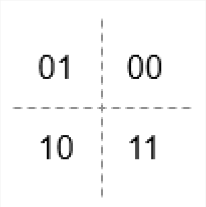
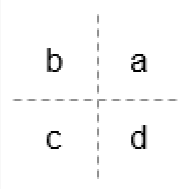
- When you set this property to `false`, the object outputs a column vector of integer values with a length equal to the number of demodulated symbols. The output values are integer representations of two bits and range from `0` to `3`.
- When you set this property to `true`, the object outputs a binary column vector of bit values. The output vector length is twice as long as the number of input symbols.

Data Types: `logical`

`SymbolMapping` — Signal constellation bit mapping

`'Gray'` (default) | `'Binary'` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `'Gray'`, `'Binary'`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping	Comment
`Gray`		The signal constellation mapping is a Gray-encoded integer.
`Binary`		The signal constellation mapping for the input integer `m` ($0 \leq m \leq 3$) is the complex value $e^{(j*(\mathtt{PhaseOffset}+\pi/4)\ +\ j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3		Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Data Types: `char` | `double`

`PulseShape` — Filtering pulse shape

`'Half sine'` (default) | `'Normal raised cosine'` | `'Root raised cosine'` | `'Custom'`

Filtering pulse shape, specified as `'Half sine'`, `'Normal raised cosine'` | `'Root raised cosine'`, or `'Custom'`.

Data Types: `char`

`RolloffFactor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property applies when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterSpanInSymbols` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to `FilterSpanInSymbols` symbols.

Dependencies

This property applies when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterNumerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This property applies when PulseShape is `'Custom'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

`4` (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

`OutputDataType` — Data type assigned to output

`'double'` (default) | `'single'` | `'uint8'`

Data type assigned to output, specified as `'double'`, `'single'`, or `'uint8'`.

Data Types: `char`

Usage

Syntax

```
outsignal = oqpskdemod(waveform)
```

Description

`outsignal = oqpskdemod(waveform)` returns the demodulated output signal. The object produces one output symbol for each input pulse.

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`waveform` — Received waveform
scalar | column vector

Received waveform, specified as a scalar or column vector.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`outsignal` — Demodulated signal
integer vector | bit vector

Demodulated signal, returned as an N_S-element integer vector or bit vector, where N_S is the number of samples.

The received waveform is pulse shaped according to the configuration properties PulseShape and SamplesPerSymbol. The setting of the BitOutput property determines the interpretation of the received waveform.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

OQPSK Signal in AWGN

Create an OQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
oqpskmod = comm.OQPSKModulator('BitInput',true);
oqpskdemod = comm.OQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
```

Create an error rate calculator. To account for the delay between the modulator and demodulator, set the `ReceiveDelay` property to `2`.

```
errorRate = comm.ErrorRate('ReceiveDelay',2);
```

Process 300 frames of data looping through these steps.

- Generate vectors with 100 elements of random binary data.
- OQPSK-modulate the data. The data frames are processed as 50 sample frames of 2-bit binary data.
- Pass the modulated data through the AWGN channel.
- OQPSK-demodulate the data.
- Collect error statistics on the frames of data.

```
for counter = 1:300
    txData = randi([0 1],100,1);
    modSig = oqpskmod(txData);
    rxSig = channel(modSig);
    rxData = oqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 3.3336e-05
```

```
numErrors = errorStats(2)
```

```
numErrors = 1
```

```
numBits = errorStats(3)
```

```
numBits = 29998
```

OQPSK Signal with Root Raised Cosine Filtering

Perform OQPSK modulation and demodulation and apply root raised cosine filtering to a waveform.

System initialization

Define simulation parameters and create objects for OQPSK modulation and demodulation.

```
sps = 12; % samples per symbol
bits = randi([0, 1], 800, 1); % transmission data
```

```
modulator = comm.OQPSKModulator('BitInput',true,'SamplesPerSymbol',sps,'PulseShape','Ro
demodulator = comm.OQPSKDemodulator(modulator);
```

Waveform transmission and reception

Use the `modulator` object to apply OQPSK modulation and transmit filtering to the input data.

```
oqpskWaveform = modulator(bits);
```

Pass the waveform through a channel.

```
snr = 0;
rxWaveform = awgn(oqpskWaveform, snr);
```

Use the `demodulator` object to apply receive filtering and OQPSK demodulation to the waveform.

```
demodData = demodulator(rxWaveform);
```

Compute the bit error rate to confirm the quality of the data recovery.

```
delay = (1+modulator.BitInput)*modulator.FilterSpanInSymbols;
[~, ber] = biterr(bits(1:end-delay), demodData(delay+1:end))
```

```
ber = 0
```

Definitions

Modulation Delays

Digital modulation and demodulation objects incur delays between their inputs and outputs that result in an offset in the arrival sample of the received data. When comparing transmitted data with received data, such as when plotting or computing error statistics, you must take system delays into account. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter and the input/output settings of the object pairs.

Pulse Shape	Input/Output Data (*)	End-to-End Delay Incurred by ay an OQPSK Modulator-Demodulator Object Pair (in samples)
'Half sine' or 'Custom'	Integer	1
	Bit	2
'Normal raised cosine' or 'Root raised cosine'	Integer	FilterSpanInSymbols
	Bit	2*FilterSpanInSymbols
(*) Set the data type property (BitInput for modulation or BitOutput for demodulation) to false for integer data and true for bit data.		

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

System Objects
comm.OQPSKModulator | comm.QPSKDemodulator

Blocks
OQPSK Demodulator Baseband

Topics

Phase Modulation

Introduced in R2012a

comm.OQPSKModulator System object

Package: comm

Modulation using OQPSK method

Description

The comm.OQPSKModulator object modulates the input signal using the offset quadrature phase shift keying (OQPSK) method and applies pulse shape filtering to the output waveform. The output is a baseband representation of the modulated signal.

To modulate a signal using offset quadrature phase shift keying:

1. Create the comm.OQPSKModulator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
oqpskmod = comm.OQPSKModulator
oqpskmod = comm.OQPSKModulator(demod)
oqpskmod = comm.OQPSKModulator(Name,Value)
oqpskmod = comm.OQPSKModulator(phase,Name,Value)
```

Description

`oqpskmod = comm.OQPSKModulator` creates a modulator System object. This object applies offset quadrature phase shift keying (OQPSK) modulation and pulse shape filtering to the input signal.

`oqpskmod = comm.OQPSKModulator(demod)` creates a modulator System object with symmetric configuration to the OQPSK demodulator object, `demod`.

`oqpskmod = comm.OQPSKModulator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.OQPSKModulator('BitInput',true)`

`oqpskmod = comm.OQPSKModulator(phase,Name,Value)` sets the PhaseOffset property of the created object to `phase` and sets any other specified `Name`, `Value` pairs.

Example: `comm.OQPSKModulator(0.5*pi,'SymbolMapping','Binary')`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`PhaseOffset` — Phase of zeroth point of constellation shifted from π/4

`0` (default) | scalar

Phase of zeroth point of constellation shifted from π/4 radians, specified as a scalar.

Data Types: `double`

`BitInput` — Option to provide input in bits

`false` (default) | `true`

Option to provide input in bits, specified as `false` or `true`.

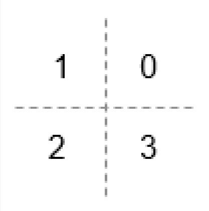
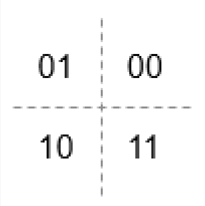
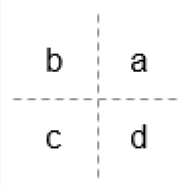
- When this property is set to `false`, the input values must be integer representations of two-bit input segments and range from 0 to 3.
- When this property is set to `true`, the input must be a binary vector of even length. Element pairs are binary representations of integers.

Data Types: `logical`

SymbolMapping — Signal constellation bit mapping

'Gray' (default) | 'Binary' | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as 'Gray', 'Binary', or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping	Comment
Gray		The signal constellation mapping is a Gray-encoded integer.
Binary		The signal constellation mapping for the input integer m (0 ≤ m ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3		Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Data Types: char | double

PulseShape — Filtering pulse shape

'Half sine' (default) | 'Normal raised cosine' | 'Root raised cosine' | 'Custom'

Filtering pulse shape, specified as 'Half sine', 'Normal raised cosine', 'Root raised cosine', or 'Custom'.

Data Types: char

`RolloffFactor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property applies when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterSpanInSymbols` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to `FilterSpanInSymbols` symbols.

Dependencies

This property applies when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterNumerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This property applies when PulseShape is `'Custom'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

`4` (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

`OutputDataType` — Data type assigned to output

`'double'` (default) | `'single'`

Data type assigned to output, specified as `'double'` or `'single'`.

Data Types: `char`

Usage

Syntax

```
waveform = oqpskmod(insignal)
```

Description

`waveform = oqpskmod(insignal)` returns baseband-modulated output. The output waveform is pulse shaped according to the configuration properties PulseShape and SamplesPerSymbol.

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`insignal` — Input signal

integer column vector | bit column vector

Input signal, specified as an N_S-element column vector of integers or bits, where N_S is the number of samples.

The setting of the BitInput property determines the interpretation of the input vector.

Data Types: `double`

Output Arguments

`waveform` — Output waveform

vector

Output waveform, returned as a vector. The output waveform is pulse-shaped according to the configuration properties PulseShape and SamplesPerSymbol.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

OQPSK Signal in AWGN

Create an OQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
oqpskmod = comm.OQPSKModulator('BitInput',true);
oqpskdemod = comm.OQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
```

Create an error rate calculator. To account for the delay between the modulator and demodulator, set the `ReceiveDelay` property to `2`.

```
errorRate = comm.ErrorRate('ReceiveDelay',2);
```

Process 300 frames of data looping through these steps.

- Generate vectors with 100 elements of random binary data.
- OQPSK-modulate the data. The data frames are processed as 50 sample frames of 2-bit binary data.
- Pass the modulated data through the AWGN channel.
- OQPSK-demodulate the data.
- Collect error statistics on the frames of data.

```
for counter = 1:300
    txData = randi([0 1],100,1);
    modSig = oqpskmod(txData);
    rxSig = channel(modSig);
    rxData = oqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 3.3336e-05
```

```
numErrors = errorStats(2)
```

```
numErrors = 1
```

```
numBits = errorStats(3)
```

```
numBits = 29998
```

Create OQPSK Modulator Using Demodulator

Use an OQPSK demodulator object to initialize an OQPSK modulator object while creating it.

Create an OQPSK demodulator, assigning it a phase offset of $\frac{1}{2}\pi$.

```
phase = 0.5*pi;
oqpskdemod = comm.OQPSKDemodulator(phase)
oqpskdemod = 
  comm.OQPSKDemodulator with properties:

   Modulation
         PhaseOffset: 1.5708
       SymbolMapping: 'Gray'
           BitOutput: false

   Filtering
          PulseShape: 'Half sine'
    SamplesPerSymbol: 4

      OutputDataType: 'double'
```

Use the demodulator object to initialize an OQPSK modulator while creating it.

```
oqpskmod = comm.OQPSKModulator(oqpskdemod)
```

```
oqpskmod = 
  comm.OQPSKModulator with properties:

   Modulation
         PhaseOffset: 1.5708
       SymbolMapping: 'Gray'
            BitInput: false

   Filtering
          PulseShape: 'Half sine'
    SamplesPerSymbol: 4

      OutputDataType: 'double'
```

OQPSK Signal with Root Raised Cosine Filtering

Perform OQPSK modulation and demodulation and apply root raised cosine filtering to a waveform.

System initialization

Define simulation parameters and create objects for OQPSK modulation and demodulation.

```
sps = 12; % samples per symbol
bits = randi([0, 1], 800, 1); % transmission data

modulator = comm.OQPSKModulator('BitInput',true,'SamplesPerSymbol',sps,'PulseShape','Ro
demodulator = comm.OQPSKDemodulator(modulator);
```

Waveform transmission and reception

Use the `modulator` object to apply OQPSK modulation and transmit filtering to the input data.

```
oqpskWaveform = modulator(bits);
```

Pass the waveform through a channel.

```
snr = 0;
rxWaveform = awgn(oqpskWaveform, snr);
```

Use the `demodulator` object to apply receive filtering and OQPSK demodulation to the waveform.

```
demodData = demodulator(rxWaveform);
```

Compute the bit error rate to confirm the quality of the data recovery.

```
delay = (1+modulator.BitInput)*modulator.FilterSpanInSymbols;
[~, ber] = biterr(bits(1:end-delay), demodData(delay+1:end))
```

```
ber = 0
```

Definitions

Modulation Delays

Digital modulation and demodulation objects incur delays between their inputs and outputs that result in an offset in the arrival sample of the received data. When comparing transmitted data with received data, such as when plotting or computing error statistics, you must take system delays into account. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter and the input/output settings of the object pairs.

Pulse Shape	Input/Output Data (*)	End-to-End Delay Incurred by ay an OQPSK Modulator-Demodulator Object Pair (in samples)
'Half sine' or 'Custom'	Integer	1
	Bit	2
'Normal raised cosine' or 'Root raised cosine'	Integer	FilterSpanInSymbols
	Bit	2*FilterSpanInSymbols
(*) Set the data type property (BitInput for modulation or BitOutput for demodulation) to false for integer data and true for bit data.		

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

System Objects
comm.OQPSKDemodulator | comm.QPSKModulator

Blocks
OQPSK Modulator Baseband

Topics

Phase Modulation

Introduced in R2012a

comm.OSTBCCombiner System object

Package: comm

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.

To combine input signals and extract the soft information of the symbols encoded by an OSTBC:

1. Define and set up your OSTBC combiner object. See “Construction” on page 3-1105.
2. Call `step` to Combine inputs using an orthogonal space-time block code according to the properties of `comm.OSTBCCombiner`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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

step	Combine inputs using orthogonal space-time block code

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Encode with OSTBC and Calculate Errors

Determine the bit error rate for a QSPK signal employing OSTBC encoding when transmitted through a 4x2 MIMO channel. Perfect channel estimation is assumed to be used by the OSTBC combiner.

Define the system parameters.

```
numTx = 4;          % Number of transmit antennas
numRx = 2;          % Number of receive antennas
Rs = 1e6;           % Sampling rate (Hz)
tau = [0 2e-6];     % Path delays (sec)
pdb = [0 -10];      % Average path gains (dB)
maxDopp = 30;       % Maximum Doppler shift (Hz)
numBits = 12000;    % Number of bits
SNR = 6;            % Signal-to-noise ratio (dB)
```

Set the random number generator to its default state to ensure repeatable results.

```
rng default
```

Create a QPSK modulator System object™. Set the `BitInput` property to `true` and the `SymbolMapping` property to `Gray`.

```
hMod = comm.QPSKModulator(...
    'BitInput',true,...
    'SymbolMapping','Gray');
```

Create a corresponding QPSK demodulator System object. Set the `SymbolMapping` property to `Gray` and the `BitOutput` property to `true`.

```
hDemod = comm.QPSKDemodulator(...
    'SymbolMapping','Gray',...
    'BitOutput',true);
```

Create an OSTBC encoder and combiner pair, where the number of antennas is specified in the system parameters.

```
hOSTBCEnc = comm.OSTBCEncoder(...
    'NumTransmitAntennas',numTx);

hOSTBCComb = comm.OSTBCCombiner(...
    'NumTransmitAntennas',numTx,...
    'NumReceiveAntennas',numRx);
```

Create a flat 4x2 MIMO Channel System object, where the channel characteristics are set using name-value pairs. The path gains are made available to serve as a perfect channel estimate for the OSTBC combiner.

```
hChan = comm.MIMOChannel(...
    'SampleRate',Rs,...
    'PathDelays',tau,...
    'AveragePathGains',pdb,...
    'MaximumDopplerShift',maxDopp,...
    'SpatialCorrelationSpecification','None',...
    'NumTransmitAntennas',numTx,...
    'NumReceiveAntennas',numRx,...
    'PathGainsOutputPort',true);
```

Create an AWGN channel System object in which the noise method is specified as a signal-to-noise ratio.

```
hAWGN = comm.AWGNChannel(...
    'NoiseMethod','Signal to noise ratio (SNR)',...
    'SNR',SNR,...
    'SignalPower',1);
```

Generate a random sequence of bits.

```
data = randi([0 1],numBits,1);
```

Apply QPSK modulation.

```
modData = step(hMod,data);
```

Encode the modulated data using the OSTBC encoder object.

```
encData = step(hOSTBCEnc,modData);
```

Transmit the encoded data through the MIMO channel and add white noise by using the `step` functions of the MIMO and AWGN channel objects, respectively.

```
[chanOut,pathGains] = step(hChan,encData);
rxSignal = step(hAWGN,chanOut);
```

Sum the `pathGains` array along the number of paths (2nd dimension) to form the channel estimate. Apply the `squeeze` function to make its dimensions conform with those of `rxSignal`.

```
chEst = squeeze(sum(pathGains,2));
```

Combine the received MIMO signal and its channel estimate using the `step` function of the OSTBC combiner object. Demodulate the combined signal.

```
combinedData = step(hOSTBCComb,rxSignal,chEst);
receivedData = step(hDemod,combinedData);
```

Compute the number of bit errors and the bit error rate.

```
[numErrors,ber] = biterr(data,receivedData)
```

```
numErrors = 11
```

```
ber = 9.1667e-04
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.OSTBCEncoder`

Introduced in R2012a

step

System object: comm.OSTBCCombiner
Package: comm

Combine inputs using orthogonal space-time block code

Syntax

```
Y = step(H,X,CEST)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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.OSTBCEncoder System object

Package: comm

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.

To encode an input symbol sequence using an orthogonal space-time block code:

1. Define and set up your OSTBC encoder object. See “Construction” on page 3-1115.
2. Call `step` to encode an input symbol sequence according to the properties of `comm.OSTBCEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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

step	Encode input using orthogonal space-time block code

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Encode BPSK Modulated Data with OSTBC

Generate random binary data, modulate using the BPSK modulation scheme, and encode the modulated data using OSTBC.

Generate an 8-by-1 vector of random binary data.

```
data = randi([0 1],8,1);
```

Create BPSK Modulator System object and modulated the data using the `step` function.

```
hMod = comm.BPSKModulator;
modData = step(hMod,data);
```

Create an OSTBC Encoder and encode the modulated signal. As the default number of transmit antennas is 2, you can see that `encData` is an 8-by-2 vector.

```
hOSTBCEnc = comm.OSTBCEncoder;
encData = step(hOSTBCEnc, modData)
```

```
encData = 8×2 complex

  -1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i  -1.0000 - 0.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i
  -1.0000 + 0.0000i   1.0000 + 0.0000i
  -1.0000 + 0.0000i  -1.0000 - 0.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.OSTBCCombiner`

Introduced in R2012a

step

System object: comm.OSTBCEncoder
Package: comm

Encode input using orthogonal space-time block code

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Generate OVSF code

Description

The `OVSFCode` object generates an orthogonal variable spreading factor (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.

To generate an OVSF code:

1. Define and set up your OVSF code object. See “Construction” on page 3-1121.
2. Call `step` to generate an OVSF code according to the properties of `comm.OVSFCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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 *C*1 = [1].
Next, assume that *Cn* is defined and let *Cn*(*k*) denote the *k*-th row of *Cn*.
Then, *C*2*n* = [*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 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 on page 3-0 property.

OutputDataType

Data type of output

Specify output data type as one of double | int8. The default is double.

Methods

reset	Reset states of OVSF code generator object
step	Generate OVSF code

Common to All System Objects	
`release`	Allow System object property value changes

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`

Introduced in R2012a

reset

System object: comm.OVSFCode
Package: comm

Reset states of OVSF code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `OVSFCode` object, `H`.

step

System object: comm.OVSFCode
Package: comm

Generate OVSF code

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H)` outputs a frame of the OVSF code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To demodulate a signal that was modulated using M-ary pulse amplitude modulation:

1. Define and set up your PAM demodulator object. See “Construction” on page 3-1126.
2. Call `step` to demodulate the signal according to the properties of `comm.PAMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0) 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` on page 3-0) 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 $0 \le m \le$ (`ModulationOrder`–1) maps to the complex value 2*m*-`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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Smallest unsigned integer`.

When you set the `BitOutput` on page 3-0 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 “Fixed-Point Support for MATLAB System Objects in DSP System Toolbox” (DSP System Toolbox).

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` on page 3-0 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` on page 3-0 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` on page 3-0 property to false and the `ProductDataType` on page 3-0 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` on page 3-0 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

`FullPrecisionOverride` on page 3-0 property to false and the `SumDataType` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using M-ary PAM method

Common to All System Objects	
`release`	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PAMModulator`

Introduced in R2012a

constellation

System object: comm.PAMDemodulator
Package: comm

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 PAM Signal Constellation

Create `comm.PAMModulator` and `comm.PAMDemodulator` System objects™, and then calculate their ideal signal constellations.

Create a modulator and demodulator objects.

```
mod = comm.PAMModulator;
demod = comm.PAMModulator;
```

Calculate the constellation points.

```
refMod = constellation(mod)
```

```
refMod = 4×1
```

```
    -3
    -1
     1
     3
```

```
refDemod = constellation(demod)
```

```
refDemod = 4×1

    -3
    -1
     1
     3
```

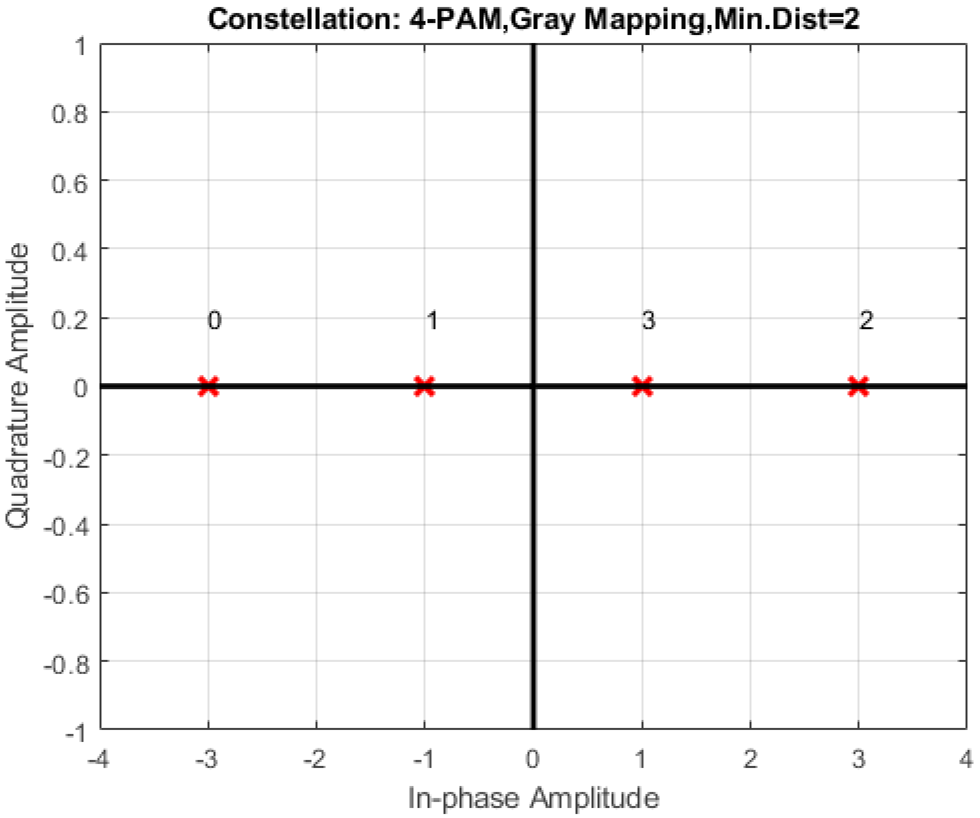
Verify that both objects produce the same points.

```
isequal(refMod,refDemod)
```

```
ans = logical
   1
```

Display the ideal signal constellation.

```
constellation(mod)
```


Constellation: 4-PAM,Gray Mapping,Min.Dist=2
Quadrature Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
3
2
-4
-3
-2
-1
0
1
2
3
4
In-phase Amplitude

step

System object: comm.PAMDemodulator
Package: comm

Demodulate using M-ary PAM method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To modulate a signal using M-ary pulse amplitude modulation:

1. Define and set up your PAM modulator object. See “Construction” on page 3-1137.
2. Call `step` to modulate the signal according to the properties of `comm.PAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0). 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` on page 3-0) 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 2*m*– `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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using M-ary PAM method

Common to All System Objects	
`release`	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PAMDemodulator`

Introduced in R2012a

constellation

System object: comm.PAMModulator
Package: comm

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 PAM Signal Constellation

Create `comm.PAMModulator` and `comm.PAMDemodulator` System objects™, and then calculate their ideal signal constellations.

Create a modulator and demodulator objects.

```
mod = comm.PAMModulator;
demod = comm.PAMModulator;
```

Calculate the constellation points.

```
refMod = constellation(mod)
```

```
refMod = 4×1
```

```
    -3
    -1
     1
     3
```

```
refDemod = constellation(demod)
```

```
refDemod = 4×1

    -3
    -1
     1
     3
```

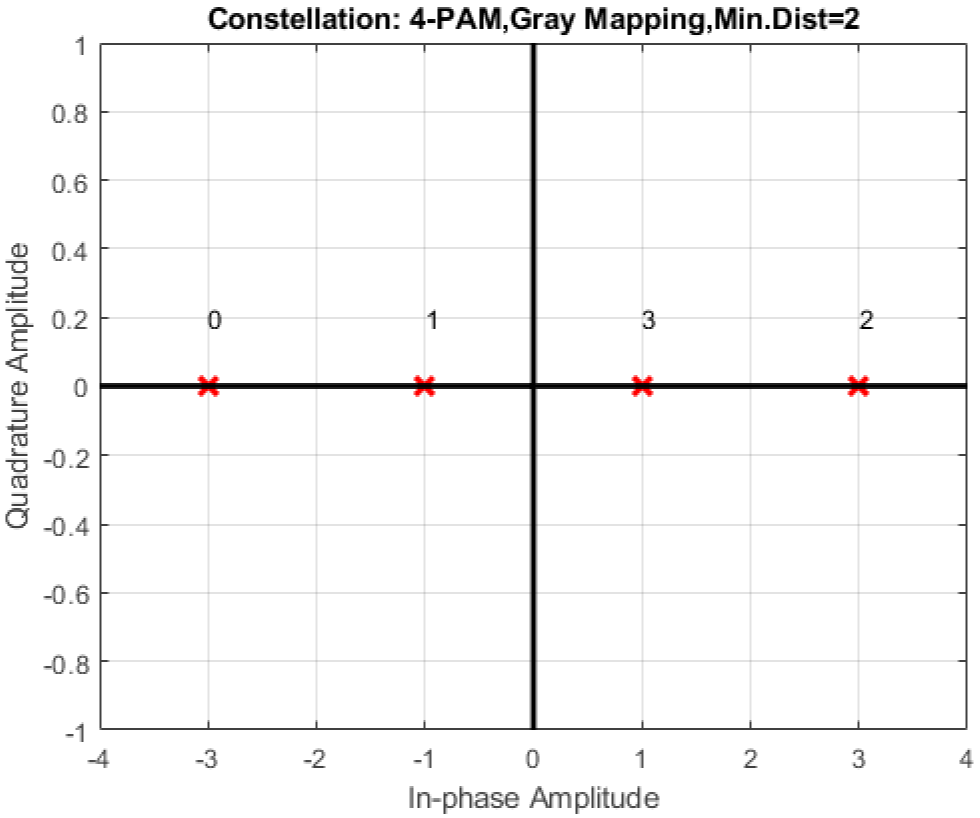
Verify that both objects produce the same points.

```
isequal(refMod,refDemod)
```

```
ans = logical
   1
```

Display the ideal signal constellation.

```
constellation(mod)
```


Constellation: 4-PAM,Gray Mapping,Min.Dist=2
Quadrature Amplitude
In-phase Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-4
-3
-2
-1
0
1
2
3
4
0
1
3
2

step

System object: comm.PAMModulator
Package: comm

Modulate using M-ary PAM method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PhaseFrequencyOffset System object

Package: comm

Apply phase and frequency offsets to input signal

Description

The `PhaseFrequencyOffset` object applies phase and frequency offsets to an incoming signal.

To apply phase and frequency offsets to the input signal:

1. Define and set up your phase frequency offset object. See “Construction” on page 3-1146.
2. Call `step` to apply phase and frequency offsets to the input signal according to the properties of `comm.PhaseFrequencyOffset`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0 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 on page 3-0 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 on page 3-0 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` on page 3-0 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

step	Apply phase and frequency offsets to input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Introduce Phase Offset to 16-QAM Signal

Introduce a phase offset to a 16-QAM signal and view its effect on the constellation.

Create a QAM modulator and a phase frequency offset System object™. Set the phase offset to 30 degrees.

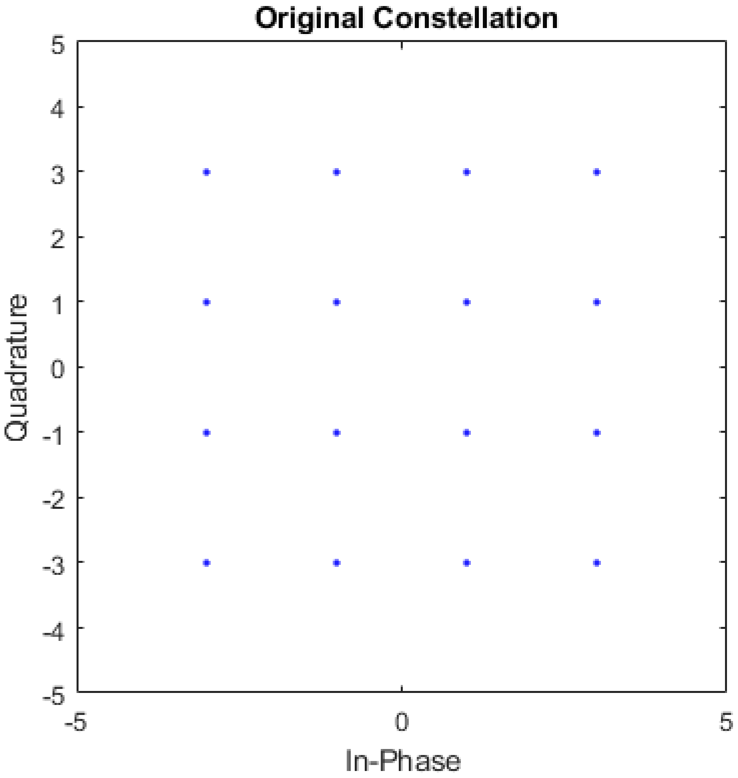
```
qamModulator = comm.RectangularQAMModulator('ModulationOrder',16);
pfo = comm.PhaseFrequencyOffset('PhaseOffset',30);
```

Generate random symbols and apply 16-QAM modulation.

```
data = (0:15)';
modData = qamModulator(data);
```

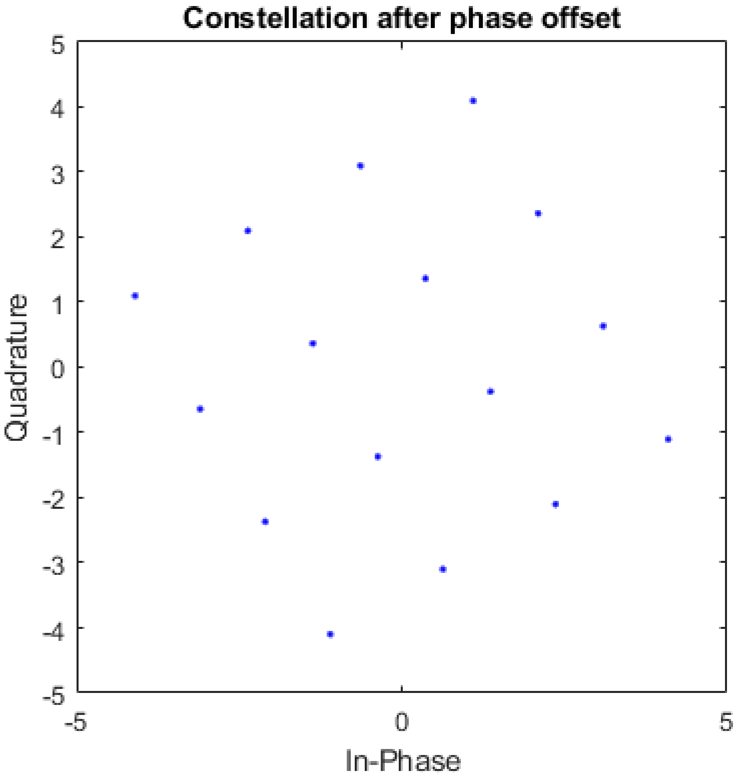
Plot the 16-QAM constellation.

```
scatterplot(modData);
title(' Original Constellation')
xlim([-5 5])
ylim([-5 5])
```



Introduce a phase offset using `pfo` and plot the offset constellation. Note that it has been shifted 30 degrees.

```
impairedData = pfo(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` on page 3-0 property, which you must specify. The block senses the sample time of the signal and therefore does not have a corresponding parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.MemorylessNonlinearity` | `comm.PhaseNoise` | `comm.ThermalNoise`

Introduced in R2012a

step

System object: comm.PhaseFrequencyOffset
Package: comm

Apply phase and frequency offsets to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,FRQ)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PhaseNoise System object

Package: comm

Apply phase noise to baseband signal

Description

The `comm.PhaseNoise` System object adds phase noise to a complex signal. This object emulates impairments introduced by the local oscillator of a wireless communication transmitter or receiver. The object generates filtered phase noise according to the specified spectral mask and adds it to the input signal. For a description of the phase noise modeling, see “Algorithms” on page 3-1161.

To add phase noise using a `comm.PhaseNoise` object:

1. Create the comm.PhaseNoise object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
phznoise = comm.PhaseNoise
phznoise = comm.PhaseNoise(Name,Value)
phznoise = comm.PhaseNoise(level,offset,samplerate)
```

Description

`phznoise = comm.PhaseNoise` creates a phase noise System object with default property values.

`phznoise = comm.PhaseNoise(Name,Value)` creates a phase noise object 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`).

`phznoise = comm.PhaseNoise(level,offset,samplerate)` creates a phase noise object with the phase noise level, frequency offset, and sample rate properties specified as value-only arguments. When specifying a value-only argument, you must specify all preceding value-only arguments.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Level` — Phase noise level

`[-60 -80]` (default) | vector of negative scalars

Phase noise level in decibels relative to carrier per hertz (dBc/Hz), specified as a vector of negative scalars. The `Level` and `FrequencyOffset` properties must have the same length.

Data Types: `double`

`FrequencyOffset` — Frequency offset

`[20 200]` (default) | vector of positive increasing values

Frequency offset in Hz, specified as a vector of positive increasing values. The `Level` and `FrequencyOffset` properties must have the same length.

Data Types: `double`

`SampleRate` — Sample rate

`1024` (default) | positive scalar

Sample rate in samples per second, specified as a positive scalar. To avoid aliasing, the sample rate must be greater than twice the largest value specified by `FrequencyOffset`.

Data Types: `double`

Usage

Syntax

```
out = phznoise(in)
```

Description

`out = phznoise(in)` adds phase noise, specified by the `phznoise` System object, to the input signal. The result is returned in `out`.

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`in` — Input signal

complex column vector

Input signal, specified as an N_S-by-1 vector of complex values. N_S is the number of samples.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`out` — Output signal

column vector

Output signal, returned as an N_S-by-1 vector of complex values. N_S equals the number of samples in the input signal.

Data Types: double
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to comm.PhaseNoise

visualize	Visualize spectrum mask of phase noise

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Phase Noise Effects on 16-QAM Signal

Add a phase noise vector and frequency offset vector to a 16-QAM signal. Then plot the signal.

Create a 16-QAM modulator having an average constellation power of 10 W.

```
modulator = comm.RectangularQAMModulator(16, ...
    'NormalizationMethod','Average power','AveragePower',10);
```

Create a phase noise object.

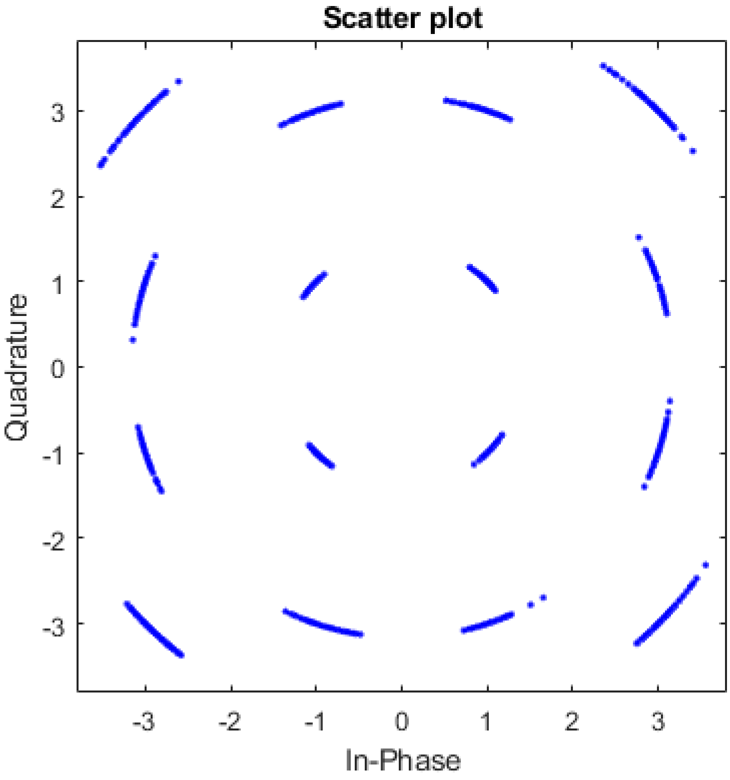
```
pnoise = comm.PhaseNoise('Level',-50,'FrequencyOffset',20);
```

Generate modulated symbols.

```
data = randi([0 15],1000,1);
modData = modulator(data);
```

Apply phase noise and plot the result.

```
y = pnoise(modData);
scatterplot(y)
```



Phase Noise Effects on Signal Spectrum

Create a sine wave generator having a carrier frequency of 100 Hz, a sample rate of 1000 Hz, and a frame size of 10,000 samples.

```
sinewave = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
    'SamplesPerFrame',1e4,'ComplexOutput',true);
```

Create a phase noise object. Specify the phase noise level to be –40 dBc/Hz for a 100 Hz offset and –70 dBc/Hz for a 200 Hz offset.

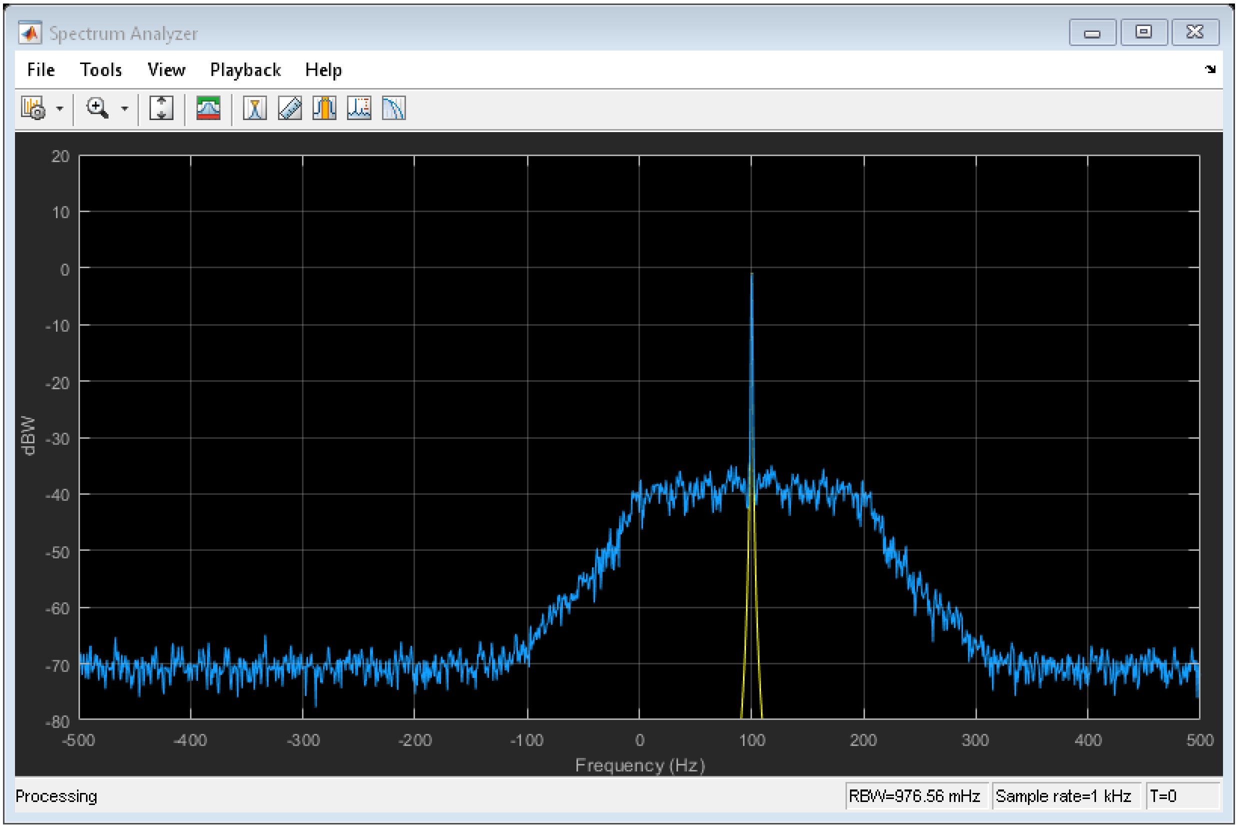
```
pnoise = comm.PhaseNoise('Level',[-40 -70],'FrequencyOffset',[100 200], ...
    'SampleRate',1000);
```

Create a spectrum analyzer.

```
spectrum = dsp.SpectrumAnalyzer('NumInputPorts',2,'SampleRate',1000, ...
    'SpectralAverages',10,'PowerUnits','dBW');
```

Generate a sine wave and add phase noise to it. Plot the spectrum of the sine wave and the noisy signal.

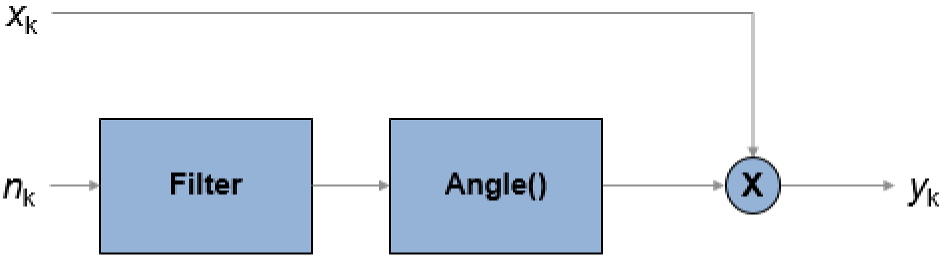
```
x = sinewave();
y = pnoise(x);

spectrum(x,y)
```



The phase noise is –40 dBW within ±100 Hz of the carrier. The phase noise is –70 dB below the carrier for offsets greater than 200 Hz.

Algorithms

The output signal, y_k, is related to input sequence x_k by $y_k=x_k e^{j\varphi_k}$, where φ_k is the phase noise. The phase noise is filtered Gaussian noise such that $\varphi_k=f(n_k)$, where n_k is the noise sequence and f represents a filtering operation.



To model the phase noise, define the power spectrum density (PSD) mask characteristic by specifying scalar or vector values for the frequency offset and phase noise level.

- For a scalar frequency offset and phase noise level specification, an IIR digital filter computes the spectrum mask. The spectrum mask has a 1/*f* characteristic that passes through the specified point.
- For a vector frequency offset and phase noise level specification, an FIR filter computes the spectrum mask. The spectrum mask is interpolated across log10(*f*). It is flat from DC to the lowest frequency offset, and from the highest frequency offset to half the sample rate.

IIR Digital Filter

For the IIR digital filter, the numerator coefficient is

$$\lambda = \sqrt{2\pi f_{offset} 10^{L/10}}\ ,$$

where f_{offset} is the frequency offset in Hz and L is the phase noise level in dBc/Hz. The denominator coefficients, γ_{i}, are recursively determined as

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1}\ ,$$

where $\gamma_1 = 1$, $i = \{1, 2, \ldots, N_t\}$, and N_{t} is the number of filter coefficients. N_{t} is a power of 2, from 2^7 to 2^{19}. The value of N_{t} grows as the phase noise offset decreases towards 0 Hz.

FIR Filter

For the FIR filter, the phase noise level is determined through log10(*f*) interpolation for frequency offsets over the range [*df*, f_s / 2], where *df* is the frequency resolution and f_s is the sample rate. The phase noise is flat from 0 Hz to the smallest frequency offset, and from the largest frequency offset to f_s / 2. The frequency resolution is equal to $\frac{f_s}{2}\left(\frac{1}{N_t}\right)$, where N_t is the number of coefficients, and is a power of 2 less than or equal to 2^{16}. If N_t < 2^7, a time domain FIR filter is used. Otherwise, a frequency domain FIR filter is used.

The algorithm increases N_t until these conditions are met:

- The frequency resolution is less than the minimum value of the frequency offset vector.
- The frequency resolution is less than the minimum difference between two consecutive frequencies in the frequency offset vector.
- The maximum number of FIR filter taps is 2^{16}.

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*. Vol. 83, No. 5, May, 1995, pp 802–827.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects

`comm.MemorylessNonlinearity` | `comm.PhaseFrequencyOffset`

Blocks

Phase Noise

Introduced in R2012a

visualize

Package: comm

Visualize spectrum mask of phase noise

Syntax

```
visualize(phznoise)
```

Description

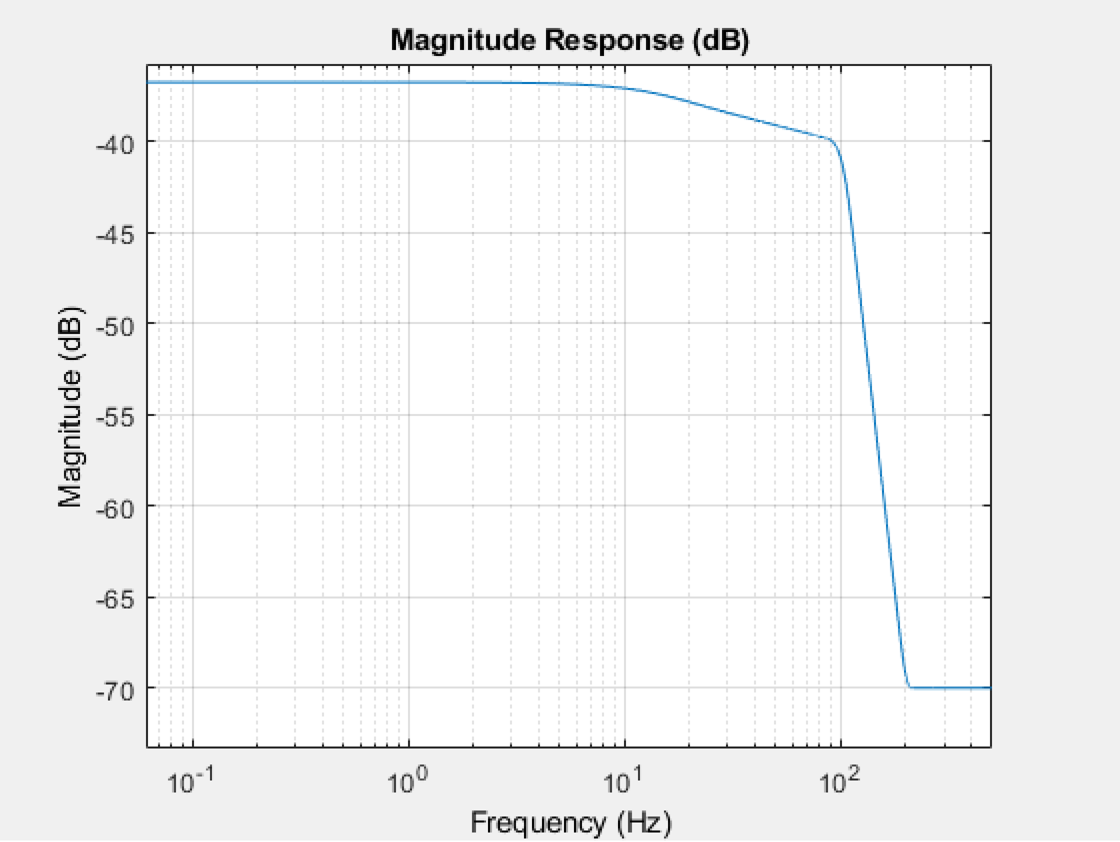
`visualize(phznoise)` displays the magnitude response of the filter defined by the `comm.PhaseNoise` System object. The function uses the `fvtool` function to display the magnitude response.

Examples

Visualize Spectrum Mask of Phase Noise

Create a phase noise object and display the spectral mask.

```
pnoise = comm.PhaseNoise('Level',[-40 -70],'FrequencyOffset',[100 200], ...
    'SampleRate',1000);
visualize(pnoise)
```



Input Arguments

phznoise — Phase noise
comm.PhaseNoise System object

Phase noise that defines the spectrum mask that is displayed, specified as a comm.PhaseNoise System object.

See Also

Functions
`fvtool`

System Objects
`comm.PhaseNoise`

Introduced in R2012a

comm.PNSequence System object

Package: comm

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.

To generate a PN sequence:

1. Define and set up your PN sequence object. See “Construction” on page 3-1168.
2. Call `step` to generate a PN sequence according to the properties of `comm.PNSequence`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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 `'z^6 + z + 1'`. You can specify the generator polynomial as a character vector or 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. Lastly, 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 $N = 2^n - 1$ (applies only to maximal length sequences).

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` on page 3-0 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 samples output per frame

Number of samples output per frame by the PN sequence object, specified as positive integer. The default is 1. If you set this property to a value of *M*, then the object 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

Option to output bit-packed words

Option to output bit-packed words, specified as false or true. Set this property to true to enable bit-packed outputs. The first bit from the left in the bit-packed word is considered the most significant bit. The default is false.

When `BitPackedOutput` is `true`, the object 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.

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 from `1` to `32`. The default is `8`.

Dependencies

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.

Dependencies

This property applies when you set the `BitPackedOutput` property to `true`.

OutputDataType

Data type of output

Specify the output data type as one of these:

- When BitPackedOutput property is `false`, `OutputDataType` can be `'double'`, `'logical'`, or `'Smallest unsigned integer'`.
- When BitPackedOutput property is `true`, `OutputDataType` can be `'double'` or `'Smallest integer'`.

The default is `double`.

Note You must have a Fixed-Point Designer user license to use this property in `'Smallest unsigned integer'` or `'Smallest integer'` mode.

Dependencies

The valid settings for output data type depends on the setting of BitPackedOutput.

Methods

reset	Reset states of PN sequence generator object
step	Generate a pseudo-noise (PN) sequence

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate Maximal Length PN Sequences

Generate a 14-sample frame of a maximal length PN sequence given generator polynomial, $x^3 + x^2 + 1$.

Generate PN sequence data by using the `comm.PNSequence` object. The sequence repeats itself as it contains 14 samples while the maximal sequence length is only 7 samples ($2^3 - 1$).

```
pnSequence = comm.PNSequence('Polynomial',[3 2 0], ...
    'SamplesPerFrame',14,'InitialConditions',[0 0 1]);
x1 = pnSequence();
[x1(1:7) x1(8:14)]
```

```
ans = 7×2

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

Create another maximal length sequence based on the generator polynomial, $x^4 + x + 1$.
As it is a fourth order polynomial, the sequence repeats itself after 15 samples ($2^4 - 1$).

```
pnSequence2 = comm.PNSequence('Polynomial','x^4+x+1', ...
    'InitialConditions',[0 0 0 1],'SamplesPerFrame',30);
x2 = pnSequence2();
[x2(1:15) x2(16:30)]
```

```
ans = 15×2

     1     1
     0     0
     0     0
     0     0
     1     1
     0     0
     0     0
     1     1
     1     1
     0     0
       ⋮
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GoldSequence` | `comm.KasamiSequence`

Introduced in R2012a

reset

System object: comm.PNSequence
Package: comm

Reset states of PN sequence generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `PNSequence` object, `H`.

step

System object: comm.PNSequence
Package: comm

Generate a pseudo-noise (PN) sequence

Syntax

```
Y = step(H)
Y = step(H,MASK)
Y = step(H,RESET)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PreambleDetector System object

Package: comm

Detect preamble in data

Description

The `comm.PreambleDetector` System object detects a preamble in an input data sequence. A preamble is a set of symbols or bits used in packet-based communication systems to indicate the start of a packet. The preamble detector object finds the location corresponding to the end of the preamble.

To detect a preamble in an input data sequence:

1. Create a `comm.PreambleDetector` object and set the properties of the object.
2. Call `step` to detect the presence of a preamble.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`prbdet = comm.PreambleDetector` creates a preamble detector object, `prbdet`, using the default properties.

`prbdet = comm.PreambleDetector(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

`prbdet = comm.PreambleDetector(prb,Name,Value)` specifies the preamble, `prb` in addition to those properties specified by using `Name,Value` pairs.

Example:

```
prbdet = comm.PreambleDetector('Input','Bit','Detections','First');
```

Properties

Input — Type of input data

'Symbol' (default) | 'Bit'

Type of input data, specified as 'Symbol' or 'Bit'. For binary inputs, set this parameter to 'Bit'. For all other inputs, set this parameter to 'Symbol'. Symbol data can be of data type single or double while bit data can, in addition, support the Boolean, int8, and uint8 data types.

Preamble — Preamble sequence

[1+1i; 1-1i] (default) | column vector

Preamble sequence, specified as a real or complex column vector. The object uses this sequence to detect the presence of the preamble in the input data. If Input is 'Bit', the preamble must be a binary sequence. If Input is 'Symbol', the preamble can be any real or complex sequence.

Data Types: double | single | logical | int8 | uint8

Threshold — Detection threshold

3 (default) | nonnegative scalar

Detection threshold, specified as a nonnegative scalar. When the computed detection metric is greater than or equal to Threshold, the preamble is detected. This property is available when Input is set to 'Symbol'. Tunable.

Detections — Number of preambles to detect

'All' (default) | 'First'

Number of preambles to detect, specified as 'All' or 'First'.

- 'All' — Detects all the preambles in the input data sequence.
- 'First' — Detect only the first preamble in the input data sequence.

Methods

reset	Reset states of preamble detector object
step	Detect preamble in data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Detect Preamble from Binary Data Sequence

Specify a six-bit preamble.

```
prb = [1 0 0 1 0 1]';
```

Create a preamble detector object using preamble `prb` and taking bit inputs.

```
prbdet = comm.PreambleDetector(prb,'Input','Bit');
```

Generate a binary data sequence containing two preambles and using random bits to represent the data fields.

```
pkt = [prb; randi([0 1],10,1); prb; randi([0 1],10,1)];
```

Locate the indices of the two preambles. The indices correspond to the end of the preambles.

```
idx = prbdet(pkt)
```

```
idx = 

     6
    22
```

The detector correctly identified indices 6 and 22 as the end of the two preambles inserted in the sequence.

Detect Preamble in Noisy QPSK Signal

Create a preamble and apply QPSK modulation.

```
p1 = [0 1 2 3 3 2 1 0]';
p = [p1; p1];
prb = pskmod(p,4,pi/4,'gray');
```

Create a `comm.PreambleDetector` object using preamble `prb`.

```
prbdet = comm.PreambleDetector(prb)
```

```
prbdet = 
  comm.PreambleDetector with properties:

         Input: 'Symbol'
      Preamble: [16x1 double]
     Threshold: 3
    Detections: 'All'
```

Generate a sequence of random symbols. The first sequence represents the last 20 symbols from a previous packet. The second sequence represents the symbols from the current packet.

```
d1 = randi([0 3],20,1);
d2 = randi([0 3],100,1);
```

Modulate the two sequences.

```
x1 = pskmod(d1,4,pi/4,'gray');
x2 = pskmod(d2,4,pi/4,'gray');
```

Create a sequence of modulated symbols consisting of the remnant of the previous packet, the preamble, and the current packet.

```
y = [x1; prb; x2];
```

Add white Gaussian noise.

```
z = awgn(y,10);
```

Determine the preamble index and the detection metric.

```
[idx,detmet] = prbdet(z);
```

Calculate the number of elements in `idx`. Because the number of elements is greater than one, the detection threshold is too low.

```
numel(idx)
```

```
ans = 80
```

Display the five largest detection metrics.

```
detmetSort = sort(detmet,'descend');
detmetSort(1:5)
```

```
ans = 5×1

   16.3115
   13.6900
   10.5698
    9.1920
    8.9706
```

Increase the threshold and determine the preamble index.

```
prbdet.Threshold = 15;
idx = prbdet(z)
```

```
idx = 36
```

The result of 36 corresponds to the sum of the preamble length (16) and the remaining samples in the previous packet (20). This indicates that the preamble has been successfully detected.

Algorithms

Bit Inputs

When the input data is composed of bits, the preamble detector uses an exact pattern match.

Symbol Inputs

When the input data is composed of symbols, the preamble detector uses a cross-correlation algorithm. An FIR filter, in which the coefficients are specified from the preamble, computes the cross-correlation between the input data and the preamble. When a sequence of input samples matches the preamble, the filter output reaches its peak. The index of the peak corresponds to the end of the preamble sequence in the input data. See Discrete FIR Filter for further information on the FIR filter algorithm.

The cross-correlation values that are greater than or equal to the specified threshold are reported as peaks. As a result, there may be no detected peaks or there may be as many detected peaks as there are input samples. Consequently, the selection of the detection threshold is very important.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer` | `comm.CoarseFrequencyCompensator` | `comm.SymbolSynchronizer`

Introduced in R2016b

reset

System object: comm.PreambleDetector
Package: comm

Reset states of preamble detector object

Syntax

```
reset(prbdet)
```

Description

`reset(prbdet)` resets the states of the `PreambleDetector` object, `prbdet`.

Introduced in R2016b

step

System object: comm.PreambleDetector
Package: comm

Detect preamble in data

Syntax

```
idx = step(prbdet,x)
[idx,detmet] = step(prbdet,x)
idx = prbdet(x)
[idx,detmet] = prbdet(x)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`idx = step(prbdet,x)` returns the location of the end of the preamble in data sequence `x`, using preamble detector `prbdet`. The index is of data type `double`.

`[idx,detmet] = step(prbdet,x)` also returns the detection metric, `detmet`. This syntax is available when the `Input` property is `'Symbol'`. `detmet` has the same dimensions and data type as `x`.

The output, `detmet`, is determined by one of these algorithms:

- If either the preamble or input data is complex, the detection metric is the absolute value of the cross-correlation of the preamble and the input signal.
- If both the preamble and input data are real, the detection metric is the cross-correlation of the preamble and the input signal.

`idx = prbdet(x)` is equivalent to the first syntax.

`[idx,detmet] = prbdet(x)` is equivalent to the second syntax.

Note `prbdet` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2016b

comm.PSKCarrierPhaseSynchronizer System object

Package: comm

Recover carrier phase of baseband PSK signal

Note comm.PSKCarrierPhaseSynchronizer has been removed. Use comm.CarrierSynchronizer instead.

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.

To recover the carrier phase of the input signal using the M-Power method:

1. Define and set up your PSK carrier phase synchronizer object. See “Construction” on page 3-1189.
2. Call step to recover the carrier phase of the input signal according to the properties of comm.PSKCarrierPhaseSynchronizer. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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

reset	Reset states of the PSK carrier phase synchronizer object
step	Recover baseband PSK signal's carrier phase

Common to All System Objects	
release	Allow System object property value changes

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer` | `comm.PSKModulator`

Introduced in R2012a

reset

System object: comm.PSKCarrierPhaseSynchronizer
Package: comm

Reset states of the PSK carrier phase synchronizer object

Syntax

```
reset(H)
```

Note `comm.PSKCarrierPhaseSynchronizer` has been removed. Use `comm.CarrierSynchronizer` instead.

Description

`reset(H)` resets the states of the `PSKCarrierPhaseSynchronizer` object, `H`.

step

System object: comm.PSKCarrierPhaseSynchronizer
Package: comm

Recover baseband PSK signal's carrier phase

Syntax

```
[Y,PH] = step(H,X)
```

Note `comm.PSKCarrierPhaseSynchronizer` has been removed. Use `comm.CarrierSynchronizer` instead.

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PSKCoarseFrequencyEstimator System object

Package: comm

Estimate frequency offset for PSK signal

Description

The `PSKCoarseFrequencyEstimator` System object estimates frequency offset for a PSK signal.

To estimate frequency offset for a PSK signal:

1. Define and set up your PSK coarse frequency estimator object. See "Construction" on page 3-1194.
2. Call `step` to estimate frequency offset for a PSK signal according to the properties of `comm.PSKCoarseFrequencyEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.PSKCoarseFrequencyEstimator` creates a PSK coarse frequency offset estimator object, `H`. This object uses an open-loop technique to estimate the carrier frequency offset in a received PSK signal.

`H = comm.PSKCoarseFrequencyEstimator(Name,Value)` creates a PSK coarse frequency offset estimator 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

`ModulationOrder`

Modulation order the object uses

Specify the modulation order of the PSK signal as a positive, real scalar of data type double. This value must be a positive power of 2. The default is `4`.

`Algorithm`

Estimation algorithm to object uses

Specify the estimation algorithm as one of `FFT-based` or `Correlation-based`. The default is `FFT-based`.

`FrequencyResolution`

Desired frequency resolution (Hz)

Specify the desired frequency resolution for offset frequency estimation as a positive, real scalar of data type double. This property establishes the FFT length used to perform spectral analysis, and must be less than or equal to half the `SampleRate` on page 3-0 property. This property applies only if the `Algorithm` property is `FFT-based`. The default is `0.001`.

`MaximumOffset`

Maximum measurable frequency offset (Hz)

Specify the maximum measurable frequency offset as a positive, real scalar of data type double. The default is `0.05`.

The value of this property must be less than `SampleRate` on page 3-0 / `ModulationOrder` on page 3-0. It is recommended that `MaximumOffset` on page 3-0 be less than or equal to `SampleRate` on page 3-0 /(4*`ModulationOrder` on page 3-0). This property is active only if the `Algorithm` property is `Correlation-based`.

`SampleRate`

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type double. The default is 1.

Methods

reset	Reset states of the PSKCoarseFrequencyEstimator object
step	Estimate frequency offset for PSK signal

Common to All System Objects	
release	Allow System object property value changes

Examples

Correct For a Frequency Offset in a QPSK Signal

Estimate and correct for a -250 Hz frequency offset in a QPSK signal using the PSK Coarse Frequency Estimator System object™.

Create a square root raised cosine transmit filter System object.

```
txfilter = comm.RaisedCosineTransmitFilter;
```

Create a phase frequency offset object, where the FrequencyOffset property is set to -250 Hz and SampleRate is set to 4000 Hz using name-value pairs.

```
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',-250, ...
    'SampleRate',4000);
```

Create a PSK coarse frequency estimator System object with a sample rate of 4 kHz and a frequency resolution of 1 Hz.

```
frequencyEst = comm.PSKCoarseFrequencyEstimator(...
    'SampleRate',4000, ...
    'FrequencyResolution',1);
```

Create a second phase frequency offset object to correct the offset. Set the `FrequencyOffsetSource` property to `Input port` so that the frequency correction estimate is an input argument.

```
pfoCorrect = comm.PhaseFrequencyOffset(...
    'FrequencyOffsetSource','Input port', ...
    'SampleRate',4000);
```

Generate a QPSK signal, filter the signal, apply the frequency offset, and pass the signal through the AWGN channel.

```
modData = pskmod(randi([0 3],4096,1),4,pi/4);     % Generate QPSK signal
txFiltData = txfilter(modData);                   % Apply Tx filter
offsetData = pfo(txFiltData);                     % Apply frequency offset
noisyData = awgn(offsetData,25);                  % Pass through AWGN channel
```

Estimate the frequency offset by using `frequencyEst`. Observe that the estimate is close to the -250 Hz target.

```
estFreqOffset = frequencyEst(noisyData)
```

```
estFreqOffset = -250
```

Correct for the frequency offset using `pfoCorrect` and the inverse of the estimated frequency offset.

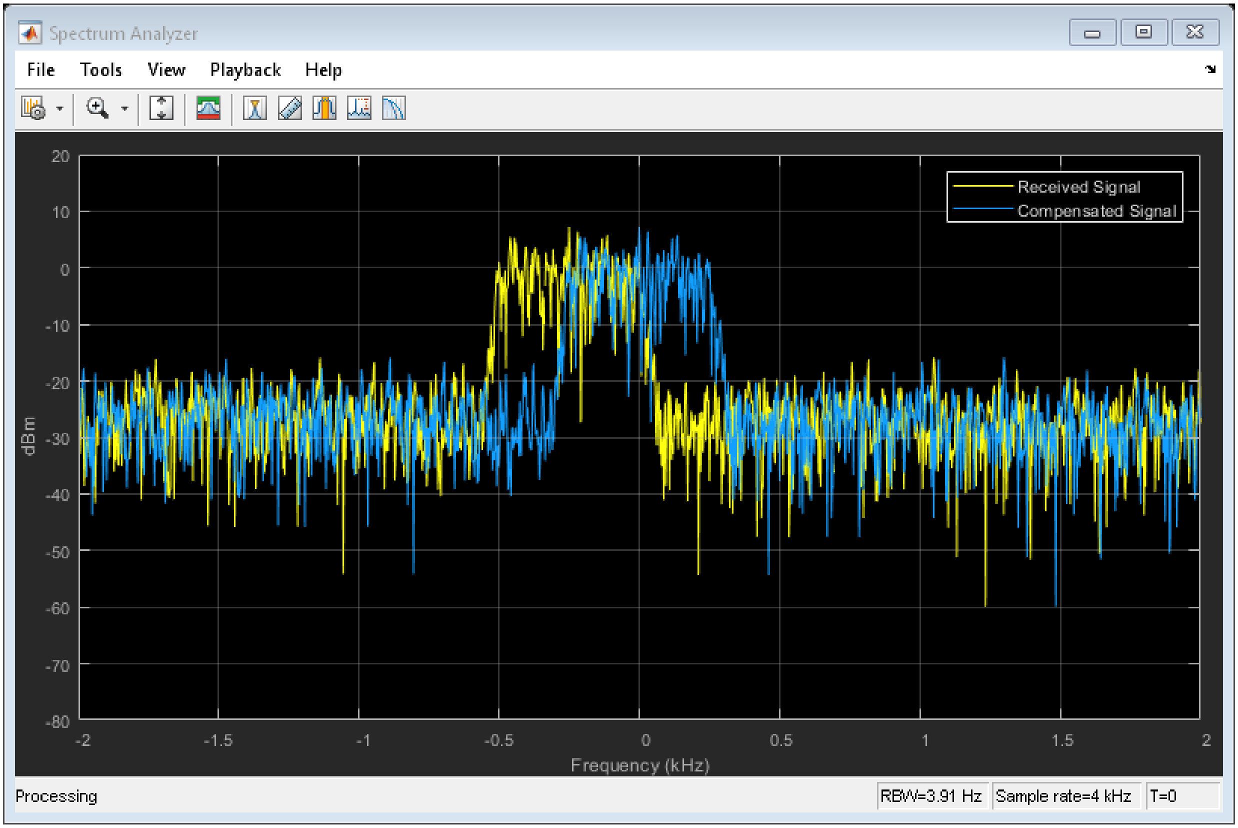
```
compensatedData = pfoCorrect(noisyData,-estFreqOffset);
```

Create a spectrum analyzer object to view the frequency response of the signals.

```
spectrum = dsp.SpectrumAnalyzer('SampleRate',4000, 'ShowLegend',true, ...
    'ChannelNames',{'Received Signal' 'Compensated Signal'});
```

Plot the frequency response of the received signal, which is shifted 250 Hz to the left, and of the compensated signal using the spectrum analyzer. The compensated signal is now properly centered.

```
spectrum([noisyData compensatedData]);
```



Selected Bibliography

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions”, *IEEE Transactions on Communications*, Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

See Also

`comm.PhaseFrequencyOffset` | `comm.QAMCoarseFrequencyEstimator` | `dsp.FFT`

Introduced in R2013b

reset

System object: comm.PSKCoarseFrequencyEstimator
Package: comm

Reset states of the `PSKCoarseFrequencyEstimator` object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the internal states of the `PSKCoarseFrequencyEstimator` object, `H`.

step

System object: comm.PSKCoarseFrequencyEstimator
Package: comm

Estimate frequency offset for PSK signal

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a complex column vector of data type double. The step method outputs the estimate `Y` as a scalar of type double.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PSKDemodulator System object

Package: comm

Demodulate using M-ary PSK method

Description

The PSKDemodulator object demodulates an input signal using the M-ary phase shift keying (M-PSK) method.

To demodulate a signal that was modulated using phase shift keying:

1. Define and set up your PSK demodulator object. See “Construction” on page 3-1202.
2. Call step to demodulate the signal according to the properties of comm.PSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

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` on page 3-0) 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` on page 3-0) 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` $\leq m \leq$ `ModulationOrder–1`) maps to the complex value **exp**($j\times$ `PhaseOffset` on page 3-0 $+ j\times 2\times \pi \times m$/`ModulationOrder`). When you set this property to `Custom`, the object uses the signal constellation defined in the CustomSymbolMapping on page 3-0 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 on page 3-0 . 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 on page 3-0 , 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 on page 3-0 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 on page 3-0 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 on page 3-0 property to true and the DecisionMethod on page 3-0 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` on page 3-0 property to true, the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio`, or `Approximate log-likelihood ratio`, and the `VarianceSource` on page 3-0 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` on page 3-0 property to `false`. It also applies when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 property to `Hard decision`. In this second case, when the `OutputDataType` on page 3-0 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` on page 3-0 property to `false. It also applies` when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 property to `Hard decision`. The object uses the derotate factor in the computations only when the `ModulationOrder` on page 3-0 property is `2`, `4`, or `8`. The `step` method input must also have a fixed-point type, and the `PhaseOffset` on page 3-0 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` on page 3-0 property to `Custom`. The word length must be a value between 2 and 128.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using M-ary PSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

16-PSK with Custom Symbol Mapping

Create 16-PSK modulator and demodulator System objects™ in which custom symbol mapping is used. Estimate the BER in an AWGN channel and compare the performance with that of a theoretical Gray-coded PSK system.

Create a custom symbol mapping for the 16-PSK modulation scheme. The 16 integer symbols must have values which fall between 0 and 15.

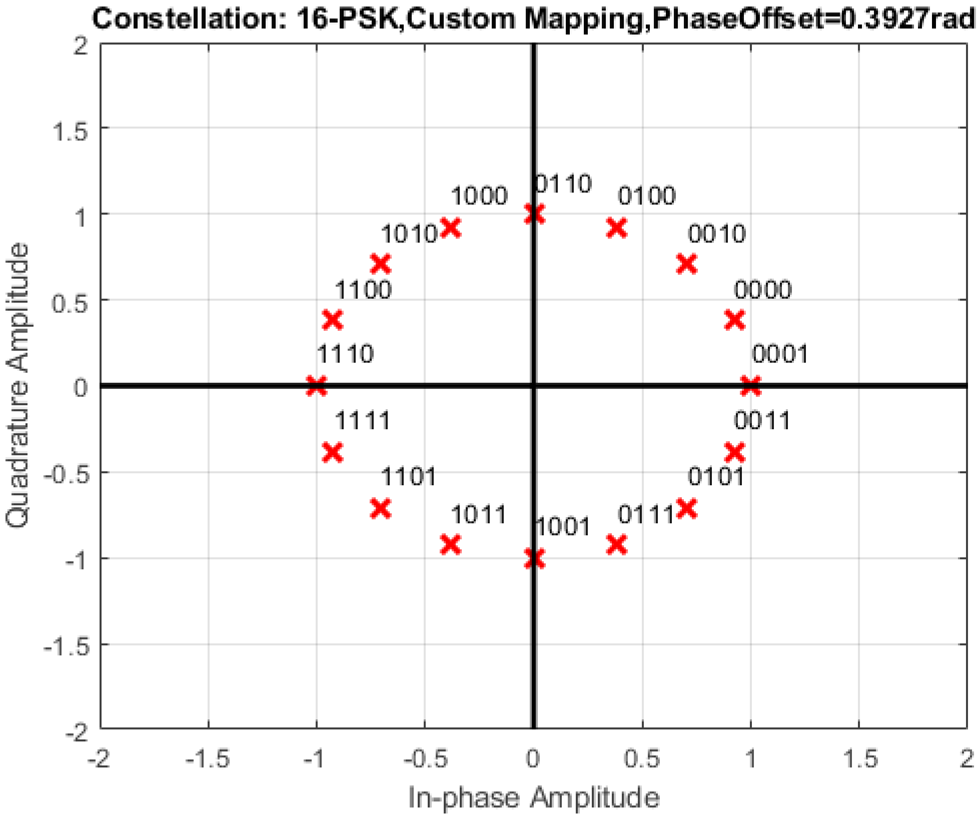
```
custMap = [0 2 4 6 8 10 12 14 15 13 11 9 7 5 3 1];
```

Create a 16-PSK modulator and demodulator pair having custom symbol mapping defined by the array, `custMap`.

```
pskModulator = comm.PSKModulator(16,'BitInput',true, ...
    'SymbolMapping','Custom', ...
    'CustomSymbolMapping',custMap);
pskDemodulator = comm.PSKDemodulator(16,'BitOutput',true, ...
    'SymbolMapping','Custom', ...
    'CustomSymbolMapping',custMap);
```

Display the modulator constellation.

```
constellation(pskModulator)
```



Create an AWGN channel System object for use with 16-ary data.

```
awgnChannel = comm.AWGNChannel('BitsPerSymbol',log2(16));
```

Create an error rate object to track the BER statistics.

```
errorRate = comm.ErrorRate;
```

Initialize the simulation vectors. The Eb/No is varied from 6 to 18 dB in 1 dB steps.

```
ebnoVec = 6:18;
ber = zeros(size(ebnoVec));
```

Estimate the BER by modulating binary data, passing it through an AWGN channel, demodulating the received signal, and collecting the error statistics.

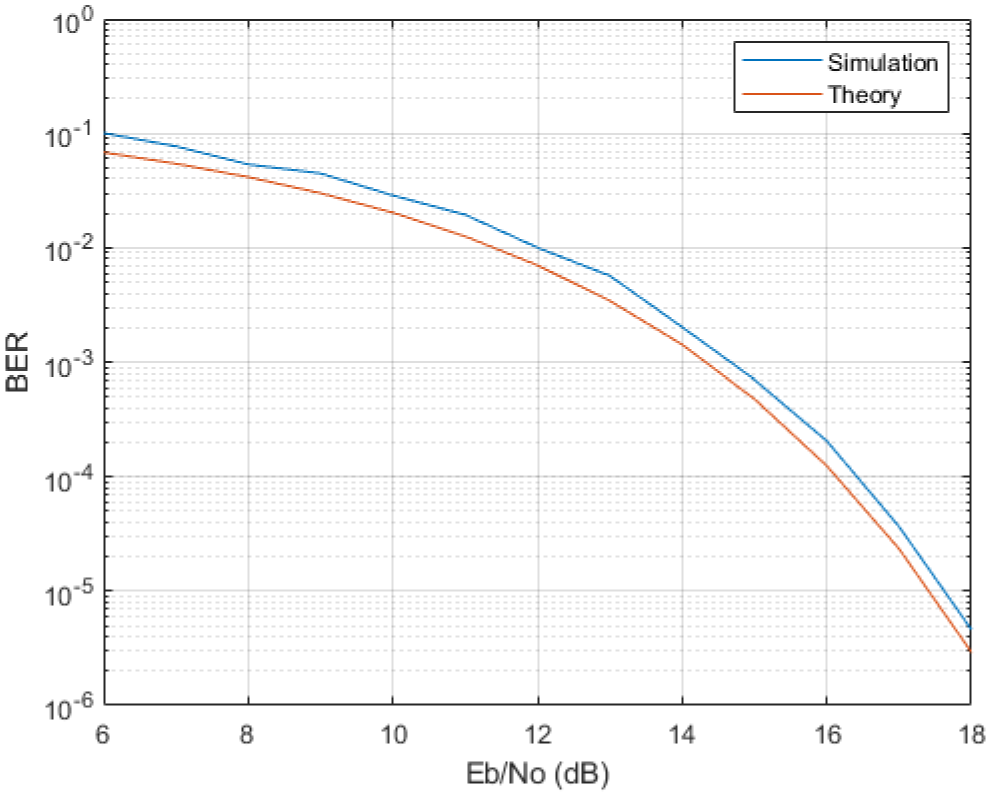
```
for k = 1:length(ebnoVec)

    % Reset the error counter for each Eb/No value
    reset(errorRate)
    % Reset the array used to collect the error statistics
    errVec = [0 0 0];
    % Set the channel Eb/No
    awgnChannel.EbNo = ebnoVec(k);

    while errVec(2) < 200 && errVec(3) < 1e7
        % Generate a 1000-symbol frame
        data = randi([0 1],4000,1);
        % Modulate the binary data
        modData = pskModulator(data);
        % Pass the modulated data through the AWGN channel
        rxSig = awgnChannel(modData);
        % Demodulate the received signal
        rxData = pskDemodulator(rxSig);
        % Collect the error statistics
        errVec = errorRate(data,rxData);
    end

    % Save the BER data
    ber(k) = errVec(1);
end
```

Generate theoretical BER data for an AWGN channel using `berawgn`.

```
berTheory = berawgn(ebnoVec,'psk',16,'nondiff');
```

Plot the simulated and theoretical results. Because the simulated results rely on 16-PSK modulation that does not use Gray codes, the performance is not as good as that predicted by theory.

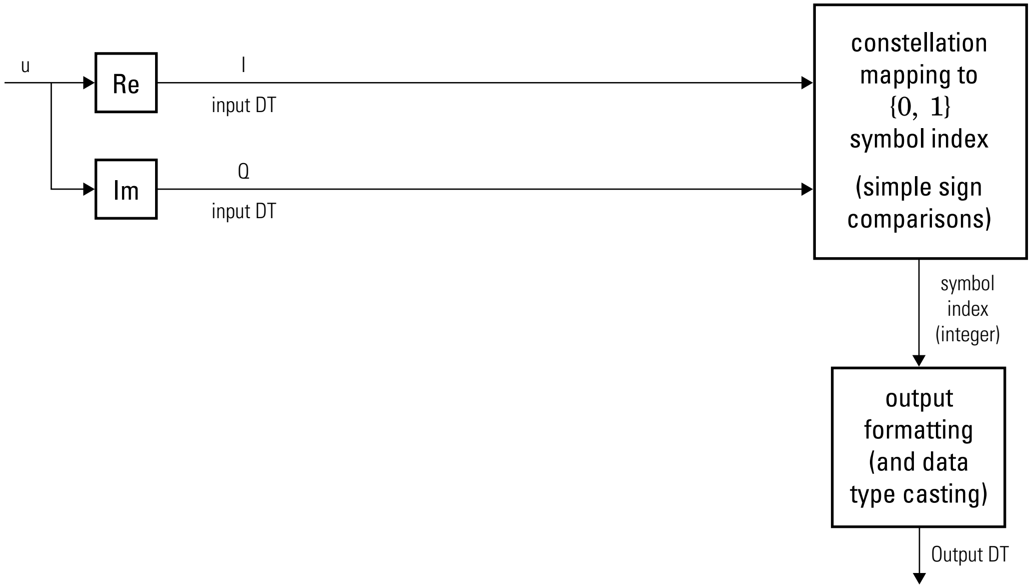
```
figure
semilogy(ebnoVec,[ber; berTheory])
xlabel('Eb/No (dB)')
ylabel('BER')
grid
legend('Simulation','Theory','location','ne')
```



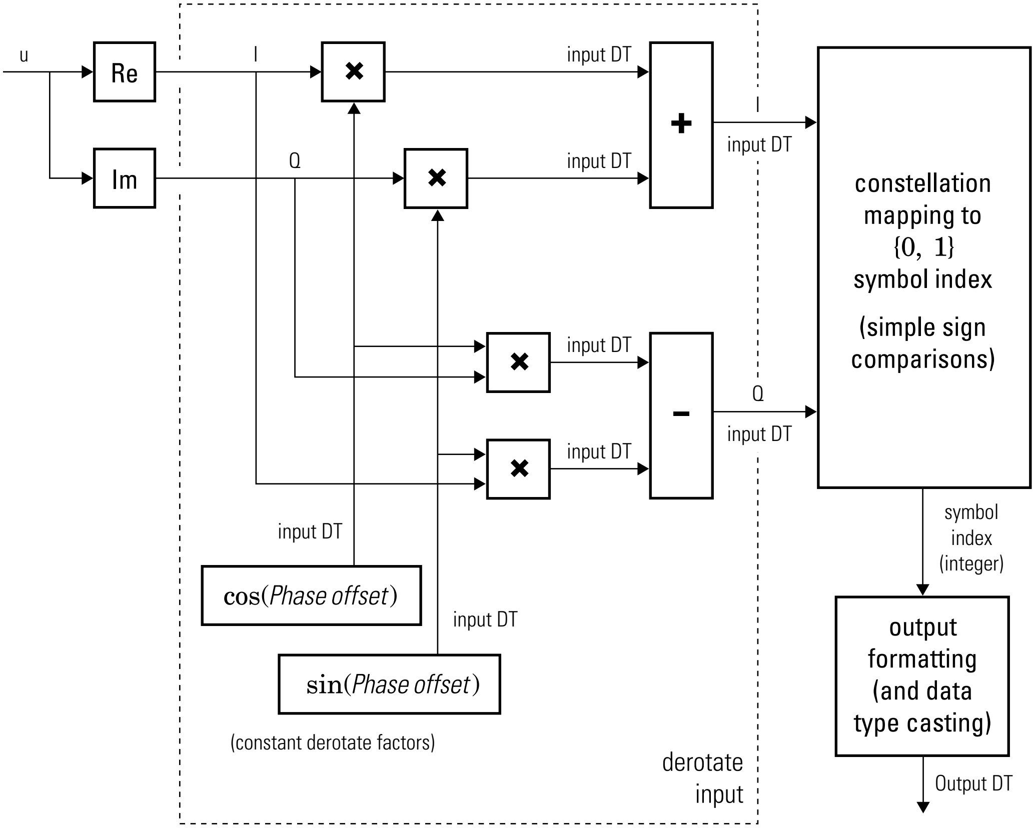
Algorithms

BPSK

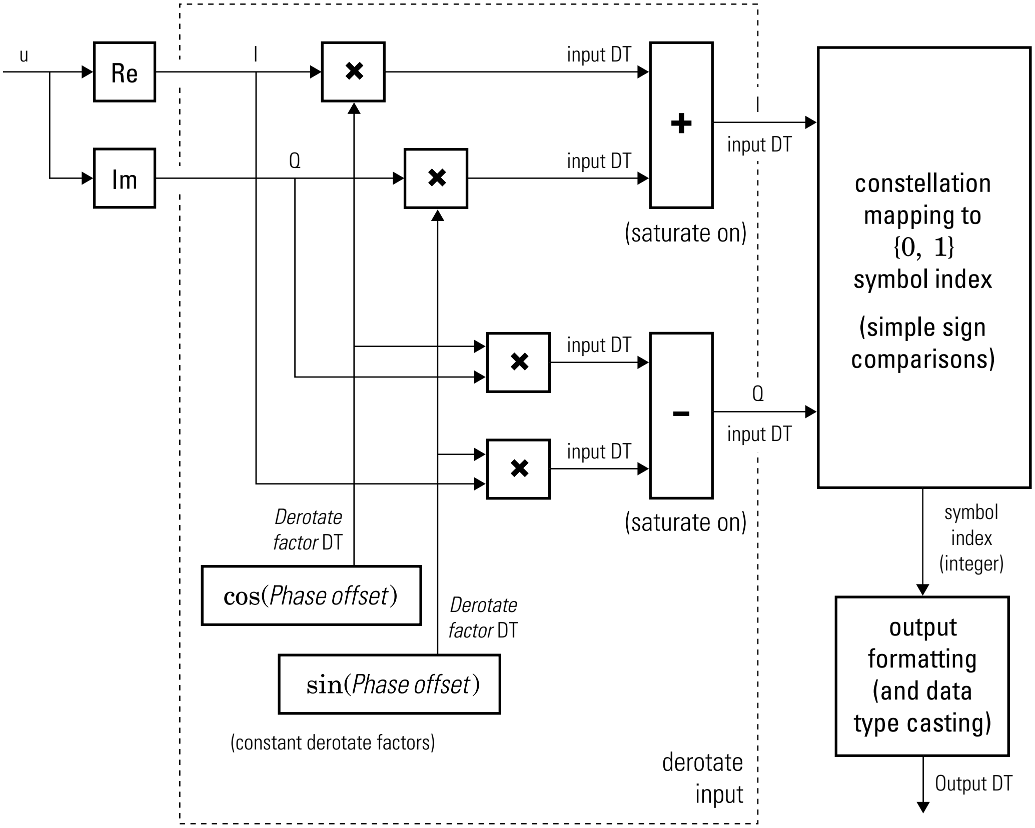
Diagrams for hard-decision demodulation of BPSK signals follow.


u
Re
I
input DT
Im
Q
input DT
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)


u
Re
Im
I
Q
×
input DT
+
I
input DT
Q
input DT
−
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
cos(Phase offset)
sin(Phase offset)
(constant derotate factors)
derotate input

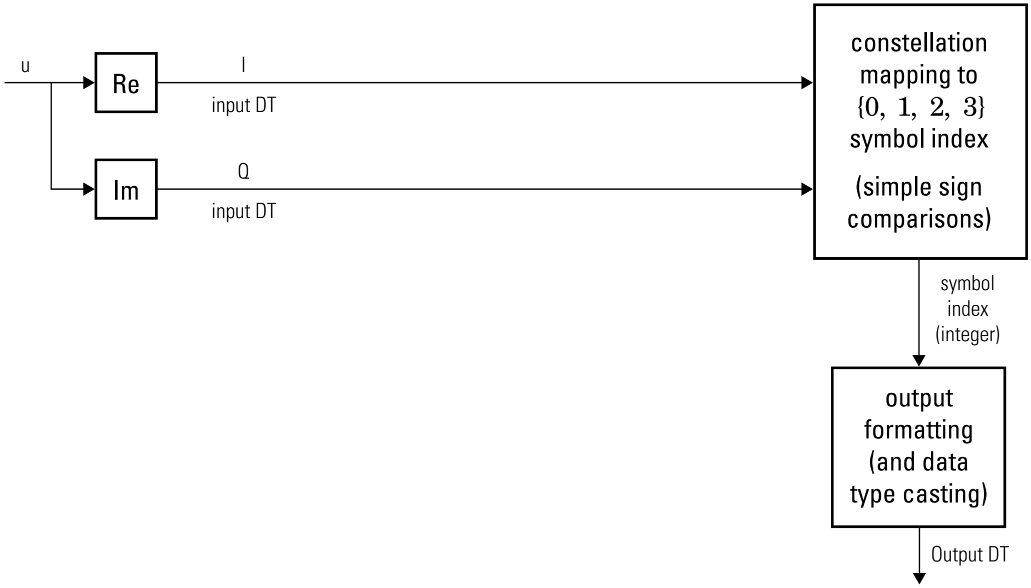
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



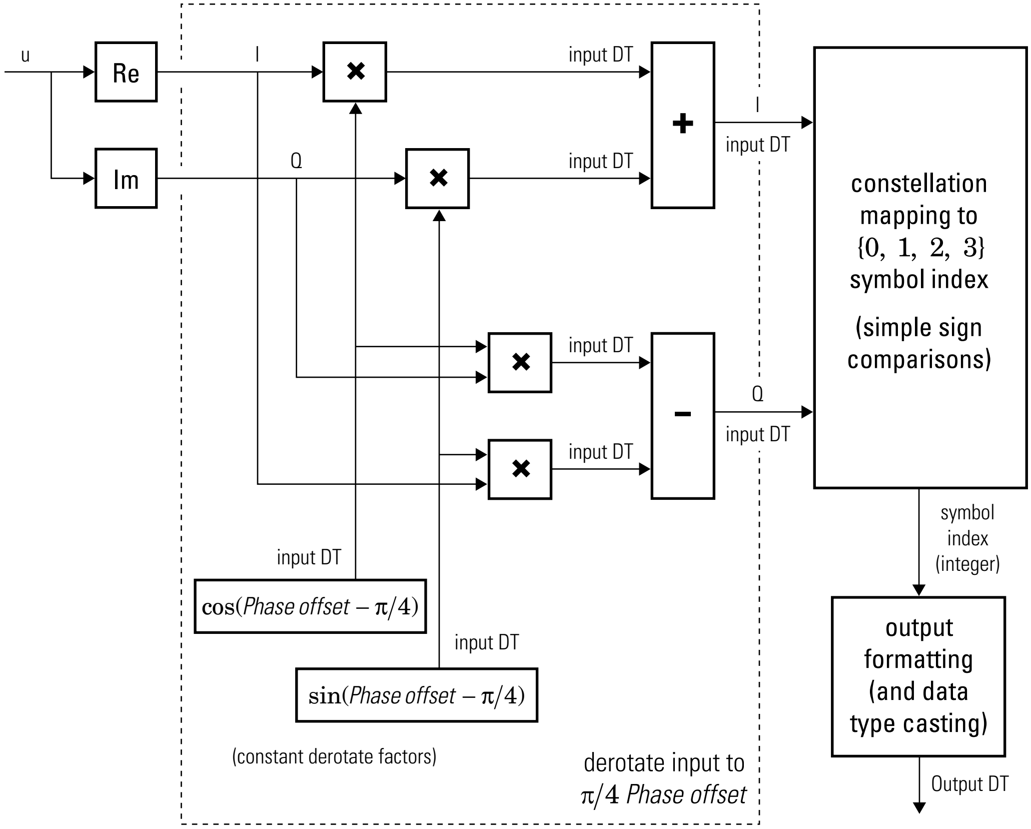
Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

QPSK

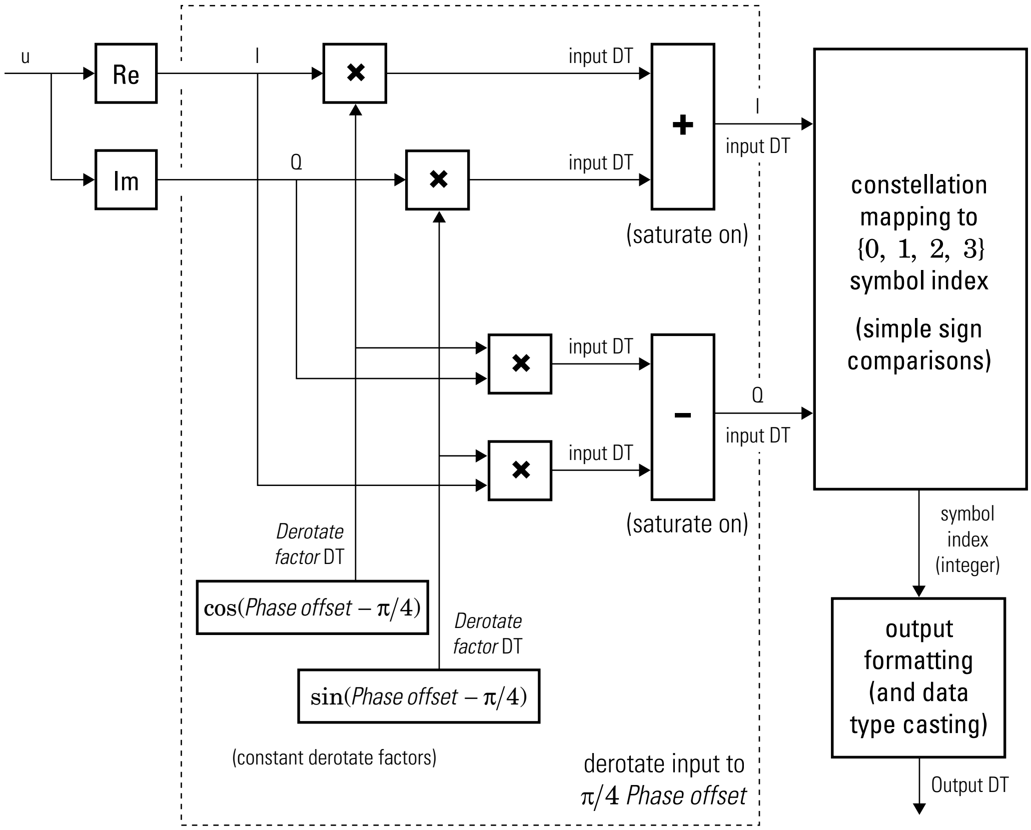
Diagrams for hard-decision demodulation of QPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to $\{0,\ 1,\ 2,\ 3\}$ symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of $\frac{\pi}{4}$)


u
Re
Im
I
Q
input DT
input DT
+
I
input DT
input DT
input DT
−
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
input DT
cos(Phase offset − π/4)
input DT
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset
output formatting (and data type casting)
Output DT

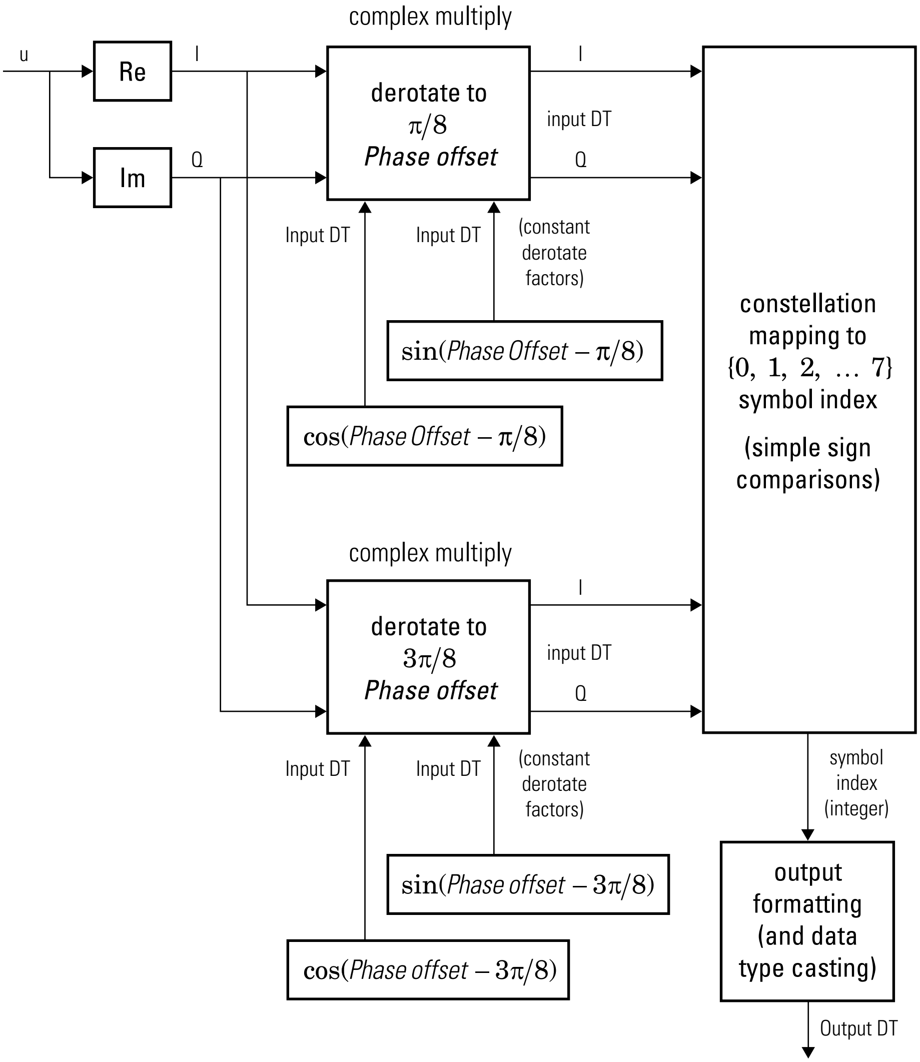
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



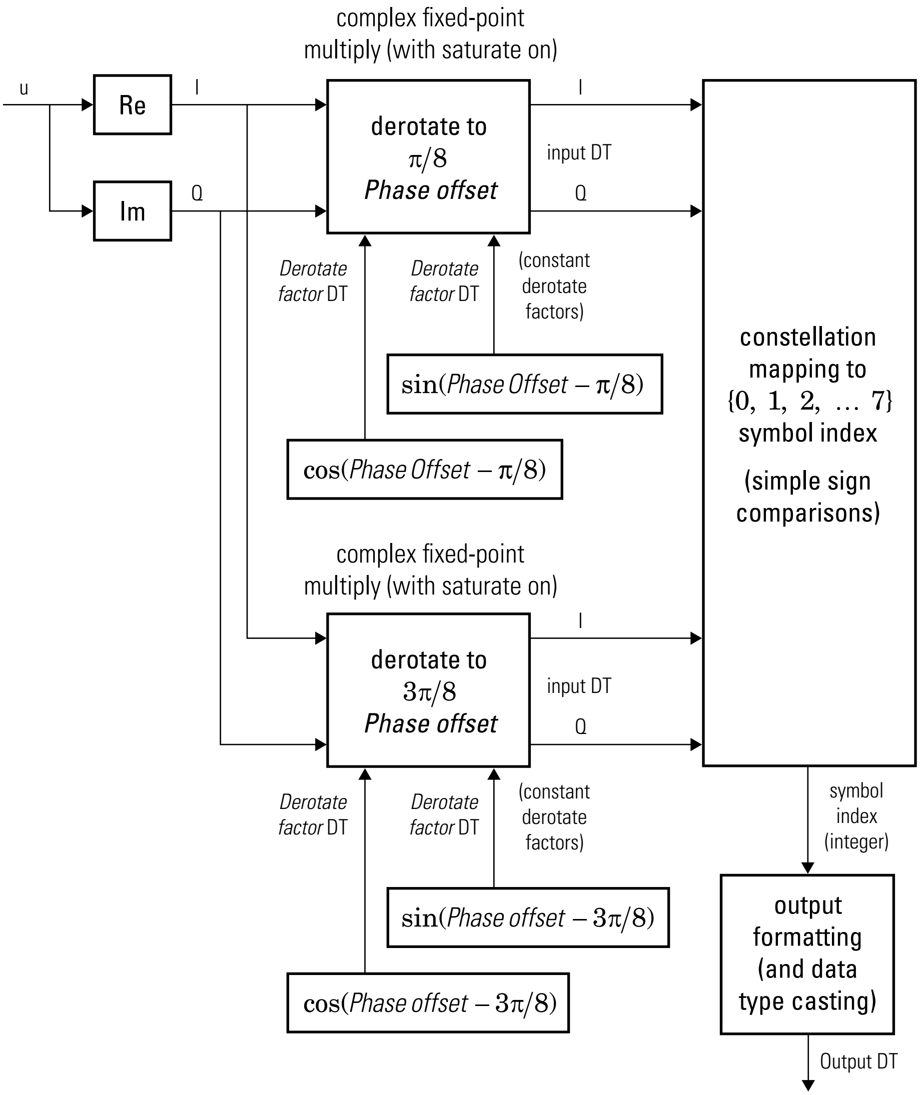
Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

Higher-Order PSK

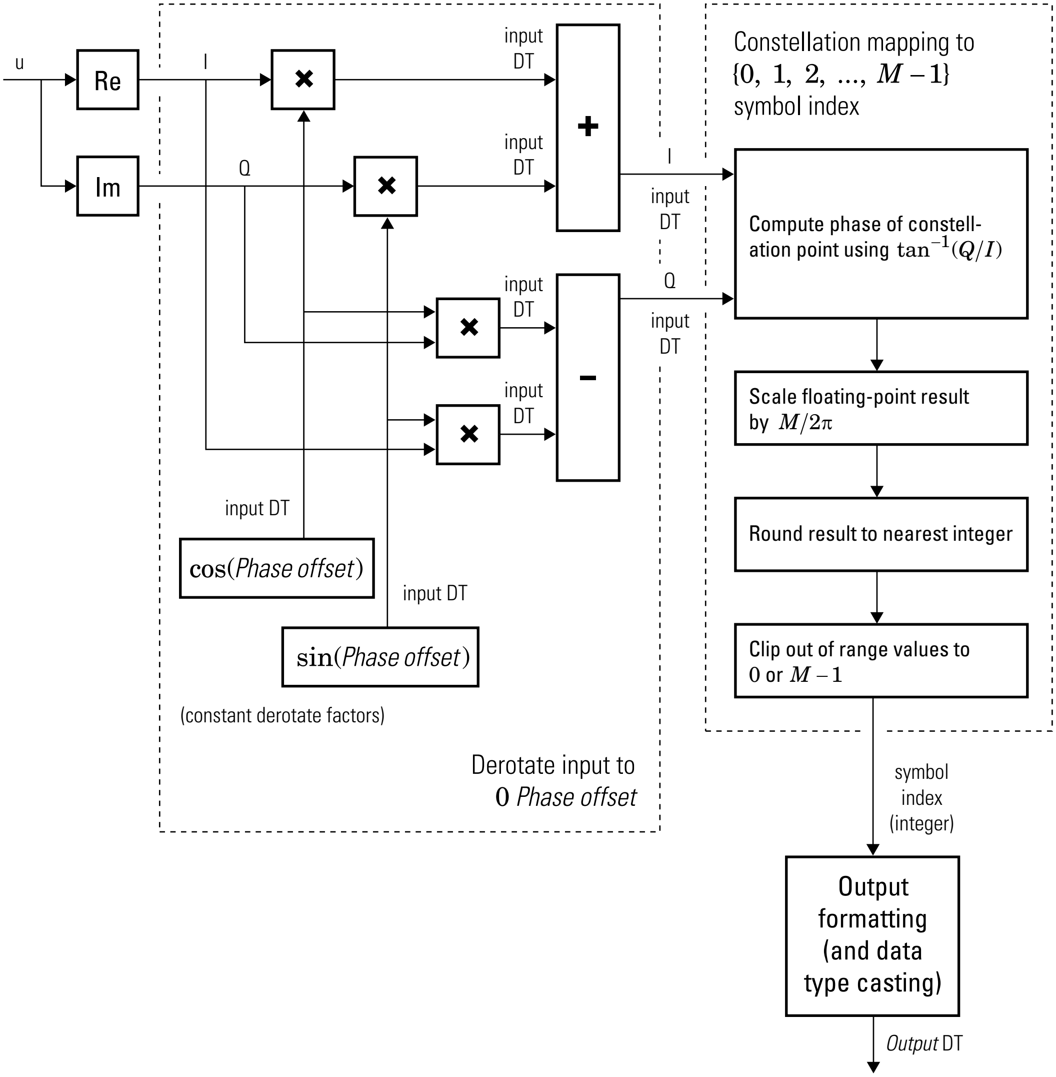
Diagrams for hard-decision demodulation of higher-order ($M \geq 8$) signals follow.



Hard-Decision 8-PSK Demodulator Floating-Point Signal Diagram


complex fixed-point
multiply (with saturate on)
u
Re
Im
I
Q
derotate to
$\pi/8$
Phase offset
I
input DT
Q
Derotate factor DT
Derotate factor DT
(constant derotate factors)
$\sin($*Phase Offset* $- \pi/8)$
$\cos($*Phase Offset* $- \pi/8)$
constellation mapping to
$\{0, 1, 2, \ldots 7\}$
symbol index
(simple sign comparisons)
complex fixed-point
multiply (with saturate on)
derotate to
$3\pi/8$
Phase offset
I
input DT
Q
Derotate factor DT
Derotate factor DT
(constant derotate factors)
$\sin($*Phase offset* $- 3\pi/8)$
$\cos($*Phase offset* $- 3\pi/8)$
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision 8-PSK Demodulator Fixed-Point Signal Diagram


u
Re
Im
I
Q
×
+
−
input DT
cos(*Phase offset*)
sin(*Phase offset*)
(constant derotate factors)
Derotate input to 0 *Phase offset*
I
Q
Constellation mapping to $\{0, 1, 2, ..., M-1\}$ symbol index
Compute phase of constellation point using $\tan^{-1}(Q/I)$
Scale floating-point result by $M/2\pi$
Round result to nearest integer
Clip out of range values to 0 or $M-1$
symbol index (integer)
Output formatting (and data type casting)
Output DT

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 `PhaseOffset` 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`.

Log-likelihood Ratio and Approximate Log-likelihood Ratio

The exact LLR and approximate LLR algorithms (soft-decision) are described in “Phase Modulation”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DPSKDemodulator` | `comm.PSKModulator`

Introduced in R2012a

constellation

System object: comm.PSKDemodulator
Package: comm

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

Plot PSK Reference Constellation

Create a PSK modulator.

```
mod = comm.PSKModulator;
```

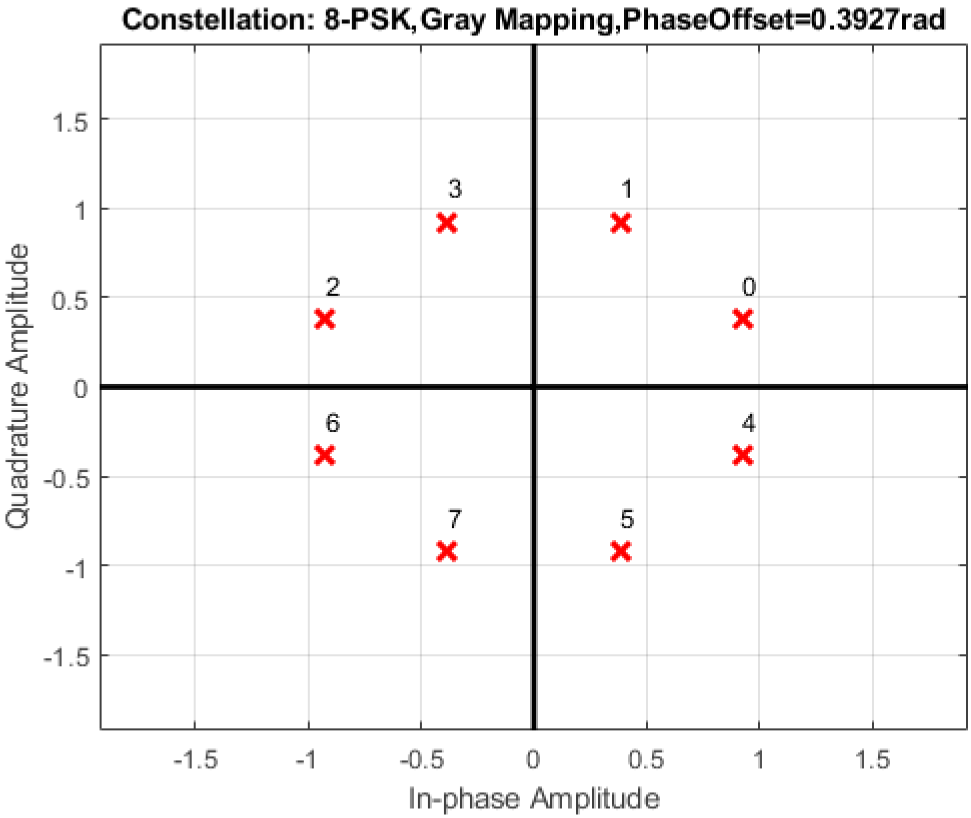
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 8×1 complex

   0.9239 + 0.3827i
   0.3827 + 0.9239i
  -0.3827 + 0.9239i
  -0.9239 + 0.3827i
  -0.9239 - 0.3827i
  -0.3827 - 0.9239i
   0.3827 - 0.9239i
   0.9239 - 0.3827i
```

Plot the constellation.

```
constellation(mod)
```

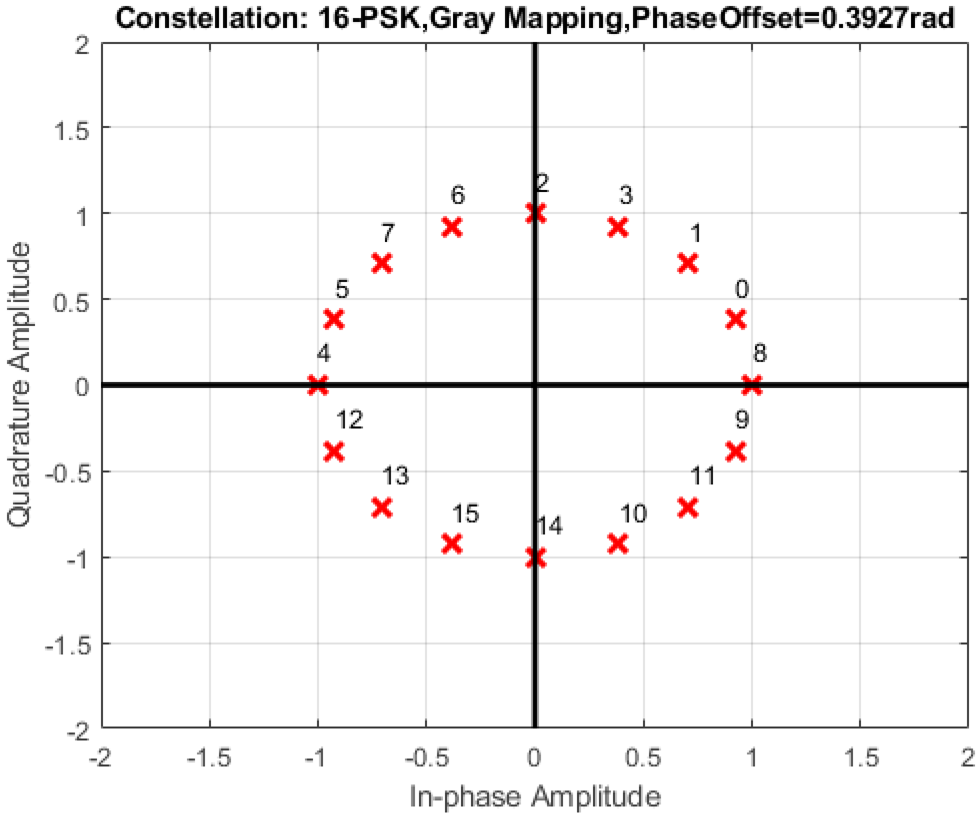


Create a PSK demodulator having modulation order 16.

```
demod = comm.PSKDemodulator(16);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



step

System object: comm.PSKDemodulator
Package: comm

Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PSKModulator System object

Package: comm

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.

To modulate a signal using phase shift keying:

1. Define and set up your PSK modulator object. See “Construction” on page 3-1225.
2. Call step to modulate the signal according to the properties of comm.PSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0). This vector contains bit representations of integers between 0 and ModulationOrder-1. When you set the BitInput on page 3-0 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 on page 3-0) 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 on page 3-0 $+ j\times 2\times \pi \times m/$ModulationOrder). When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping on page 3-0 property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. This property requires a row or column vector of size `ModulationOrder` on page 3-0 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` on page 3-0 , 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` on page 3-0 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` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using M-ary PSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Add White Gaussian Noise to 8-PSK Signal

Modulate an 8-PSK signal, add white Gaussian noise, and plot the signal to observe the effects of noise.

Create a PSK modulator System object™. The default modulation order for the PSK modulator object is 8.

```
pskModulator = comm.PSKModulator;
```

Modulate the signal.

```
modData = pskModulator(randi([0 7],2000,1));
```

Add white Gaussian noise to the modulated signal by passing the signal through an AWGN channel.

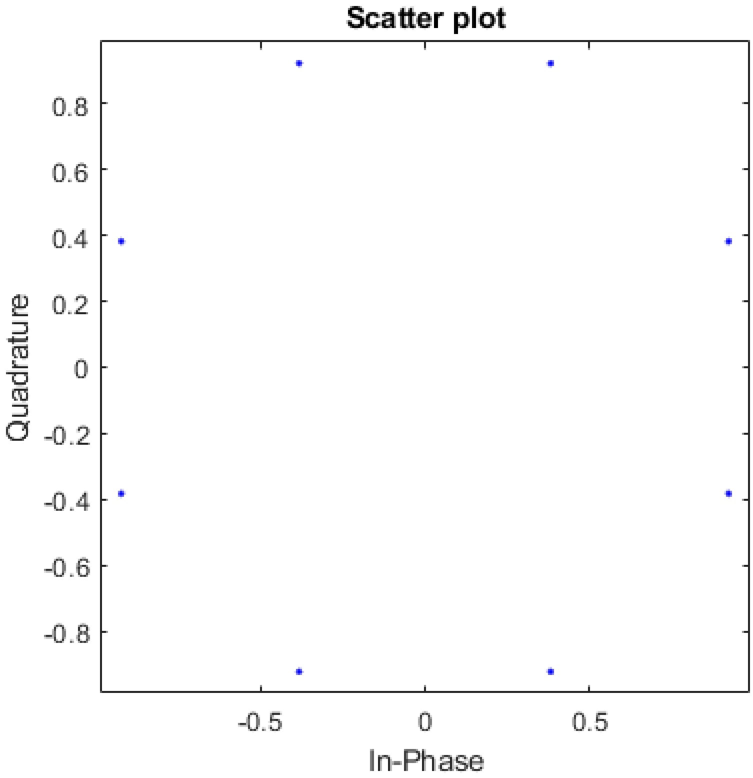
```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',3);
```

Transmit the signal through the AWGN channel.

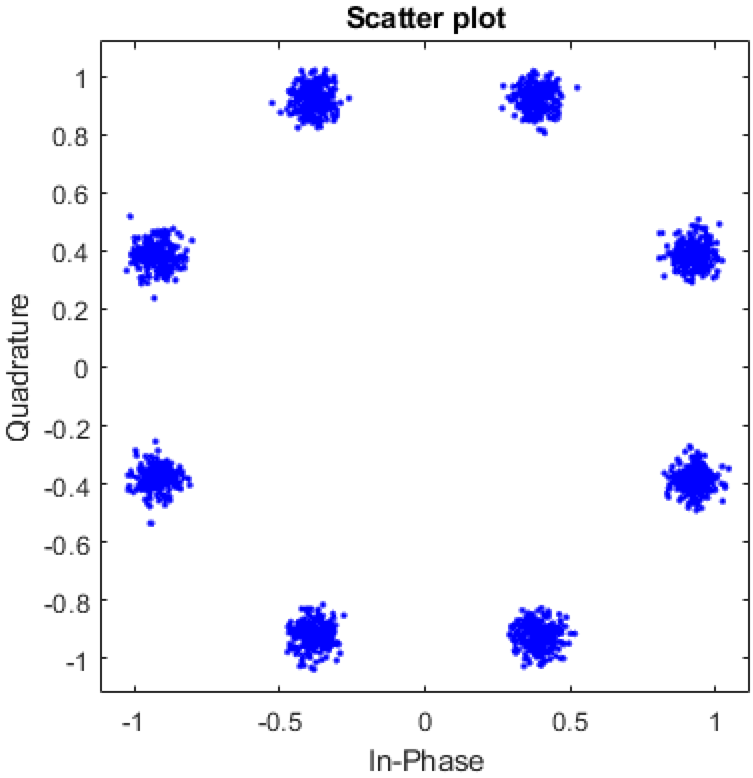
```
channelOutput = channel(modData);
```

Plot the noiseless and noisy data using scatter plots to observe the effects of noise.

```
scatterplot(modData)
```



```
scatterplot(channelOutput)
```



Change the EbNo property to 10 dB to increase the noise.

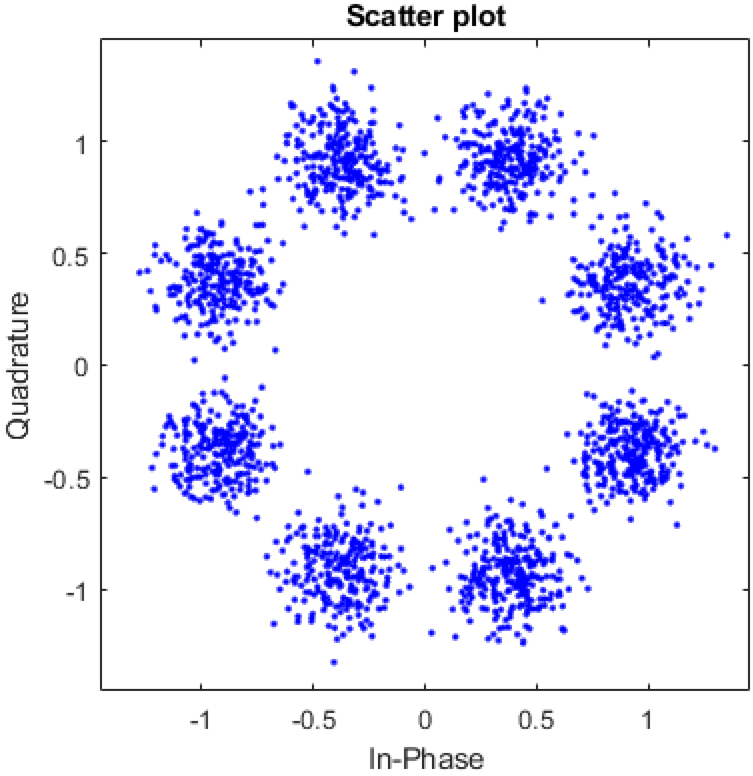
```
channel.EbNo = 10;
```

Pass the modulated data through the AWGN channel.

```
channelOutput = channel(modData);
```

Plot the channel output. You can see the effects of increased noise.

```
scatterplot(channelOutput)
```



Algorithms

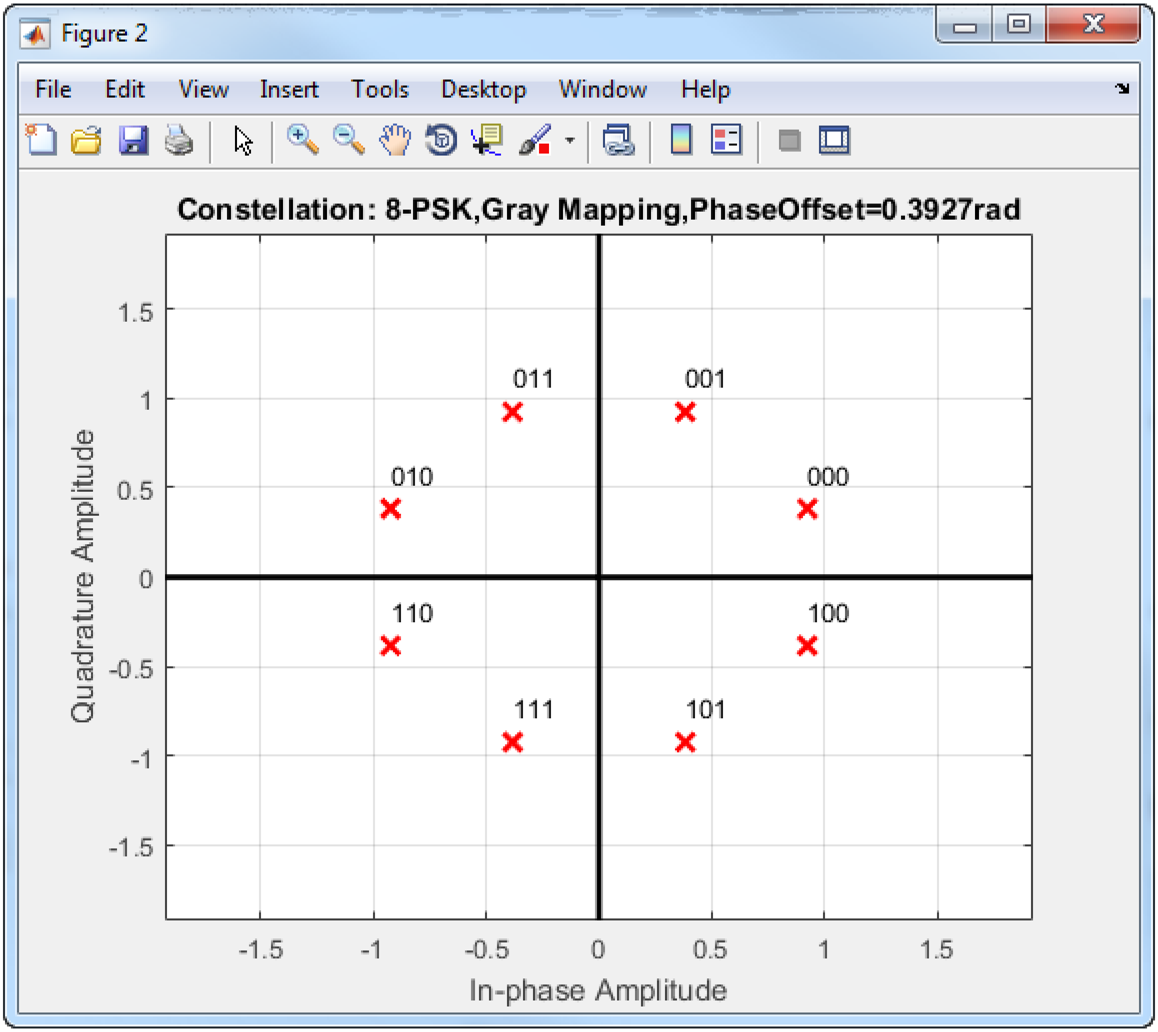
The block outputs a baseband signal by mapping input bits or integers to complex symbols according to the following:

$$s_n(t) = \exp\left(j\pi\left(\frac{2n+1}{M}\right)\right); \quad n \in \{0,1,\ldots,M-1\}.$$

This applies when a natural binary ordering is used. Another common mapping is Gray coding, which has the advantage that only one bit changes between adjacent constellation points. This results in better bit error rate performance. For 8-PSK modulation with Gray coding, the mapping between the input and output symbols is shown.

Input	Output
0	0 *(000)*
1	1 *(001)*
2	3 *(011)*
3	2 *(010)*
4	6 *(110)*
5	7 *(111)*
6	5 *(101)*
7	4 *(100)*

The corresponding constellation diagram follows.



When the input signal is composed of bits, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $\log_2(M)$ bits.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PSKDemodulator` | `comm.QPSKModulator`

Introduced in R2012a

constellation

System object: comm.PSKModulator
Package: comm

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

Plot PSK Reference Constellation

Create a PSK modulator.

```
mod = comm.PSKModulator;
```

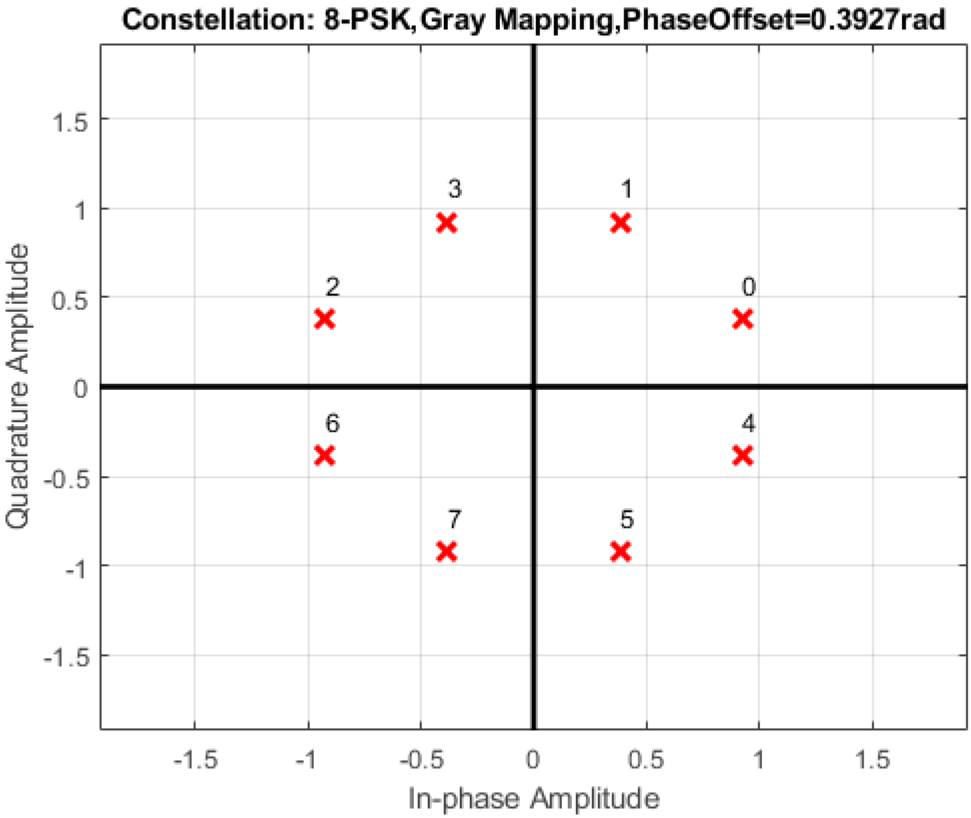
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 8×1 complex

   0.9239 + 0.3827i
   0.3827 + 0.9239i
  -0.3827 + 0.9239i
  -0.9239 + 0.3827i
  -0.9239 - 0.3827i
  -0.3827 - 0.9239i
   0.3827 - 0.9239i
   0.9239 - 0.3827i
```

Plot the constellation.

```
constellation(mod)
```

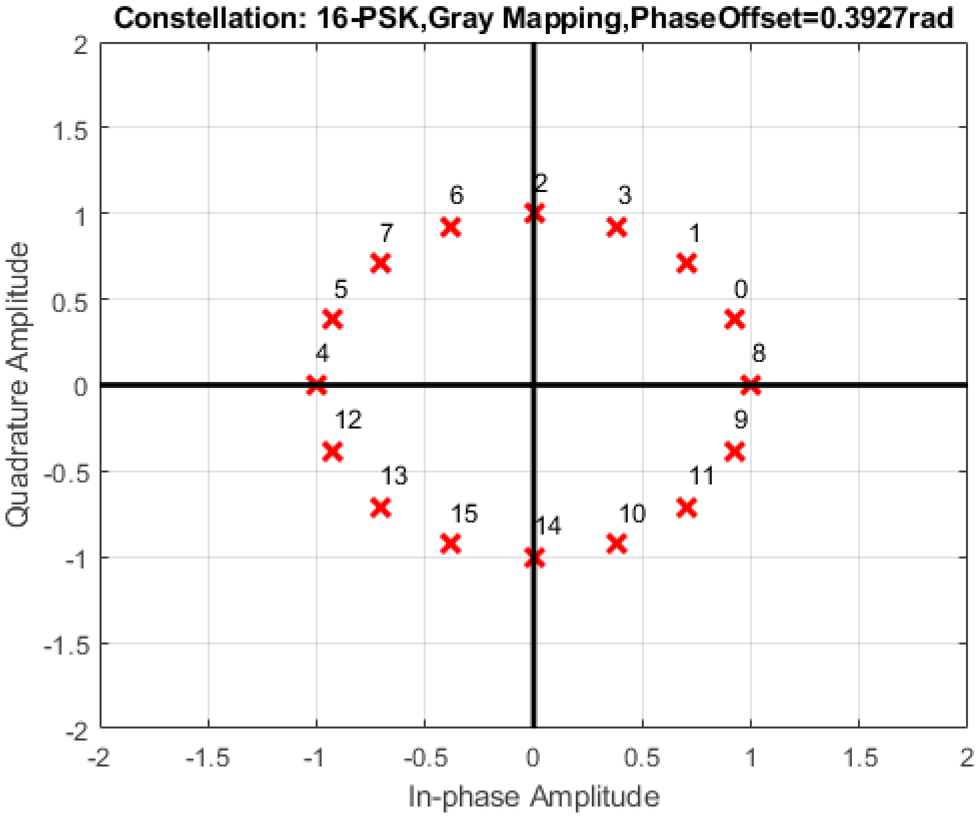


Create a PSK demodulator having modulation order 16.

```
demod = comm.PSKDemodulator(16);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



step

System object: comm.PSKModulator
Package: comm

Modulate using M-ary PSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Demodulate convolutionally encoded data mapped to M-ary PSK signal 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.

To demodulate a signal that was modulated using trellis-coded modulation:

1. Define and set up your PSK TCM demodulator object. See “Construction” on page 3-1239.
2. Call `step` to demodulate the signal according to the properties of `comm.PSKTCMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 property to `Continuous`, the decoding delay consists of TracebackDepth zero symbols or TracebackDepth × 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` on page 3-0 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` on page 3-0 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

reset	Reset states of the PSK TCM demodulator object
step	Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate Noisy PSK QAM Data

Modulate and demodulate data using 8-PSK TCM modulation in an AWGN channel. Estimate the resulting error rate.

Define a trellis structure with four input symbols and eight output symbols.

```
t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
```

Create modulator and demodulator System objects™ using trellis, `t`, having modulation order 8.

```
hMod = comm.PSKTCMModulator(t,'ModulationOrder',8);
hDemod = comm.PSKTCMDemodulator(t,'ModulationOrder',8, ...
    'TracebackDepth',16);
```

Create an AWGN channel object.

```
hAWGN = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)', ...
    'SNR',7);
```

Create an error rate calculator with delay in bits equal to `TracebackDepth` times the number of bits per symbol.

```
hError = comm.ErrorRate('ReceiveDelay',...
    hDemod.TracebackDepth*log2(t.numInputSymbols));
```

Generate random binary data and modulate with 8-PSK TCM. Pass the modulated signal through the AWGN channel and demodulate. Calculate the error statistics.

```
for counter = 1:10
    % Transmit frames of 250 2-bit symbols
    data = randi([0 1],500,1);
    % Modulate
    modSignal = step(hMod,data);
    % Pass through AWGN channel
    noisySignal = step(hAWGN,modSignal);
    % Demodulate
    receivedData = step(hDemod,noisySignal);
    % Calculate error statistics
    errorStats = step(hError,data,receivedData);
end
```

Display the BER and the number of bit errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1),errorStats(2))
```

```
Error rate = 2.17e-02
Number of errors = 108
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

comm.GeneralQAMTCMDemodulator | comm.PSKTCMModulator | comm.RectangularQAMTCMDemodulator | comm.ViterbiDecoder

Introduced in R2012a

reset

System object: comm.PSKTCMDemodulator
Package: comm

Reset states of the PSK TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `PSKTCMDemodulator` object, `H`.

step

System object: comm.PSKTCMDemodulator
Package: comm

Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.PSKTCMModulator System object

Package: comm

Convolutionally encode binary data and map using M-ary PSK signal 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.

To modulate a signal using trellis-coded modulation:

1. Define and set up your PSK TCM modulator object. See “Construction” on page 3-1247.
2. Call `step` to modulate the signal according to the properties of `comm.PSKTCMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

reset	Reset states of the PSK TCM modulator object
step	Convolutionally encode binary data and map using M-ary PSK constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data Using 8-PSK TCM Modulation

Modulate random data using 8-PSK TCM modulation and display the constellation diagram.

Create binary data.

```
data = randi([0 1],1000,1);
```

Define a trellis structure with four input symbols and eight output symbols.

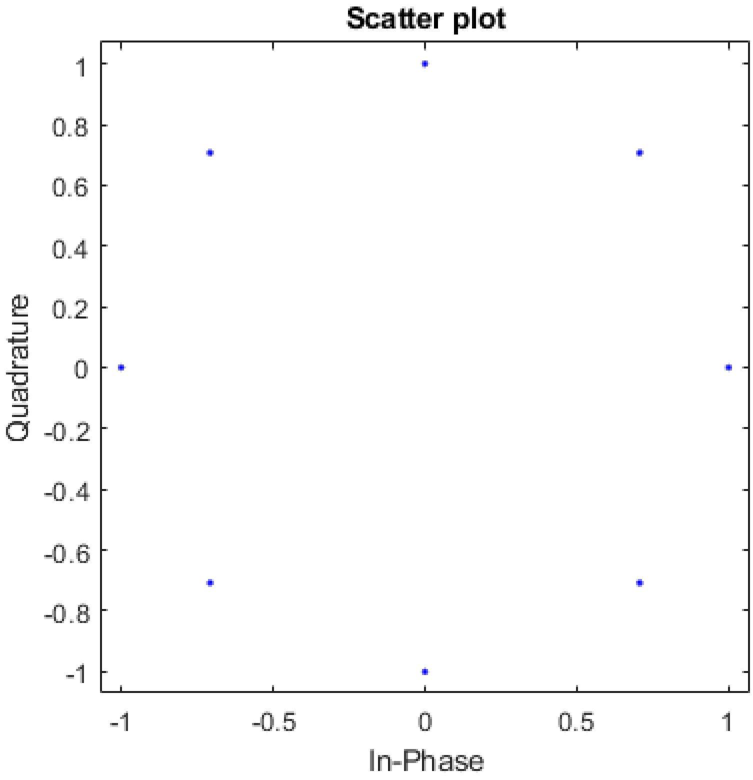
```
t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
```

Create an 8-PSK TCM modulator object using the trellis structure variable, `t`.

```
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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMTCMModulator` | `comm.PSKTCMDemodulator` | `comm.RectangularQAMTCMModulator`

Introduced in R2012a

reset

System object: comm.PSKTCMModulator
Package: comm

Reset states of the PSK TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `PSKTCMModulator` object, `H`.

step

System object: comm.PSKTCMModulator
Package: comm

Convolutionally encode binary data and map using M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.QAMCoarseFrequencyEstimator System object

Package: comm

Estimate frequency offset for QAM signal

Description

The `QAMCoarseFrequencyEstimator` System object estimates frequency offset for a QAM signal.

To estimate frequency offset for a QAM signal:

1. Define and set up your QAM Coarse Frequency Estimator object. See “Construction” on page 3-1255.
2. Call `step` to estimate frequency offset for a QAM signal according to the properties of `comm.QAMCoarseFrequencyEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.QAMCoarseFrequencyEstimator` creates a rectangular QAM coarse frequency offset estimator object, `H`. This object uses an open-loop, FFT-based technique to estimate the carrier frequency offset in a received rectangular QAM signal.

`H = comm.QAMCoarseFrequencyEstimator(Name,Value)` creates a rectangular QAM coarse frequency offset estimator 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

FrequencyResolution

Desired frequency resolution (Hz)

Specify the desired frequency resolution for offset frequency estimation as a positive, real scalar of data type double. This property establishes the FFT length that the object uses to perform spectral analysis. The value for this property must be less than or equal to half the `SampleRate` on page 3-0 property. The default is `0.001`.

SampleRate

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type double. The default is `1`.

Methods

reset	Reset states of the `QAMCoarseFrequencyEstimator` object
step	Estimate frequency offset for QAM signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Correct Frequency Offset in 16-QAM Signal

Estimate and correct for a -250 Hz frequency offset in a 16-QAM signal using the QAM Coarse Frequency Estimator System object™.

Create a rectangular QAM modulator System object using name-value pairs to set the modulation order to 16 and the constellation to have an average power of 1 W.

```
qamModulator = comm.RectangularQAMModulator('ModulationOrder',16, ...
    'NormalizationMethod','Average power', ...
    'AveragePower',1);
```

Create a square root raised cosine transmit filter System object.

```
txfilter = comm.RaisedCosineTransmitFilter;
```

Create a phase frequency offset object, where the `FrequencyOffset` property is set to -250 Hz and `SampleRate` is set to 4000 Hz using name-value pairs.

```
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',-250, ...
    'SampleRate',4000);
```

Create a QAM coarse frequency estimator System object with a sample rate of 4 kHz and a frequency resolution of 1 Hz.

```
frequencyEst = comm.QAMCoarseFrequencyEstimator(...
    'SampleRate',4000, ...
    'FrequencyResolution',1);
```

Create a second phase frequency offset object to correct the offset. Set the `FrequencyOffsetSource` property to `Input port` so that the frequency correction estimate is an input argument.

```
pfoCorrect = comm.PhaseFrequencyOffset(...
    'FrequencyOffsetSource','Input port', ...
    'SampleRate',4000);
```

Create a spectrum analyzer object to view the frequency response of the signals.

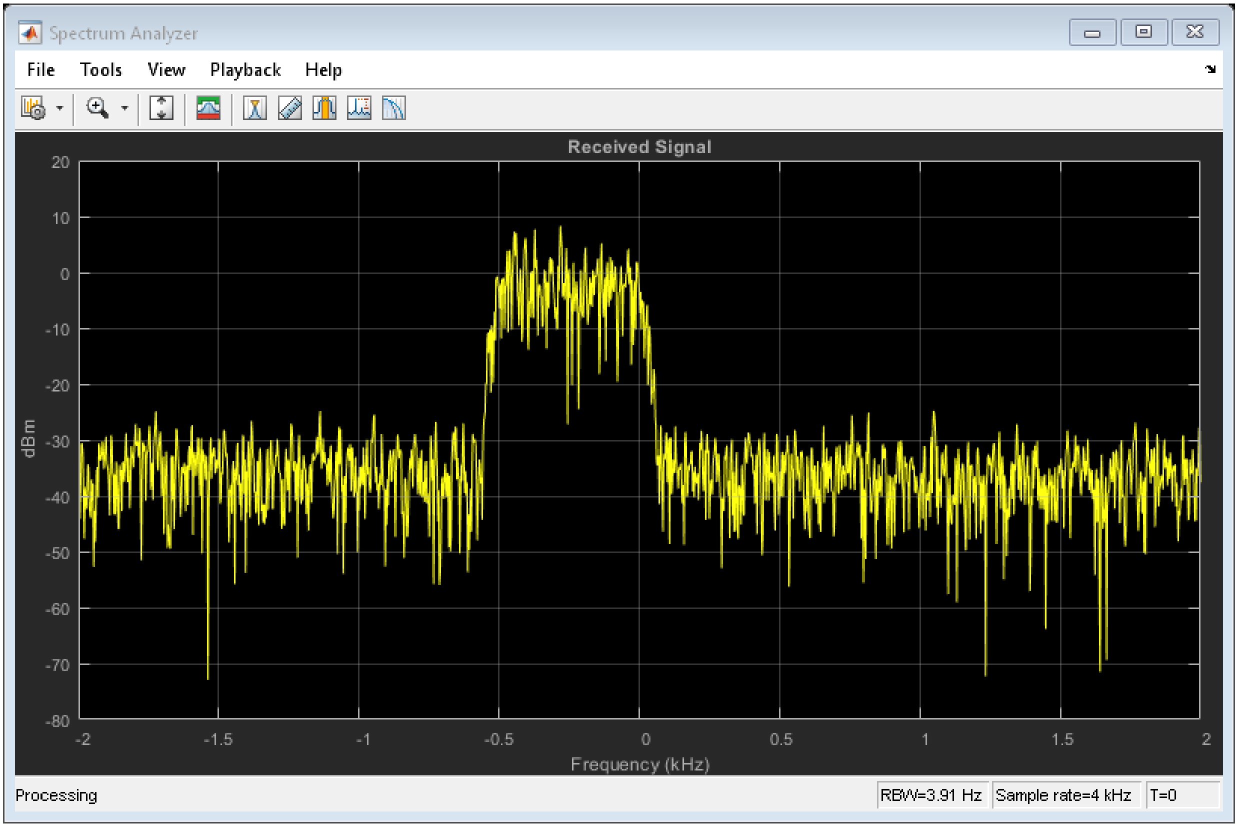
```
spectrum = dsp.SpectrumAnalyzer('SampleRate',4000);
```

Generate a 16-QAM signal, filter the signal, apply the frequency offset, and pass the signal through the AWGN channel.

```
modData = qamModulator(randi([0 15],4096,1));    % Generate QAM signal
txFiltData = txfilter(modData);                  % Apply Tx filter
offsetData = pfo(txFiltData);                    % Apply frequency offset
noisyData = awgn(offsetData,25,'measured');      % Pass through AWGN channel
```

Plot the frequency response of the noisy, frequency-offset signal using the spectrum analyzer. The signal is shifted 250 Hz to the left.

```
spectrum.Title = 'Received Signal';
spectrum(noisyData);
```



Estimate the frequency offset using `frequencyEst`. Observe that the estimate is close to the -250 Hz target.

```
estFreqOffset = frequencyEst(noisyData)
```

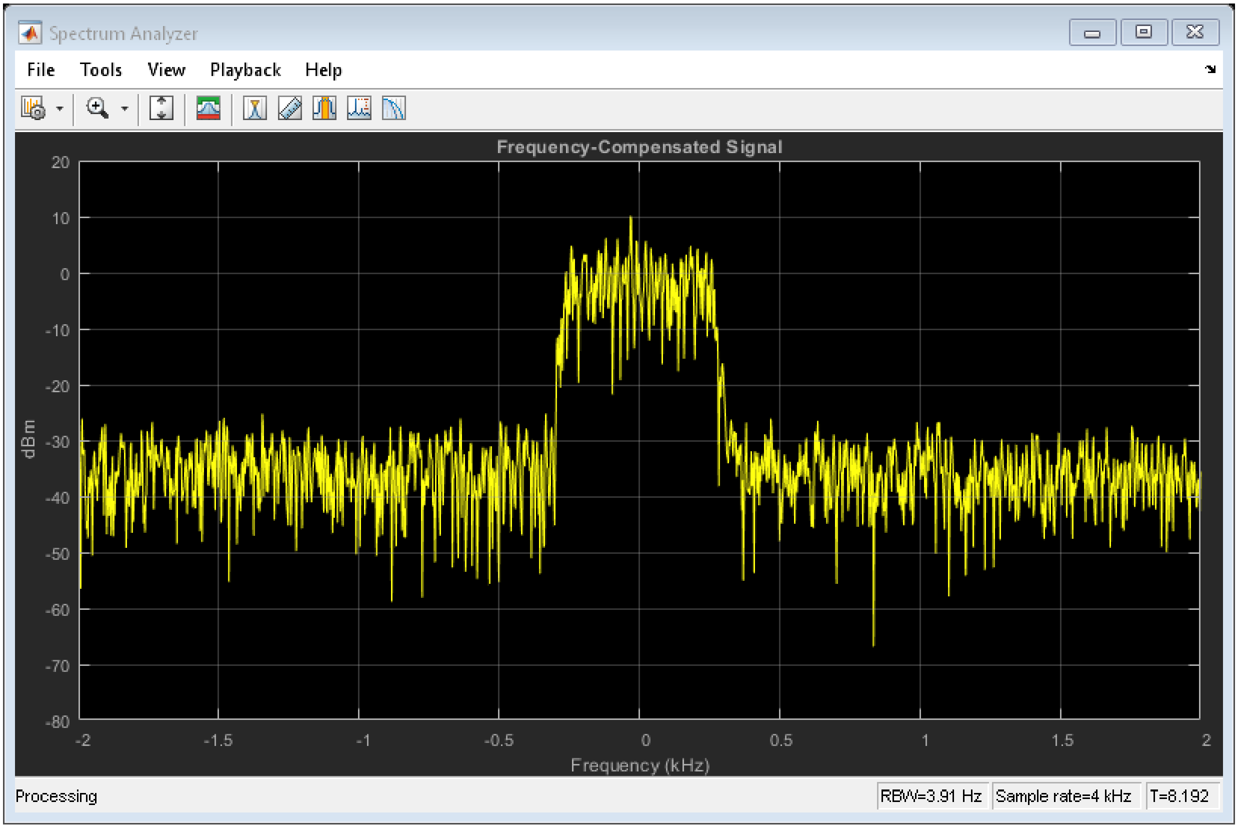
```
estFreqOffset = -250
```

Correct for the frequency offset using `pfoCorrect` and the inverse of the estimated frequency offset.

```
compensatedData = pfoCorrect(noisyData,-estFreqOffset);
```

Plot the frequency response of the compensated signal using the spectrum analyzer. The signal is now properly centered.

```
spectrum.Title = 'Frequency-Compensated Signal';
spectrum(compensatedData);
```



Selected Bibliography

[1] Nakagawa, T., Matsui, M., Kobayashi, T., Ishihara, K., Kudo, R., Mizoguchi, M., and Y. Miyamoto. “Non-data-aided wide-range frequency offset estimator for QAM optical coherent receivers”, *Optical Fiber Communication Conference and*

Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference , March, 2011, pp. 1–3.

[2] Wang, Y., Shi. K., and E. Serpedin. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach”, *EURASIP Journal on Advances in Signal Processing*, Vol. 13, 2004, pp. 1993–2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PSKCoarseFrequencyEstimator` | `comm.PhaseFrequencyOffset` | `dsp.FFT`

Introduced in R2013b

reset

System object: comm.QAMCoarseFrequencyEstimator
Package: comm

Reset states of the `QAMCoarseFrequencyEstimator` object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the internal states of the `QAMCoarseFrequencyEstimator` object, `H`.

step

System object: comm.PSKCoarseFrequencyEstimator
Package: comm

Estimate frequency offset for QAM signal

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a complex column vector of data type double. The step method outputs the estimate `Y` as a scalar of type double.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.QPSKDemodulator System object

Package: comm

Demodulate using QPSK method

Description

The `QPSKDemodulator` object demodulates a signal that was modulated using the quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using quadrature phase shift keying:

1. Define and set up your QPSK demodulator object. See “Construction” on page 3-1263.
2. Call `step` to demodulate the signal according to the properties of `comm.QPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 $+ j\times 2\pi\times {}^{m}\!/_{4}$).

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` on page 3-0 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` on page 3-0 property to `true` and the `DecisionMethod` on page 3-0 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` on page 3-0 property to `true`, the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the `VarianceSource` on page 3-0 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 on page 3-0 property to false. The property also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. In this second case, when the OutputDataType on page 3-0 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 on page 3-0 property to false. The property also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 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 on page 3-0 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 on page 3-0 property to Custom.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using QPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

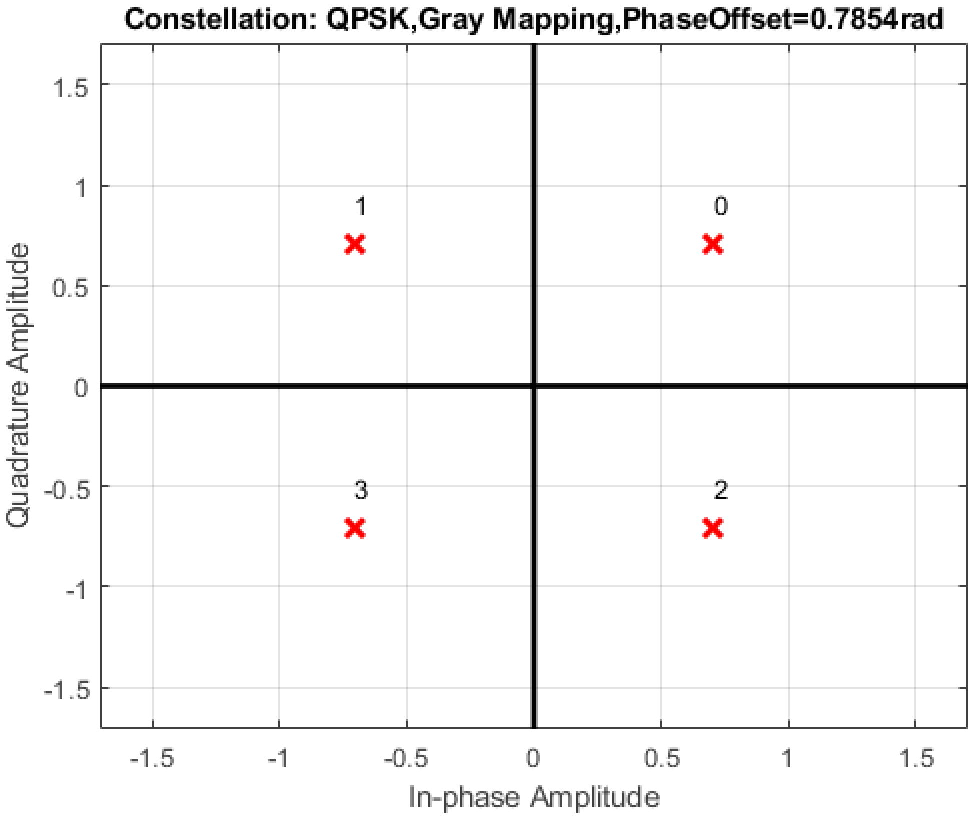
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

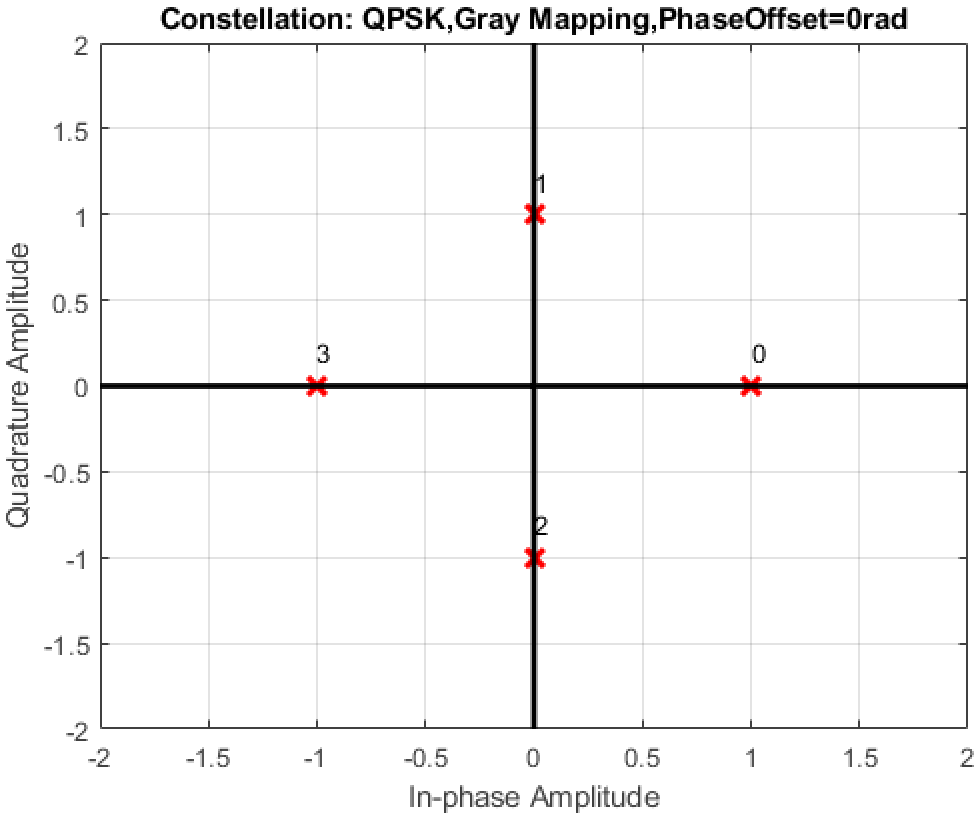


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



BER Estimate of QPSK Signal

Create a QPSK modulator and demodulator pair that operate on bits.

```
qpskModulator = comm.QPSKModulator('BitInput',true);
qpskDemodulator = comm.QPSKDemodulator('BitOutput',true);
```

Create an AWGN channel object and an error rate counter.

```
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
errorRate = comm.ErrorRate;
```

Generate random binary data and apply QPSK modulation.

```
data = randi([0 1],1000,1);
txSig = qpskModulator(data);
```

Pass the signal through the AWGN channel and demodulate it.

```
rxSig = channel(txSig);
rxData = qpskDemodulator(rxSig);
```

Calculate the error statistics. Display the BER.

```
errorStats = errorRate(data,rxData);

errorStats(1)
```

```
ans = 0.0100
```

- “LLR vs. Hard Decision Demodulation”

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PSKDemodulator` | `comm.QPSKModulator`

Topics

"LLR vs. Hard Decision Demodulation"

Introduced in R2012a

constellation

System object: comm.QPSKDemodulator
Package: comm

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

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

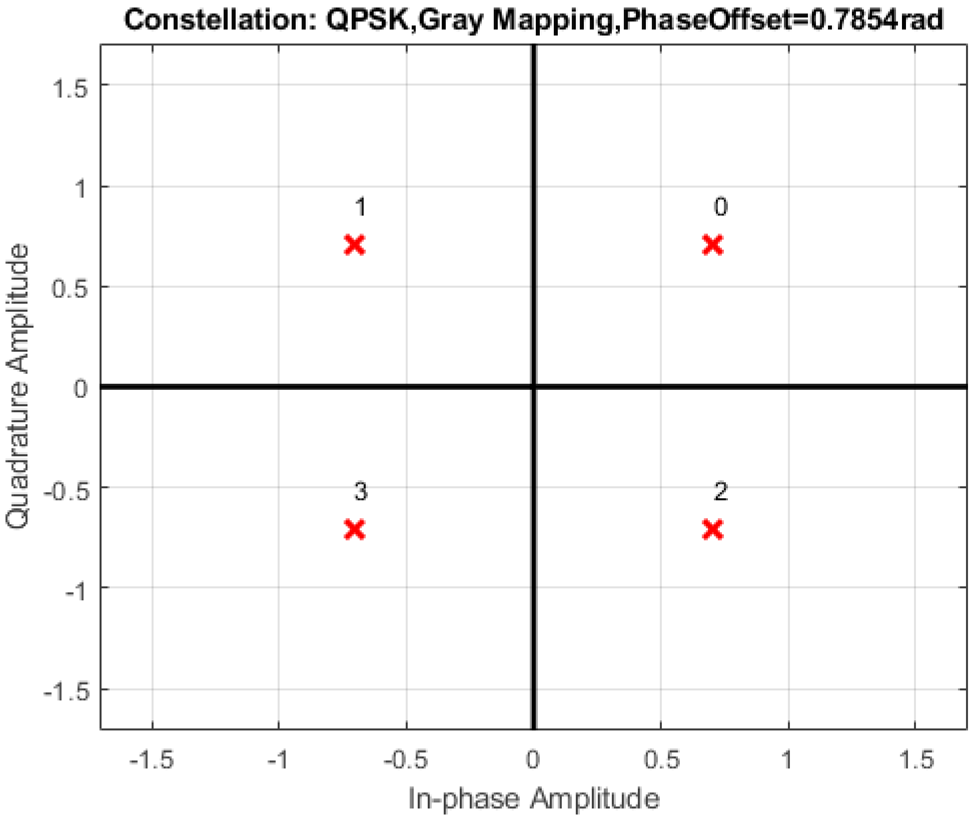
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

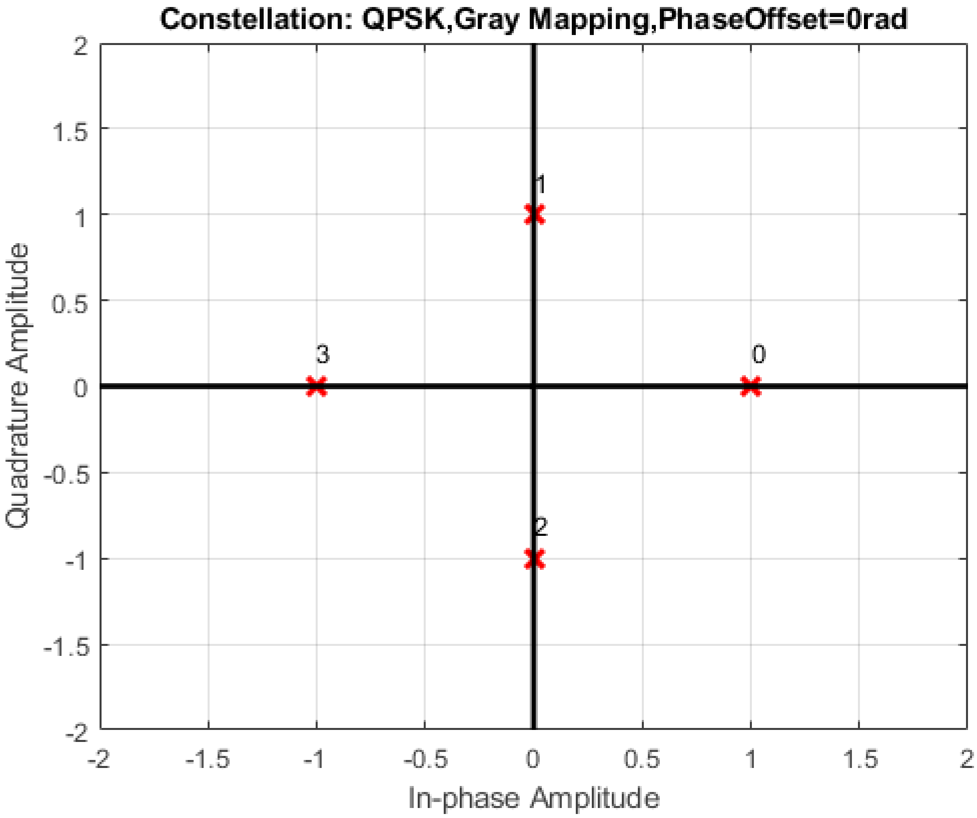


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```


Constellation: QPSK,Gray Mapping,PhaseOffset=0rad
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
3
0
2

step

System object: comm.QPSKDemodulator
Package: comm

Demodulate using QPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.QPSKModulator System object

Package: comm

Modulate using QPSK method

Description

The `QPSKModulator` object modulates using the quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using quadrature phase shift keying:

1. Define and set up your QPSK modulator object. See “Construction” on page 3-1277.
2. Call `step` to modulate the signal according to the properties of `comm.QPSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 **exp**($j\times$ `PhaseOffset` on page 3-0 $+ j\times 2\times \pi\times {}^{m}\!/_{4}$).

`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` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using QPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

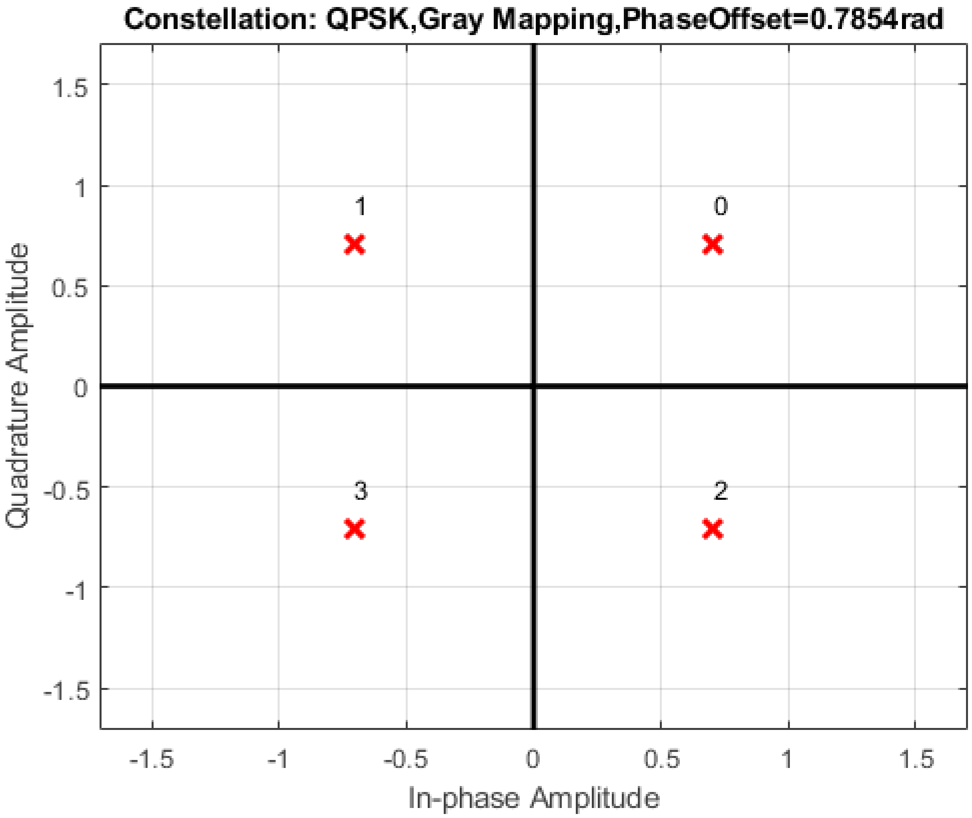
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

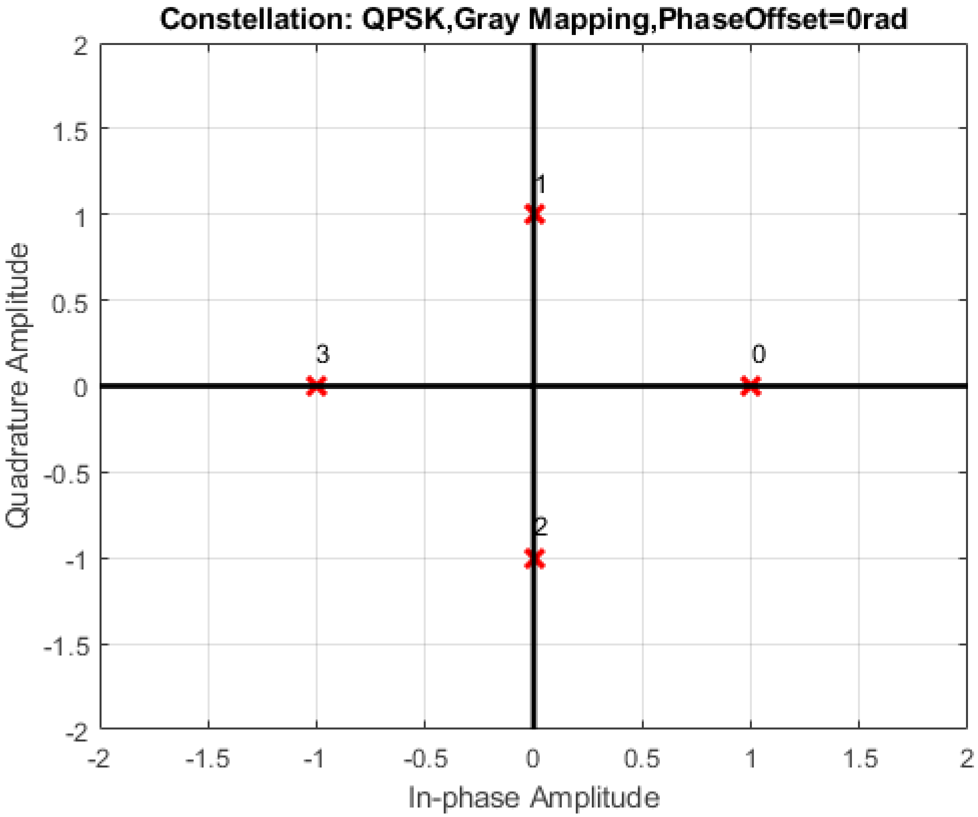


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



Phase Noise on QPSK Signal

Create a QPSK modulator object and a phase noise object.

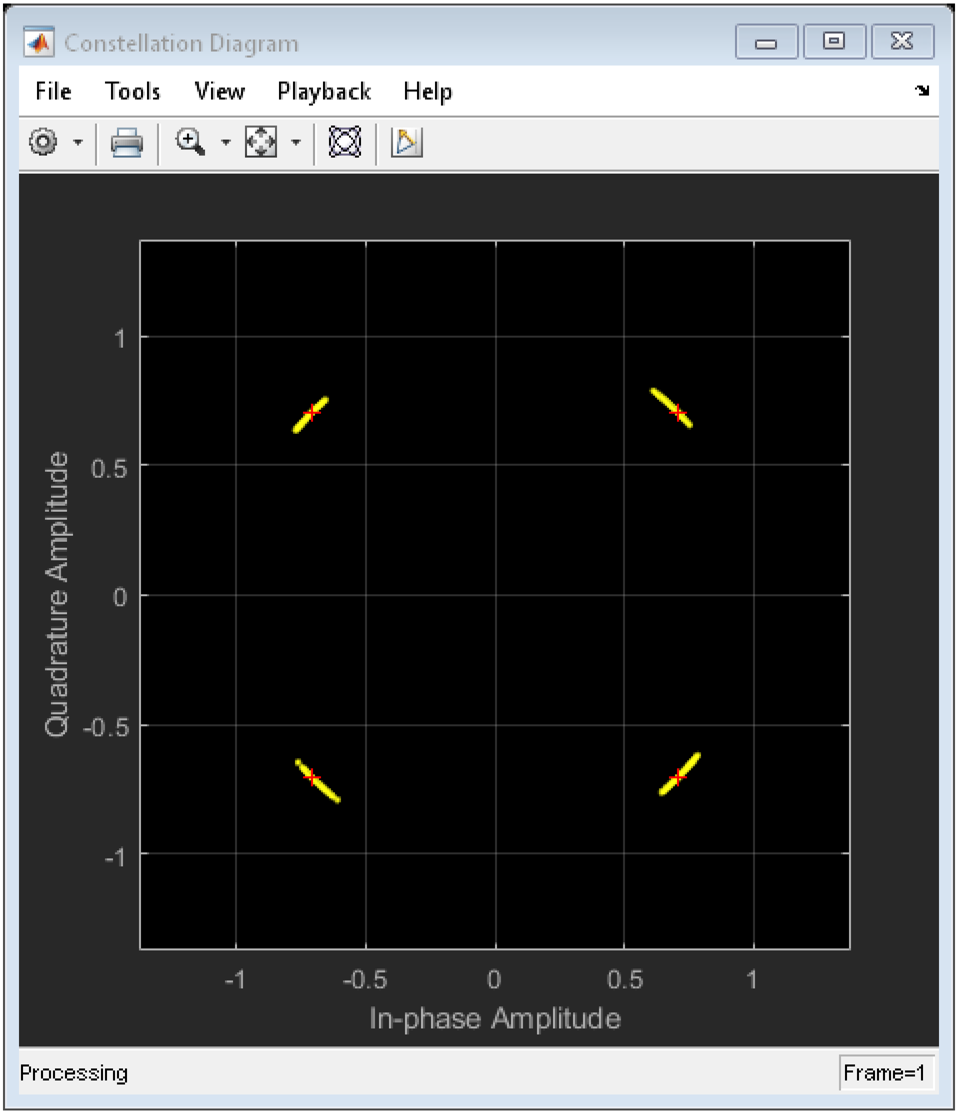
```
qpskModulator = comm.QPSKModulator;
phNoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',1000);
```

Generate random QPSK data. Pass the signal through the phase noise object.

```
d = randi([0 3],1000,1);
x = qpskModulator(d);
y = phNoise(x);
```

Display the constellation diagram of the QPSK signal. The phase noise has introduced a rotational distortion on the constellation diagram.

```
constDiagram = comm.ConstellationDiagram;

constDiagram(y)
```


Constellation Diagram
File
Tools
View
Playback
Help
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=1

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PSKModulator` | `comm.QPSKDemodulator`

Introduced in R2012a

constellation

System object: comm.QPSKModulator
Package: comm

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

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

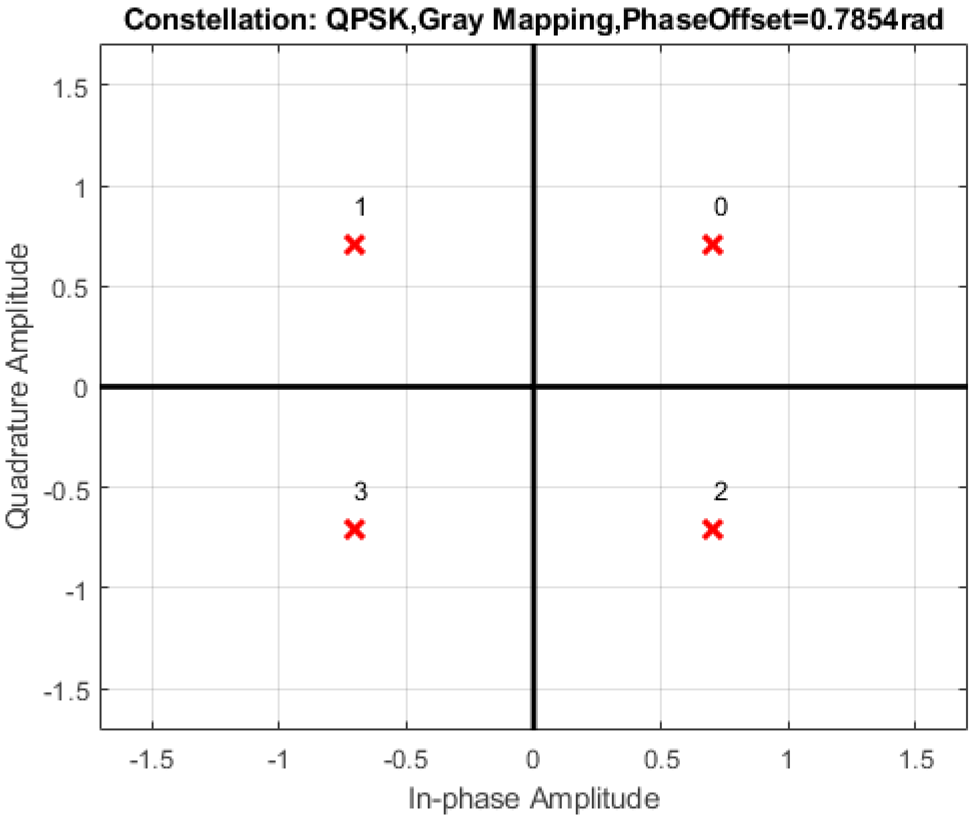
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

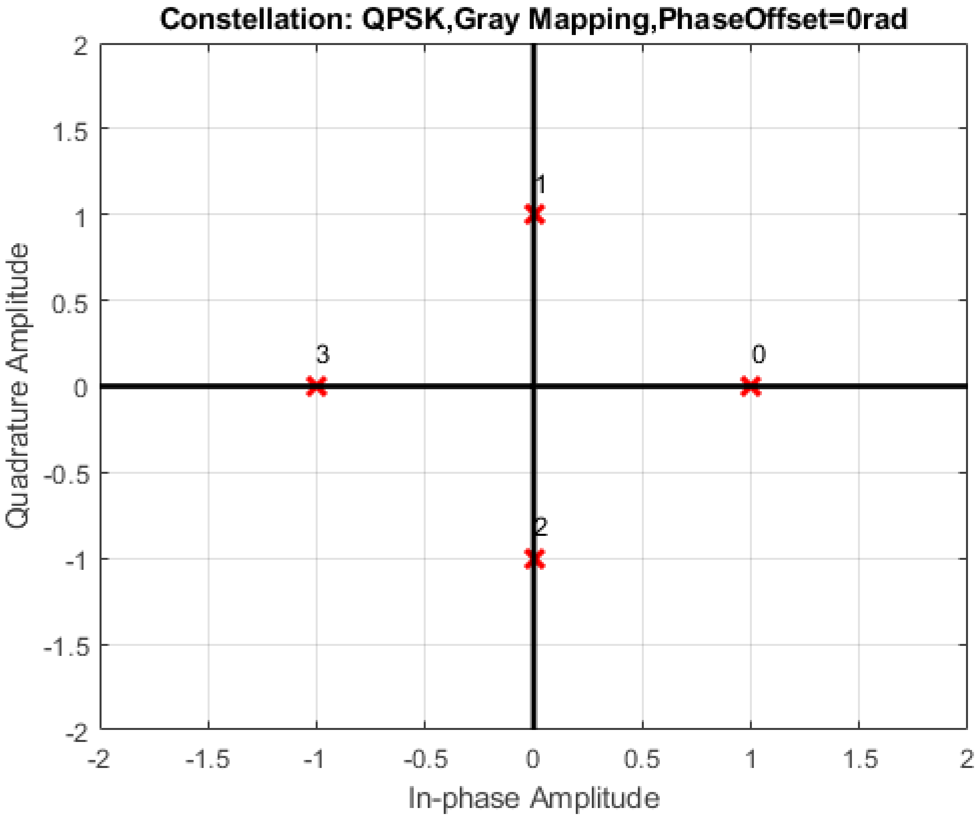


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```


Constellation: QPSK,Gray Mapping,PhaseOffset=0rad
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
3
0
2

step

System object: comm.QPSKModulator
Package: comm

Modulate using QPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RaisedCosineReceiveFilter System object

Package: comm

Apply pulse shaping by decimating signal using raised cosine filter

Description

The `Raised Cosine Receive Filter` System object applies pulse-shaping by decimating an input signal using a raised cosine FIR filter.

To decimate the input signal:

1. Define and set up your raised cosine receive filter object. See “Construction” on page 3-1289.
2. Call `step` to decimate the input signal according to the properties of `comm.RaisedCosineReceiveFilter`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RaisedCosineReceiveFilter` returns a raised cosine receive filter System object, `H`, which decimates the input signal. The filter uses an efficient polyphase FIR decimation structure and has unit energy.

`H = comm.RaisedCosineReceiveFilter(PropertyName,PropertyValue, ...)` returns a raised cosine receive filter object, `H`, with each specified property set to the specified value.

Properties

`Shape`

Filter shape

Specify the filter shape as one of `Normal` or `Square root`. The default is `Square root`.

`RolloffFactor`

Rolloff factor

Specify the rolloff factor as a scalar between `0` and `1`. The default is `0.2`.

`FilterSpanInSymbols`

Filter span in symbols

Specify the number of symbols the filter spans as an integer-valued positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the object truncates the impulse response to the value you specify for this property.

`InputSamplesPerSymbol`

Input samples per symbol

Specify the number of input samples that represent a symbol. The default is `8`. This property accepts an integer-valued, positive double or single scalar value. The raised cosine filter has (FilterSpanInSymbols x `InputSamplesPerSymbol` + 1) taps.

`DecimationFactor`

Decimation factor

Specify the factor by which the object reduces the sampling rate of the input signal. The default value is `8`. This property accepts a positive integer scalar value between `1` and `InputSamplesPerSymbol`. The value must evenly divide into InputSamplesPerSymbol. The number of input rows must be a multiple of the decimation factor. If you set `DecimationFactor` to `1`, then the object only applies filtering without downsampling.

DecimationOffset

Specify the number of filtered samples the System object discards before downsampling. The default is `0`. This property accepts an integer valued scalar between `0` and `DecimationFactor` − 1.

Gain

Linear filter gain

Specify the linear gain of the filter as a positive numeric scalar. The default is `1`. The object designs a raised cosine filter that has unit energy, and then applies the linear gain to obtain final tap values.

Methods

coeffs	Returns coefficients for filters
reset	Reset internal states of System object
step	Output decimated values of input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Filter Signal Using Square Root Raised Cosine Receive Filter

Filter the output of a square root raised cosine transmit filter using a matched square root raised cosine receive filter. The input signal has eight samples per symbol.

Create a raised cosine transmit filter and set the `OutputSamplesPerSymbol` property to 8.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',8);
```

Create a raised cosine receive filter and set the `InputSamplesPerSymbol` property to 8 and the `DecimationFactor` property to 8.

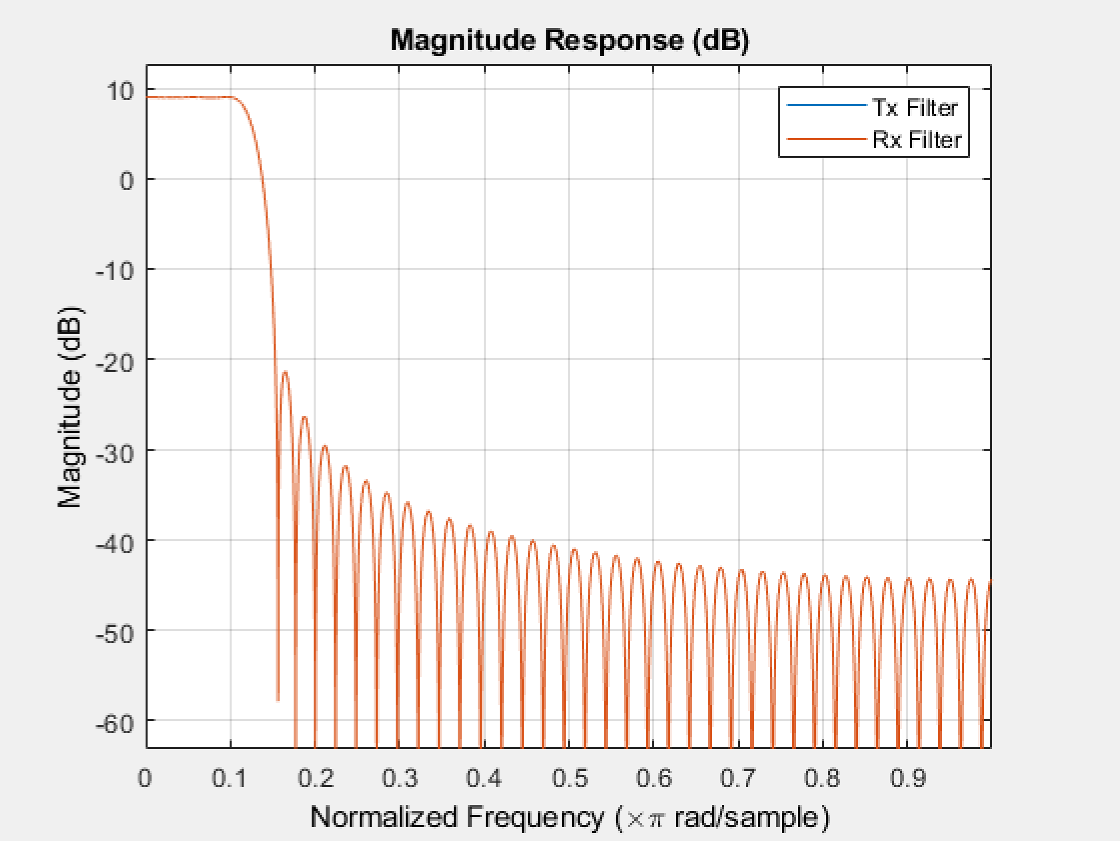
```
rxfilter = comm.RaisedCosineReceiveFilter('InputSamplesPerSymbol',8, ...
    'DecimationFactor',8);
```

Use the `coeffs` function to determine the filter coefficients for both filters.

```
txCoef = coeffs(txfilter);
rxCoef = coeffs(rxfilter);
```

Launch the filter visualization tool and display the magnitude responses of the two filters. Observe that they have identical responses.

```
 fvtool(txCoef.Numerator,1,rxCoef.Numerator,1);
 legend('Tx Filter','Rx Filter')
```



Generate a random bipolar signal and then interpolate.

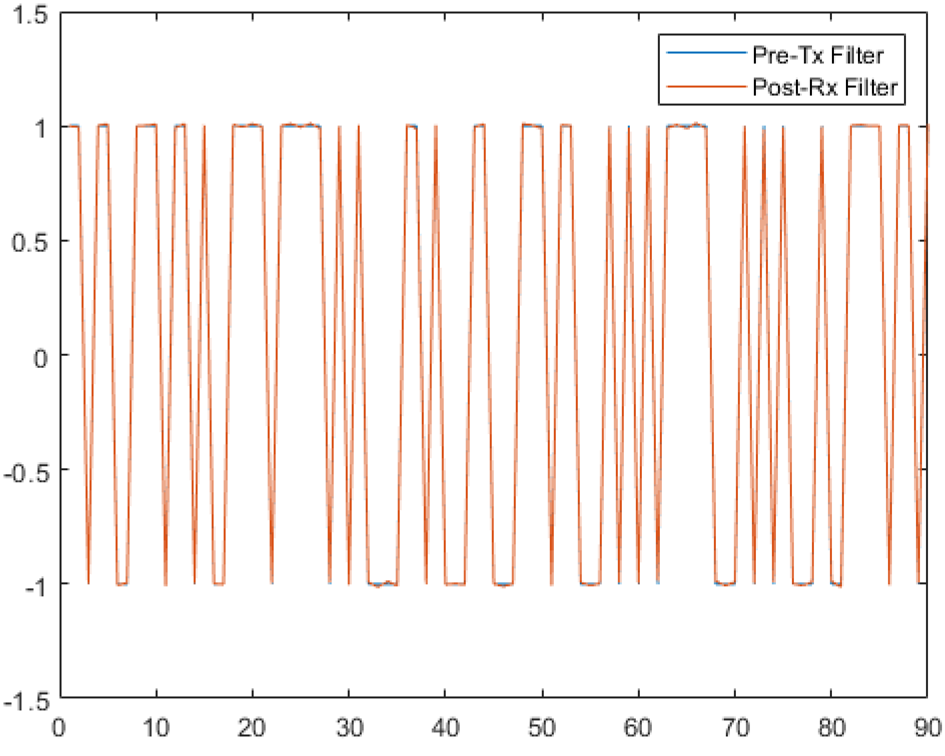
```
preTx = 2*randi([0 1],100,1) - 1;
y = txfilter(preTx);
```

Decimate the signal using the raised cosine receive filter System object.

```
postRx = rxfilter(y);
```

The filter delay is equal to the `FilterSpanInSymbols` property. Adjust for the delay to compare the pre-Tx filter signal with the post-Rx filter signal.

```
delay = txfilter.FilterSpanInSymbols;
x = (1:(length(preTx)-delay));
plot(x,preTx(1:end-delay),x,postRx(delay+1:end))
legend('Pre-Tx Filter','Post-Rx Filter')
```



You can see that the two signals overlap one another since the receive filter is *matched* to the transmit filter.

Specify Filter Span of Raised Cosine Receive Filter

Decimate a bipolar signal using a square root raised cosine filter whose impulse response is truncated to six symbol durations.

Create a raised cosine transmit filter and set the `FilterSpanInSymbols` property to 6. The object truncates the impulse response to six symbols.

```
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',6);
```

Generate a random bipolar signal and filter it using `txfilter`.

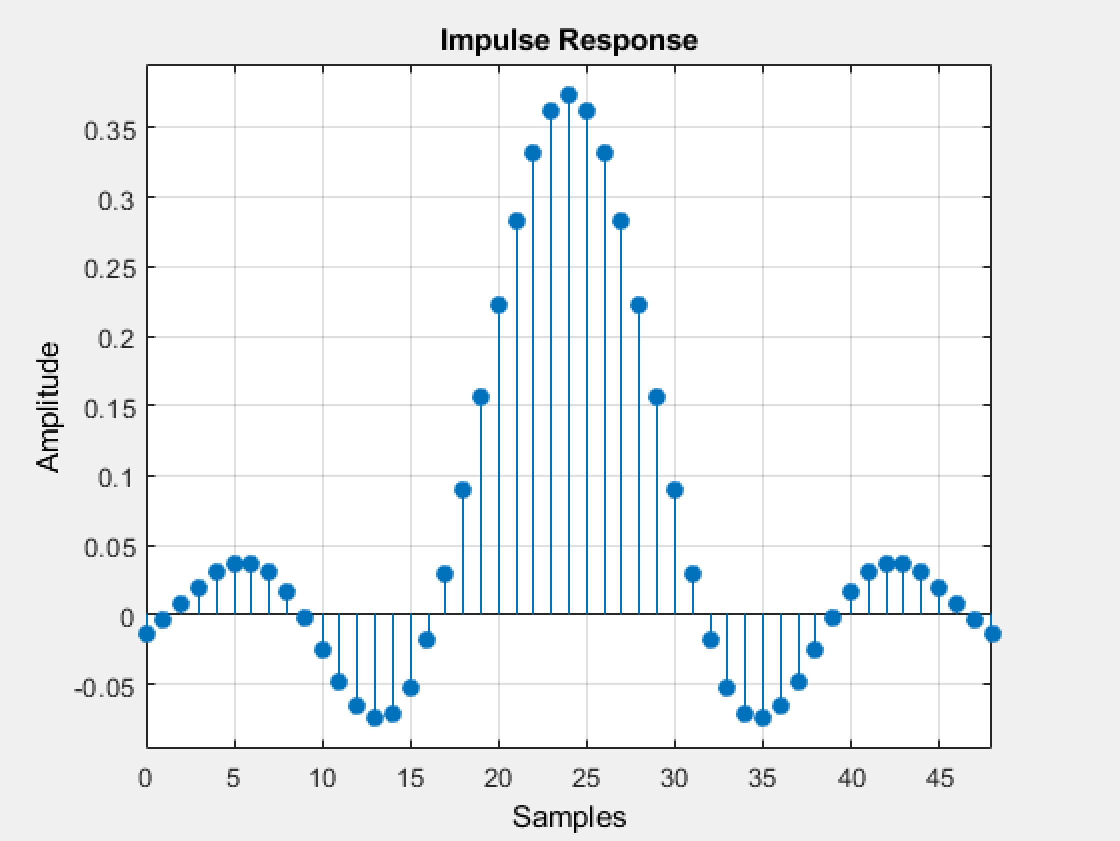
```
x = 2*randi([0 1],25,1) - 1;
y = txfilter(x);
```

Create a matched raised cosine receive filter System object.

```
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',6);
```

Launch the filter visualization tool to show the impulse response of the receive filter.

```
fvtool(rxfilter,'Analysis','impulse')
```

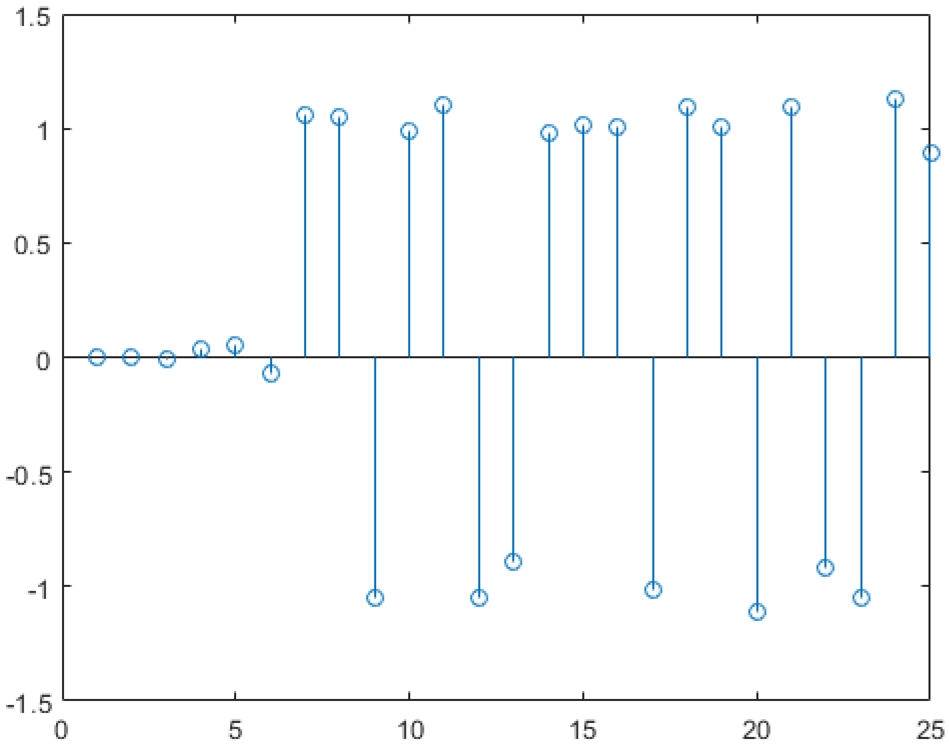


Filter the output signal from the transmit filter using the matched receive filter object, `rxfilter`.

```
r = rxfilter(y);
```

Plot the interpolated signal. Because of the filter span, there is a delay of six symbols before data passes through the filter.

```
stem(r)
```



Raised Cosine Receive Filter with Unity Passband Gain

Create a raised cosine receive filter with unity passband gain.

Create a raised cosine receive filter System object™. Obtain the filter coefficients using the `coeffs` function.

```
rxfilter = comm.RaisedCosineReceiveFilter;
b = coeffs(rxfilter);
```

A filter with unity passband gain has filter coefficients such that the sum of coefficients is 1. Therefore, set the `Gain` property to the inverse of the sum of `b.Numerator`.

```
rxfilter.Gain = 1/sum(b.Numerator);
```

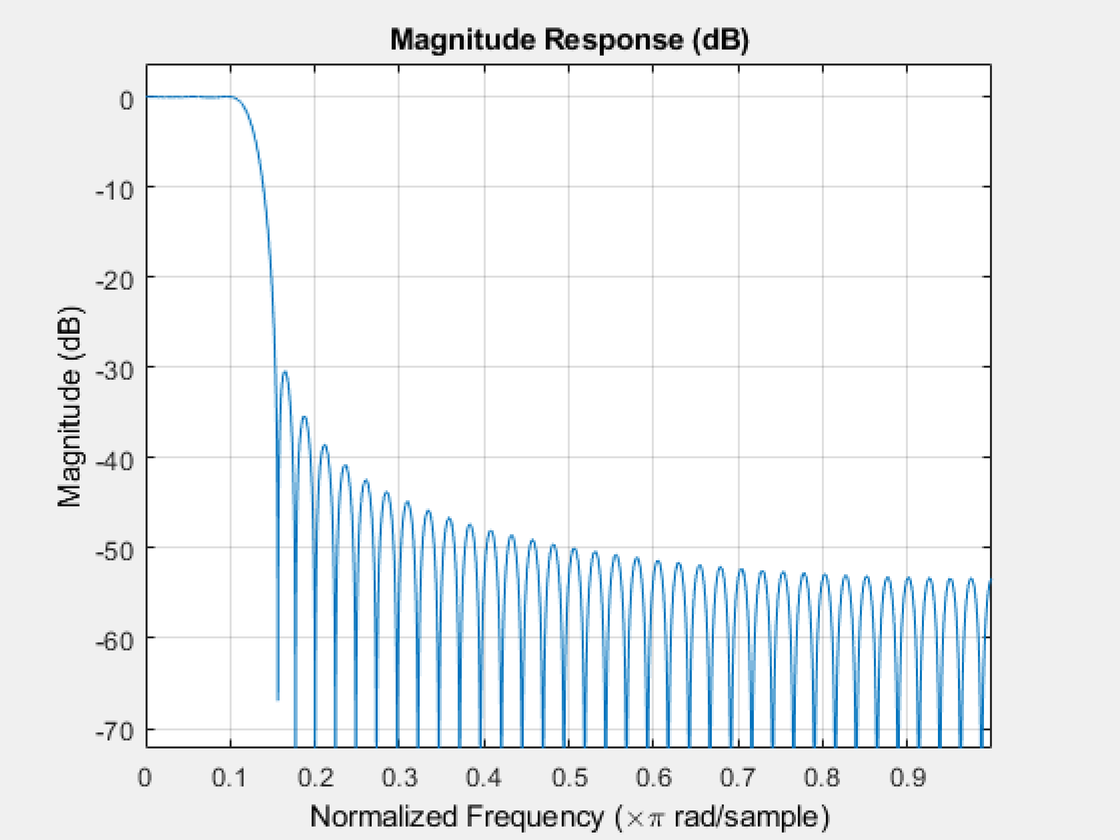
Verify that the sum of the coefficients from the resulting filter equal 1.

```
bNorm = coeffs(rxfilter);
sum(bNorm.Numerator)
```

```
ans = 1.0000
```

Plot the frequency response of the filter. Note that it shows a passband gain of 0 dB, which is unity gain.

```
fvtool(rxfilter)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.RaisedCosineTransmitFilter` | `rcosdesign`

Introduced in R2013b

coeffs

System object: comm.RaisedCosineReceiveFilter
Package: comm

Returns coefficients for filters

Syntax

```
S = coeffs(H)
S = coeffs(H,'Arithmetic',ARITH,...)
```

Description

`S = coeffs(H)` Returns the coefficients of filter System object, `H`, in the structure `S`.

`S = coeffs(H,'Arithmetic',ARITH,...)` analyzes the filter System object, `H`, based on the arithmetic specified in the `ARITH` input. `ARITH` can be set to one of `double`, `single`, or `fixed`. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The coeffs method returns the quantized filter coefficients when you set `ARITH` to `single` or `fixed`.

reset

System object: comm.RaisedCosineReceiveFilter
Package: comm

Reset internal states of System object

Syntax

```
reset(OBJ)
```

Description

`reset(OBJ)` resets the internal states of System object OBJ to their initial values.

step

System object: comm.RaisedCosineReceiveFilter
Package: comm

Output decimated values of input signal

Syntax

```
Y = step(H, X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H, X)` outputs the decimated values, `Y`, of the input signal `X`. The System object treats the input matrix K_i-by-`N` as `N` independent channels. The System object filters each channel over time and generates a K_o-by-`N` output matrix. $K_o = K_i/M$ where `M` represents the decimation factor. The System object supports real and complex floating-point inputs.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RaisedCosineTransmitFilter System object

Package: comm

Apply pulse shaping by interpolating signal using raised cosine filter

Description

The `Raised Cosine Transmit Filter` System object applies pulse-shaping by interpolating an input signal using a raised cosine FIR filter.

To interpolate the input signal:

1. Define and set up your raised cosine transmit filter object. See “Construction” on page 3-1302.
2. Call `step` to interpolate the input signal according to the properties of `comm.RaisedCosineTransmitFilter`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RaisedCosineTransmitFilter` returns a raised cosine transmit filter System object, `H`, which interpolates an input signal using a raised cosine FIR filter. The filter uses an efficient polyphase FIR interpolation structure and has unit energy.

`H = comm.RaisedCosineTransmitFilter(PropertyName,PropertyValue, ...)` returns a raised cosine transmit filter object, `H`, with each specified property set to the specified value.

Properties

Shape

Filter shape

Specify the filter shape as one of `Normal` or `Square root`. The default is `Square root`.

RolloffFactor

Rolloff factor

Specify the rolloff factor as a scalar between `0` and `1`. The default is `0.2`.

FilterSpanInSymbols

Filter span in symbols

Specify the number of symbols the filter spans as an integer-valued, positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the object truncates the impulse response to the value you specify for this property.

OutputSamplesPerSymbol

Output samples per symbol

Specify the number of output samples for each input symbol. The default is `8`. This property accepts an integer-valued, positive scalar value. The raised cosine filter has (FilterSpanInSymbols x OutputSamplesPerSymbol + 1) taps.

Gain

Linear filter gain

Specify the linear gain of the filter as a positive numeric scalar. The default is `1`. The object designs a raised cosine filter that has unit energy, and then applies the linear gain to obtain final tap values.

Methods

coeffs	Returns coefficients for filters
reset	Reset internal states of System object
step	Output interpolated values of input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Interpolate Signal Using Square Root Raised Cosine Filter

This example shows how to interpolate a signal using the `comm.RaisedCosineTransmitFilter` System object and to display its spectrum.

Create a square root raised square root cosine transmit filter object. You can see that its default settings are such that the filter has a square root shape and that there are 8 samples per symbol.

```
txfilter = comm.RaisedCosineTransmitFilter
```

```
txfilter = 
  comm.RaisedCosineTransmitFilter with properties:

                    Shape: 'Square root'
            RolloffFactor: 0.2000
       FilterSpanInSymbols: 10
    OutputSamplesPerSymbol: 8
                     Gain: 1
```

Generate random bipolar data.

```
data = 2*randi([0 1],10000,1) - 1;
```

Filter the data by using the RRC filter.

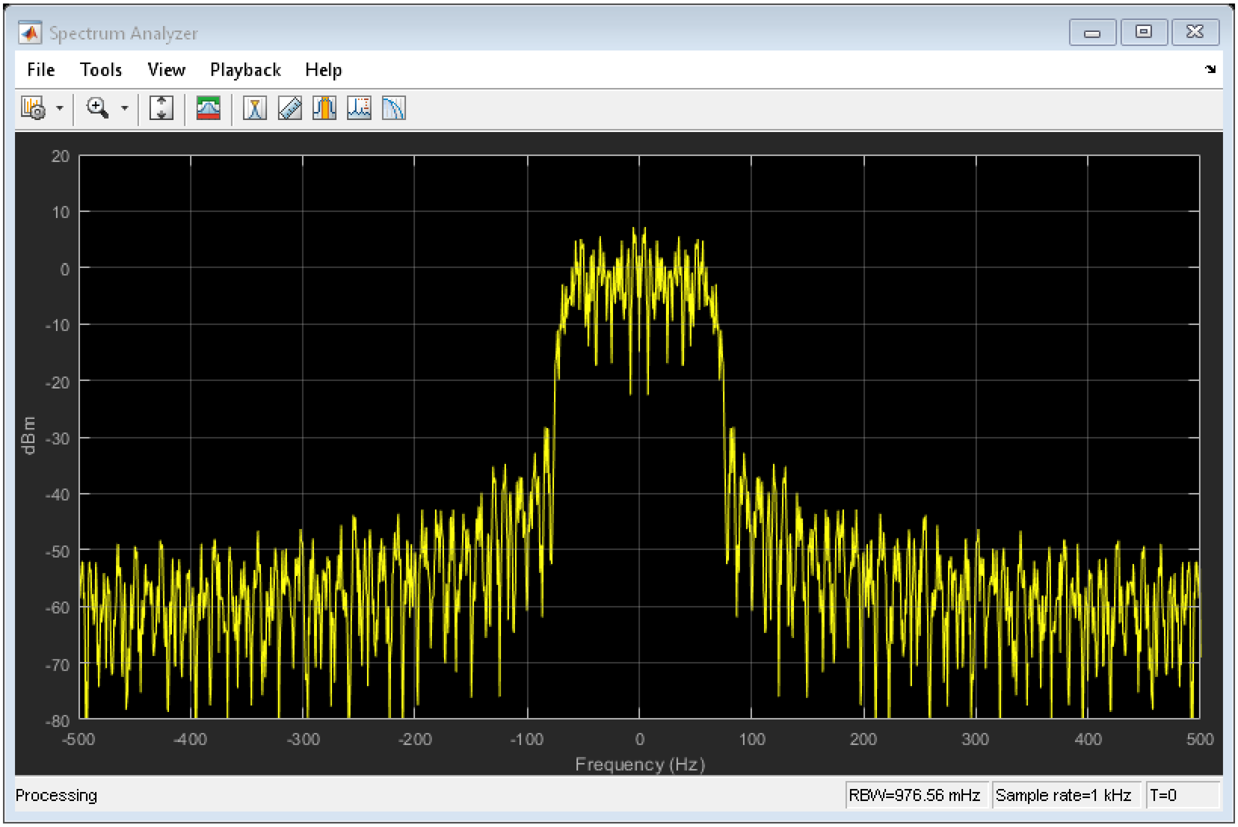
```
filteredData = txfilter(data);
```

To view the spectrum of the filtered signal, create a spectrum analyzer object with a sample rate of 1000 Hz.

```
spectrumAnalyzer = dsp.SpectrumAnalyzer('SampleRate',1000);
```

View the spectrum of the filtered signal using the spectrum analyzer.

```
spectrumAnalyzer(filteredData)
```



Specify Filter Span of Raised Cosine Transmit Filter

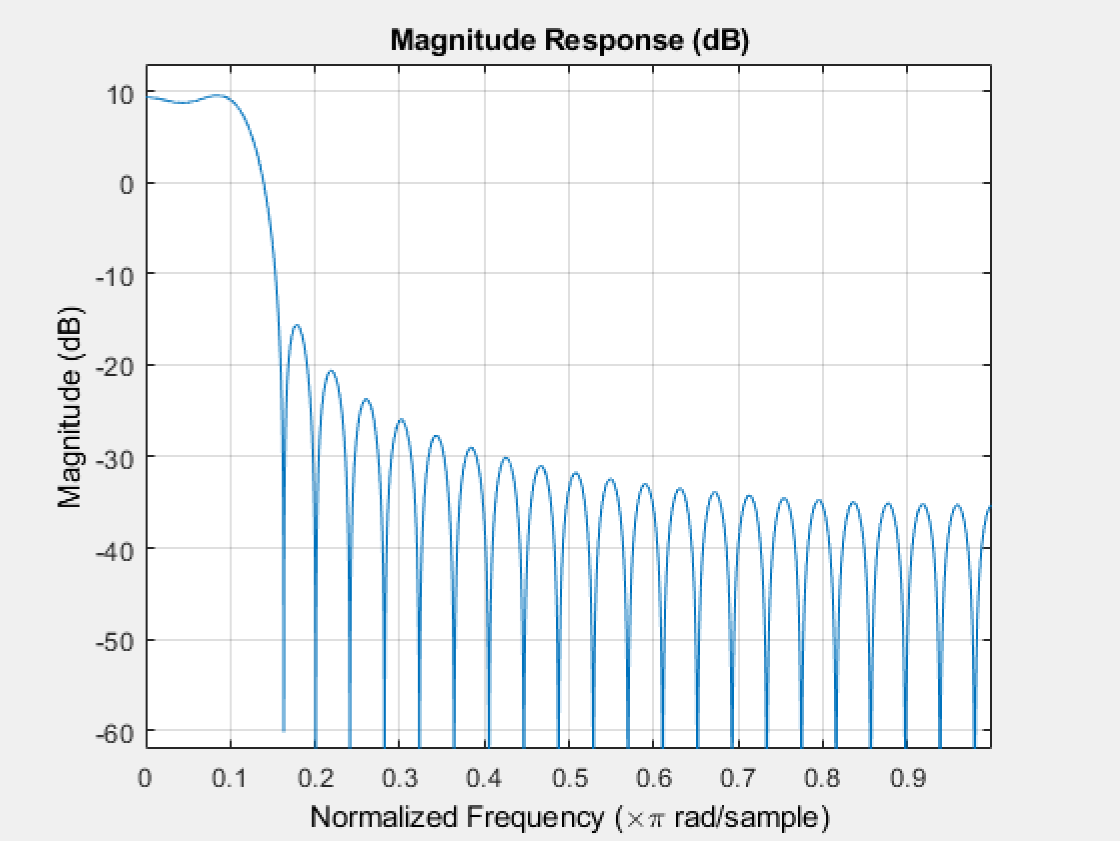
This example shows to create an interpolated signal from a square root raised cosine filter that is truncated to six symbol durations.

Create a raised cosine filter and set the `FilterSpanInSymbols` to 6. The object truncates the impulse response to six symbols.

```
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',6);
```

Launch the filter visualization tool to show the impulse response.

```
fvtool(txfilter)
```

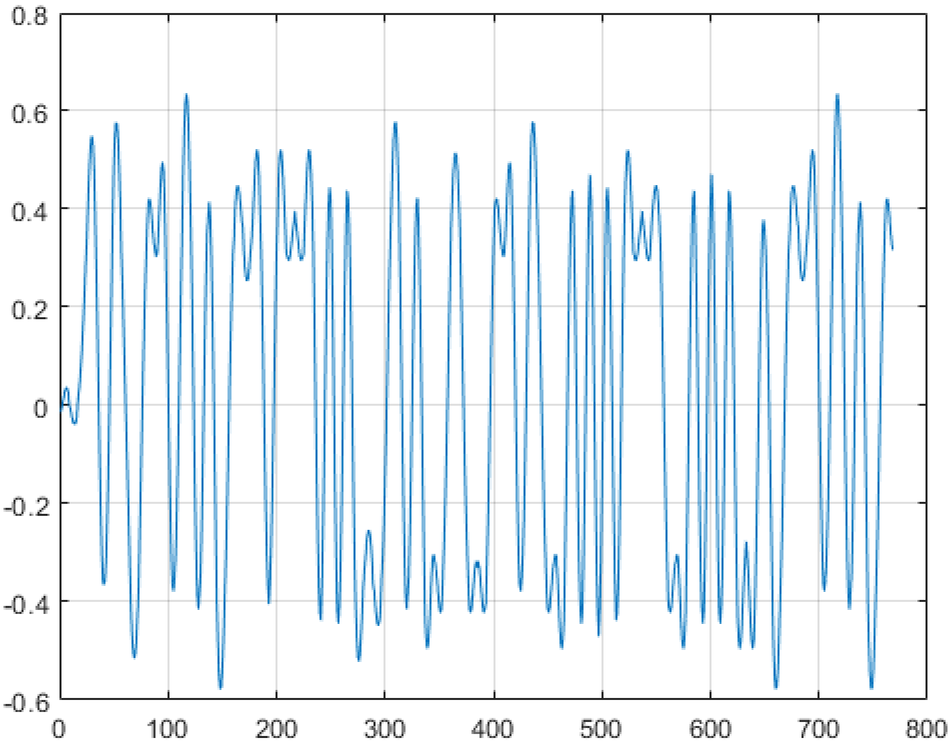


Generate random bipolar data and pass it through the filter.

```
x = 2*randi([0 1],96,1) - 1;
y = txfilter(x);
```

Plot the interpolated signal.

```
plot(y)
grid on
```



Raised Cosine Transmit Filter with Unity Passband Gain

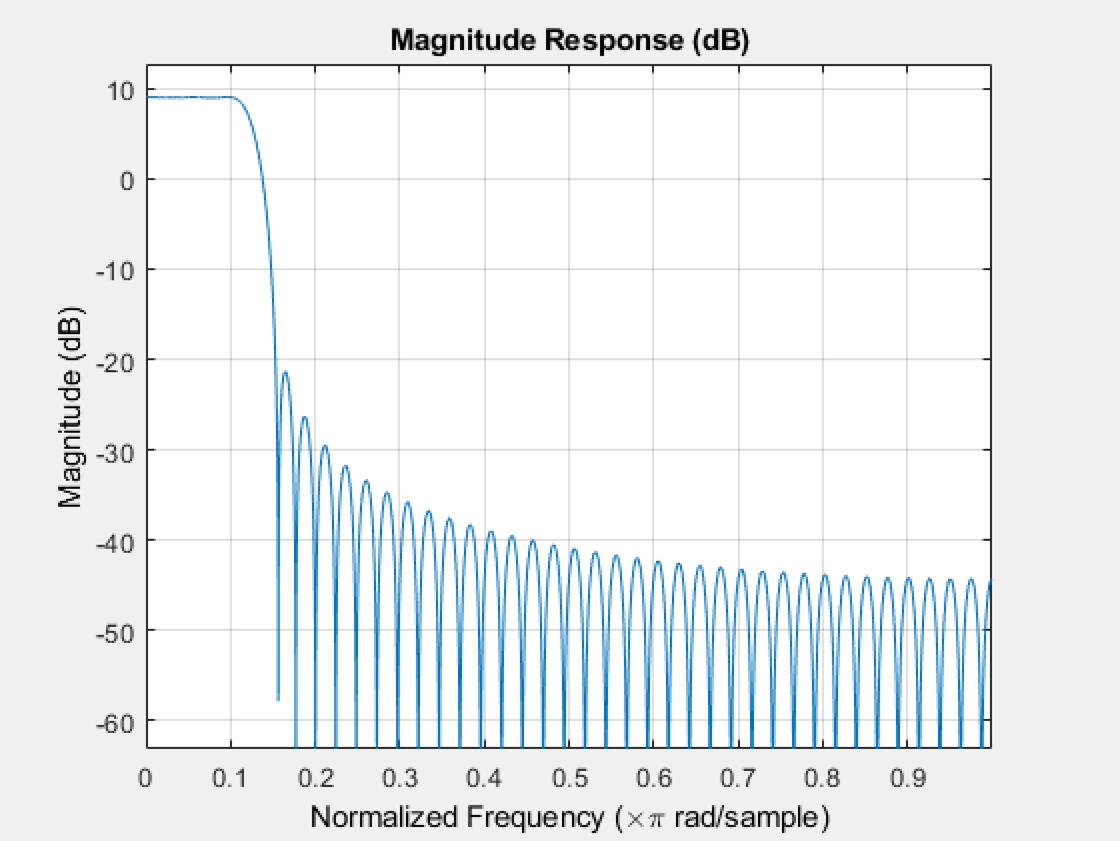
This example shows how to create a raised cosine transmit filter with unity passband gain.

Generate a filter with unit energy. You can obtain the filter coefficients using the `coeffs` function.

```
txfilter = comm.RaisedCosineTransmitFilter;
b = coeffs(txfilter);
```

Plot the filter response. You can see that its gain is greater than unity (more than 0 dB).

```
fvtool(txfilter)
```



A filter with unity passband gain has filter coefficients that sum to 1. Set the `Gain` property to the inverse of the sum of `b.Numerator`

```
txfilter.Gain = 1/sum(b.Numerator);
```

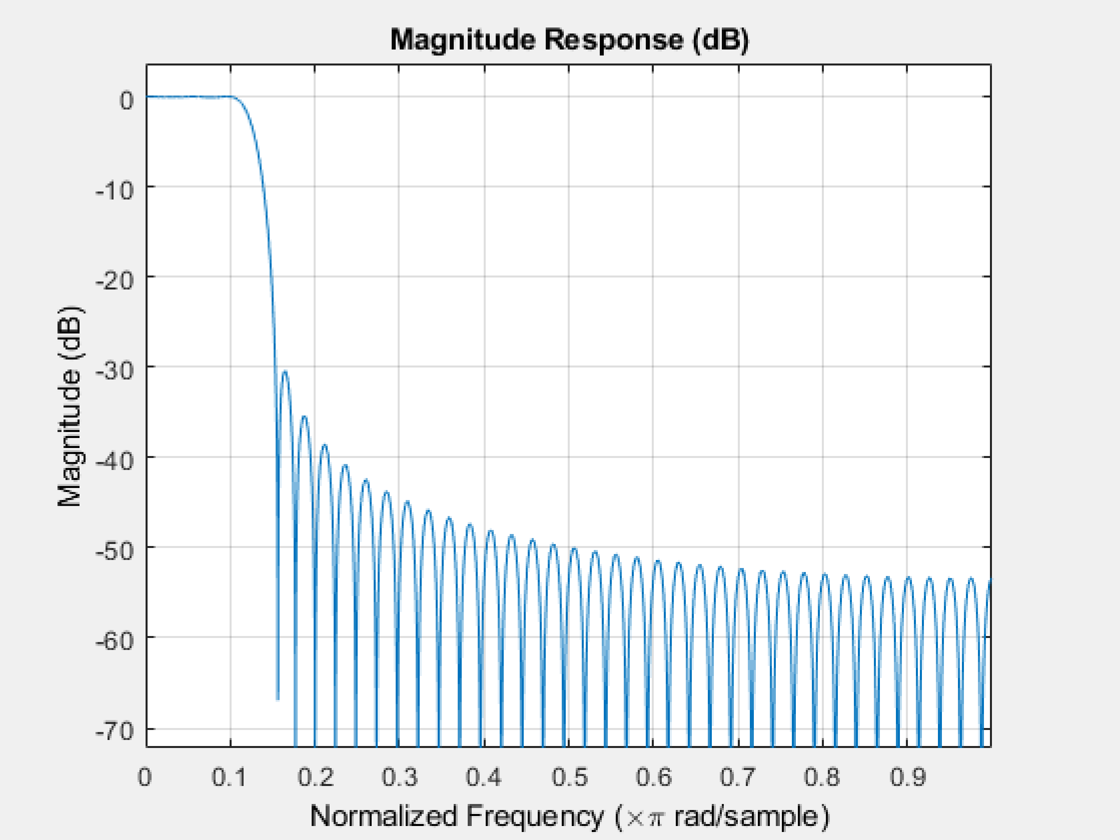
Verify that the resulting filter coefficients sum to 1.

```
bNorm = coeffs(txfilter);
sum(bNorm.Numerator)
```

```
ans = 1.0000
```

Plot the filter frequency response. Note that it shows a passband gain of 0 dB.

```
fvtool(txfilter)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.RaisedCosineReceiveFilter` | `rcosdesign`

Introduced in R2013b

coeffs

System object: comm.RaisedCosineTransmitFilter
Package: comm

Returns coefficients for filters

Syntax

```
S = coeffs(H)
S = coeffs(H,'Arithmetic',ARITH,...)
```

Description

`S = coeffs(H)` Returns the coefficients of filter System object, `H`, in the structure `S`.

`S = coeffs(H,'Arithmetic',ARITH,...)` analyzes the filter System object, `H`, based on the arithmetic specified in the `ARITH` input. `ARITH` can be set to one of `double`, `single`, or `fixed`. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The coeffs method returns the quantized filter coefficients when you set `ARITH` to `single` or `fixed`.

reset

System object: comm.RaisedCosineTransmitFilter
Package: comm

Reset internal states of System object

Syntax

```
reset(OBJ)
```

Description

`reset(OBJ)` resets the internal states of System object OBJ to their initial values.

step

System object: comm.RaisedCosineTransmitFilter
Package: comm

Output interpolated values of input signal

Syntax

```
Y = step(H, X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H, X)` outputs the interpolated values, `Y`, of the input signal `X`. The System object treats the input matrix K_i-by-`N` as `N` independent channels. The object interpolates each channel over the first dimension and then generates a K_o-by-`N` output matrix. In the output matrix, $K_o = K_i*L$, where `L` represents the output samples per symbol. The object supports real and complex floating-point inputs.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RayleighChannel System object

Package: comm

Filter input signal through a Rayleigh multipath fading channel

Description

The `RayleighChannel` System object filters an input signal through a Rayleigh fading channel. The fading processing per link is per the Methodology for Simulating Multipath Fading Channels

To filter an input signal using a Rayleigh multipath fading channel:

1. Define and set up your Rayleigh channel object. See "Construction" on page 3-1314.
2. Call `step` to filter the input signal through a Rayleigh multipath fading channel according to the properties of `comm.Rayleighhannel`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RayleighChannel` creates a frequency-selective or frequency-flat multipath Rayleigh fading channel System object, `H`. This object filters a real or complex input signal through the multipath channel to obtain the channel impaired signal.

`H = comm.RayleighChannel(Name,Value)` creates a multipath Rayleigh 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 `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 channel is frequency flat.

When you set `PathDelays` to a vector, the 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` on page 3-0 .

NormalizePathGains

Normalize average path gains to 0 dB

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.

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 most Doppler spectrum types, the value of f_c is `1`. For Gaussian and BiGaussian Doppler spectrum types, f_c is dependent on the Doppler spectrum structure fields. For more details about how f_c is defined, see “Cutoff Frequency Factor” on page 3-1333.

DopplerSpectrum

Doppler spectrum object(s)

Specify the Doppler spectrum shape for the path(s) of the channel. This property accepts a single Doppler spectrum structure returned from the `doppler` function or a row cell array of such structures. The maximum Doppler shift value necessary to specify the Doppler spectrum/spectra is given by the `MaximumDopplerShift` on page 3-0 property. This property applies when the `MaximumDopplerShift` on page 3-0 property value is greater than 0. The default value of this property is `doppler('Jakes')`.

If you assign a single Doppler spectrum structure to `DopplerSpectrum`, all paths have the same specified Doppler spectrum. If the `FadingTechnique` property is `Sum of sinusoids`, `DopplerSpectrum` must be `doppler('Jakes')`; otherwise, select from the following:

- doppler('Jakes')
- doppler('Flat')
- doppler('Rounded', ...)
- doppler('Bell', ...)
- doppler('Asymmetric Jakes', ...)
- doppler('Restricted Jakes', ...)
- doppler('Gaussian', ...)
- doppler('BiGaussian', ...)

If you assign a row cell array of different Doppler spectrum structures (which can be chosen from any of those on the previous list) to DopplerSpectrum, each path has the Doppler spectrum specified by the corresponding structure in the cell array. In this case, the length of DopplerSpectrum must be equal to the length of PathDelays.

To generate C code, specify this property to a single Doppler spectrum structure. The default value of this property is doppler('Jakes').

FadingTechnique

Fading technique used to model the channel

Select between `Filtered Gaussian noise` and `Sum of sinusoids` to specify the way in which the channel is modeled. The default value is `Filtered Gaussian noise`.

NumSinusoids

Number of sinusoids used to model the fading process

The `NumSinuoids` property is a positive integer scalar that specified the number of sinusoids used in modeling the channel and is available only when the `FadingTechnique` property is set to `Sum of sinusoids`. The default value is `48`.

InitialTimeSource

Source to control the start time of the fading process

Specify the initial time source as either `Property` or `Input port`. This property is available when the `FadingTechnique` property is set to `Sum of sinusoids`. When `InitialTimeSource` is set to `Input port`, the start time of the fading process is specified using the `INITIALTIME` input to the `step` function. The input value can change in consecutive calls to the `step` function. The default value is `Property`.

InitialTime

Start time of the fading process (s)

Specify the time offset of the fading process as a real nonnegative scalar in seconds. This property applies when the `FadingTechnique` property is set to `Sum of sinusoids` and the `InitialTimeSource` property is set to `Property`. The default value is `0`.

`InitialTime` must be greater than the last frame end time. When `InitialTime` is not a multiple of 1/`SampleRate`, it is rounded up to the nearest sample position.

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.

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`.

Visualization

Enable channel visualization

Specify the type of channel visualization to display as one of `Off` | `Impulse response` | `Frequency response` | `Impulse and frequency responses` | `Doppler spectrum`. The default value of this property is `Off`.

SamplesToDisplay

Specify percentage of samples to display

You can specify the percentage of samples to display, since displaying fewer samples will result in better performance at the expense of lower accuracy. Specify the property as one of `10%` | `25%` | `50%` | `100%`. This applies when `Visualization` is set to `Impulse` response, Frequency response, or Impulse and frequency responses. The default value is 25%.

PathsForDopplerDisplay

Specify path for Doppler display

You can specify an integer scalar which selects the discrete path used in constructing a Doppler spectrum plot. The specified path must be an element of {1, 2, ..., N_p}, where N_p is the number of discrete paths per link specified in the object. This property applies when Visualization is set to Doppler spectrum. The default value is 1.

Methods

info	Display information about the RayleighChannel object
reset	Reset states of the RayleighChannel object
step	Filter input signal through multipath Rayleigh fading channel

Common to All System Objects	
release	Allow System object property value changes

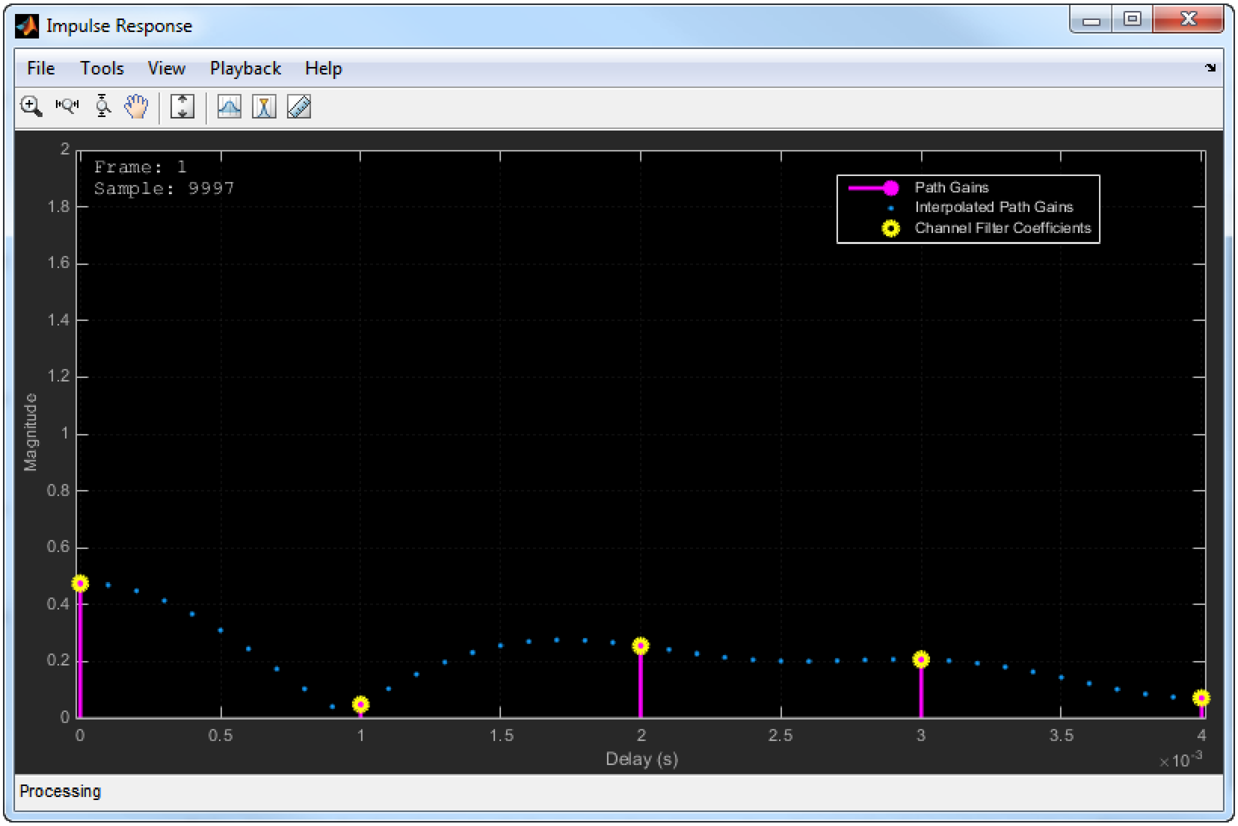
Visualization

Impulse Response

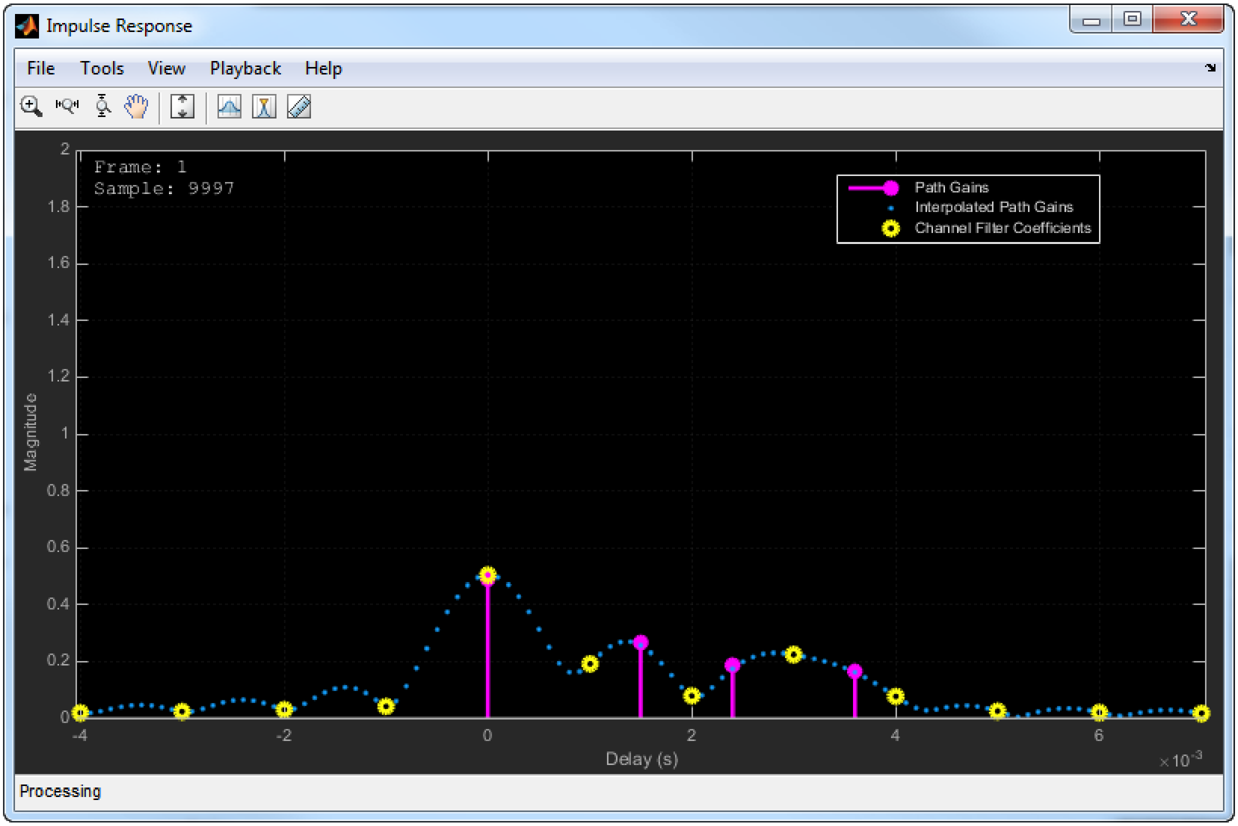
The impulse response plot displays the path gains, the channel filter coefficients, and the interpolated path gains. The path gains shown in magenta occur at time instances which correspond to the specified PathDelays property and may not be aligned with the input sampling time. The channel filter coefficients shown in yellow are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. In cases in which the path gains are aligned with the sampling time, they will overlap the filter coefficients. Sinc interpolation is used to connect the channel filter coefficients and is shown in blue. These points are used solely for display purposes and not used in subsequent channel filtering. For a flat fading channel (one path), the sinc interpolation curve is not displayed. For all impulse response plots, the frame and sample numbers are shown in the display’s upper left corner.

The impulse response plot shares the same toolbar and menus as the System object it was based on, `dsp.ArrayPlot`.

In the figure, the impulse response of a channel is shown for the case in which the path gains are aligned with the sample time. The overlap between the path gains and filter coefficients is evident.



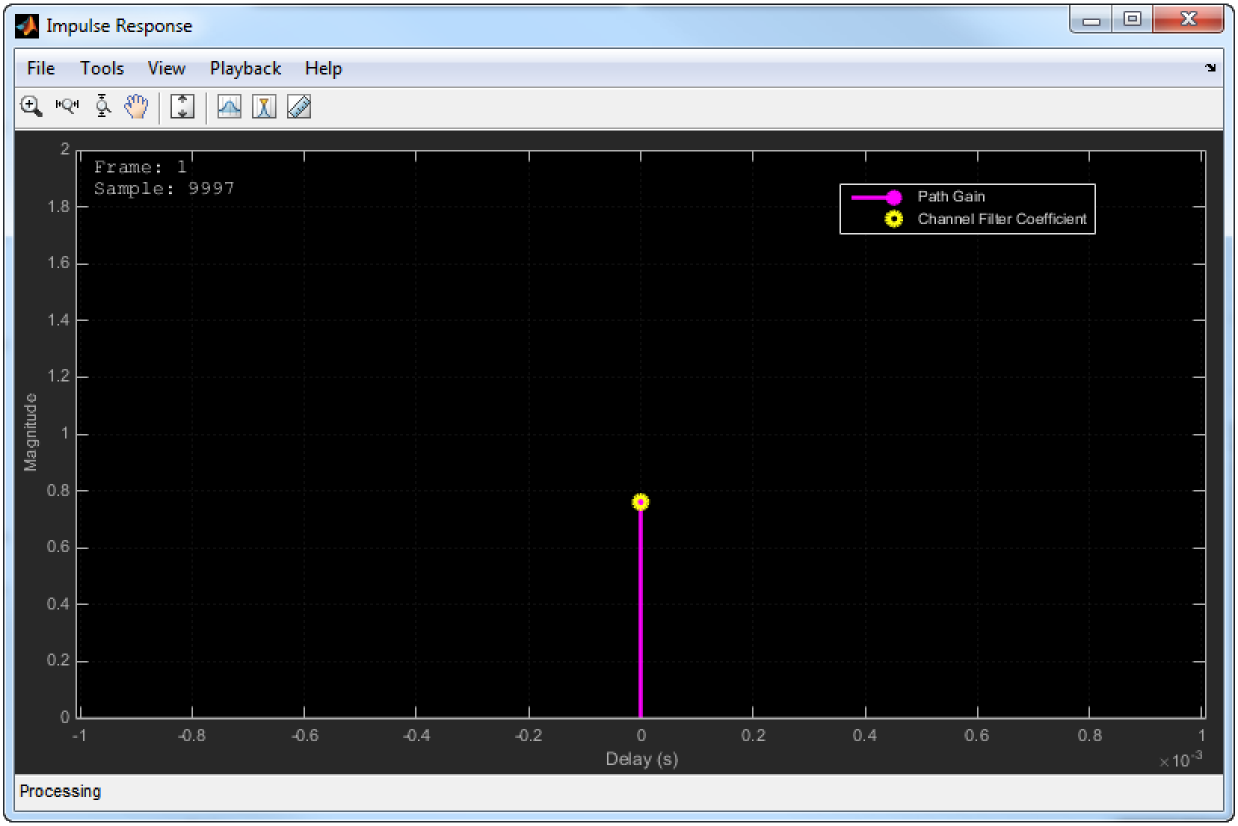
The case in which the specified path gains are not aligned with the `SampleRate` property is shown below. Observe that the path gains and the channel filter coefficients do not overlap and that the filter coefficients are equally distributed.



```
Frame: 1
Sample: 9997
```

Processing

The impulse response for a frequency flat channel is shown below. You can see that the interpolated path gains are not displayed.



Note

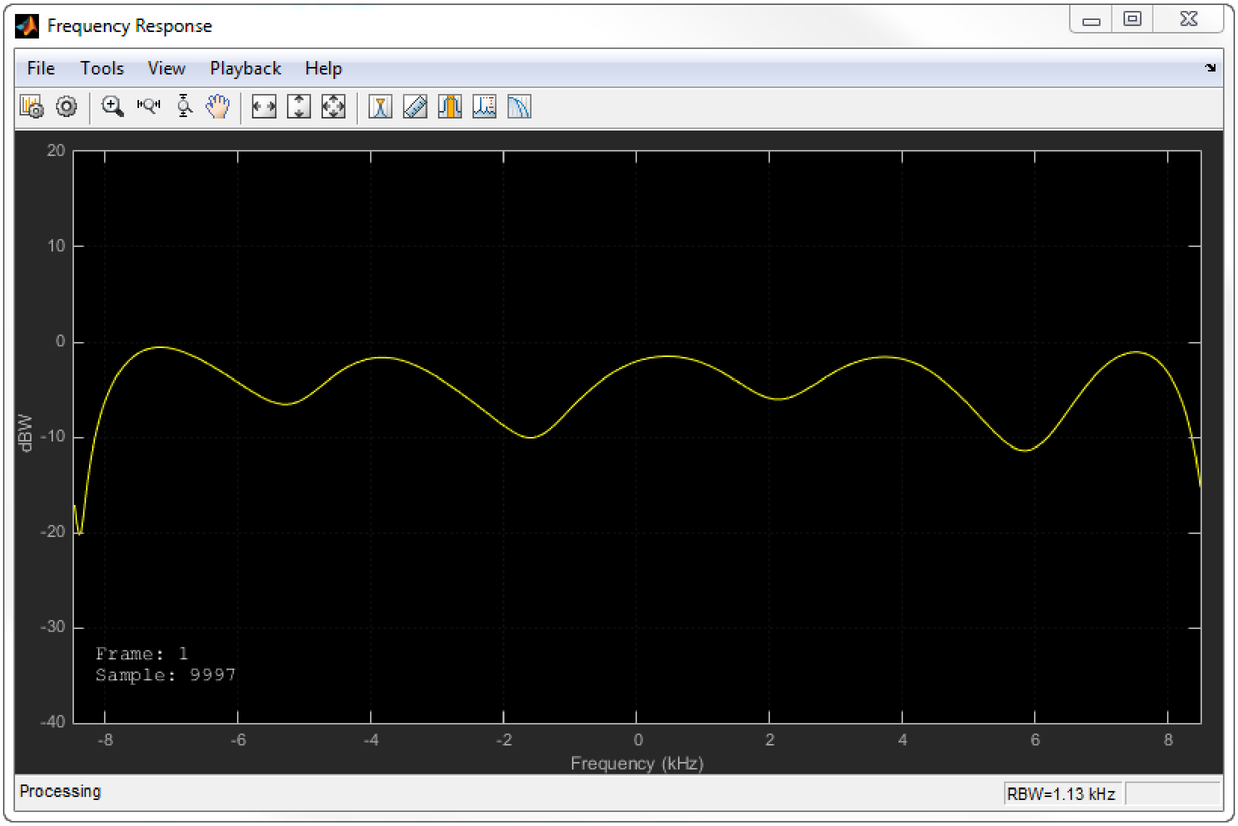
- The displayed and specified path gain locations can differ by as much as 5% of the input sample time.
- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Reduce Updates to Improve Performance** menu item. Reducing the percentage of samples to display and the enabling reduced updates will speed up the rendering of the impulse response.
- After the impulse response plots are manually closed, the `step` call for the Rayleigh channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

Frequency Response

The frequency response plot displays the Rayleigh channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. The frequency response plot shares the same toolbar and menus as the System object it was based on, `dsp.SpectrumAnalyzer`. The default parameter settings are shown below. These parameters can be changed from their default values by using the **View > Spectrum Settings** menu.

Parameter	Value
`Window`	`Rectangular`
`WindowLength`	Channel filter length
`FFTLength`	512
`PowerUnits`	`dBW`
`YLimits`	Based on `NormalizePathGains` and `AveragePathGains` properties

The frequency response plot for a frequency selective channel is shown.

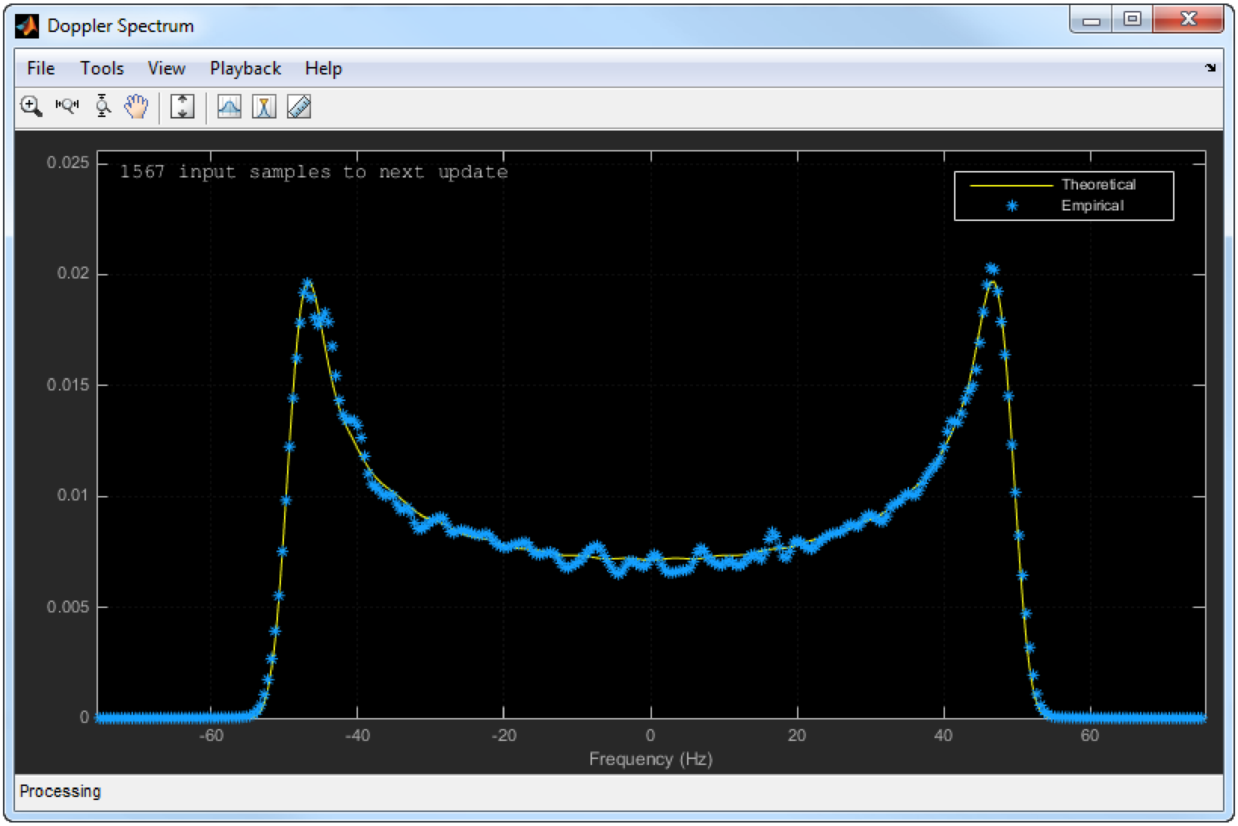


Note

- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Reduce Plot Rate to Improve Performance** menu item. Reducing the percentage of samples to display and the enabling reduced updates will speed up the rendering of the frequency response.
- After the frequency response plots are manually closed, the `step` call for the Rayleigh channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

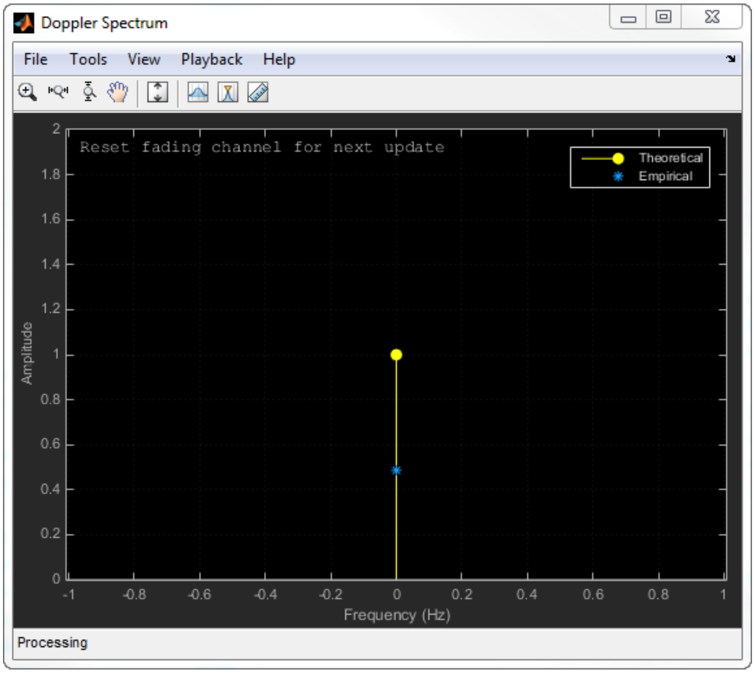
Doppler Spectrum

The Doppler spectrum plot displays both the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a yellow line for the case of non-static channels and as a yellow point for static channels, while the empirical data is shown in blue. There is an internal buffer which must be completely filled with filtered Gaussian samples before the empirical plot is updated. The empirical plot is the running mean of the spectrum calculated from each full buffer. For non-static channels, the number of input samples needed before the next update is displayed in the upper left hand corner. The samples needed is a function of the sample rate and the maximum Doppler shift. For static channels, the text `Reset fading channel for next update` is displayed.



1567 input samples to next update

Processing



Note

- After the Doppler spectrum plots are manually closed, the `step` call for the Rayleigh channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

Examples

Produce the Same Outputs Using Two Different Random Number Generation Methods

The Rayleigh Channel System object™ has two methods for random number generation. You can use the current global stream or the mt19937ar algorithm with a specified seed. By interacting with the global stream, the object can produce the same outputs from the two methods.

Create a PSK Modulator System object to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
channelInput = pskModulator(randi([0 pskModulator.ModulationOrder-1],1024,1));
```

Create a Rayleigh channel System object.

```
rayChan = comm.RayleighChannel(...
    'SampleRate',10e3, ...
    'PathDelays',[0 1.5e-4], ...
    'AveragePathGains',[2 3], ...
    'NormalizePathGains',true, ...
    'MaximumDopplerShift',30, ...
    'DopplerSpectrum',{doppler('Gaussian',0.6),doppler('Flat')}, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',22, ...
    'PathGainsOutputPort',true);
```

Filter the modulated data using the Rayleigh channel System object, `rayChan`.

```
[chanOut1,pathGains1] = rayChan(channelInput);
```

Use global stream for random number generation.

```
release(rayChan);
rayChan.RandomStream = 'Global stream';
```

Set the global stream to have the same seed that was specified above.

```
rng(22)
```

Filter the modulated data using `rayChan` for the second time.

```
[chanOut2,pathGains2] = rayChan(channelInput);
```

Verify that the channel and path gain outputs are the same for each of the two methods.

```
isequal(chanOut1,chanOut2)
```

```
ans = logical
   1
```

```
isequal(pathGains1,pathGains2)
```

```
ans = logical
   1
```

Display Impulse and Frequency Responses of a Rayleigh Channel

This example shows how to create a frequency selective Rayleigh channel and display its impulse and frequency responses.

Set the sample rate to 3.84 MHz and specify path delays and gains using ITU pedestrian B channel parameters. Set the maximum Doppler shift to 50 Hz.

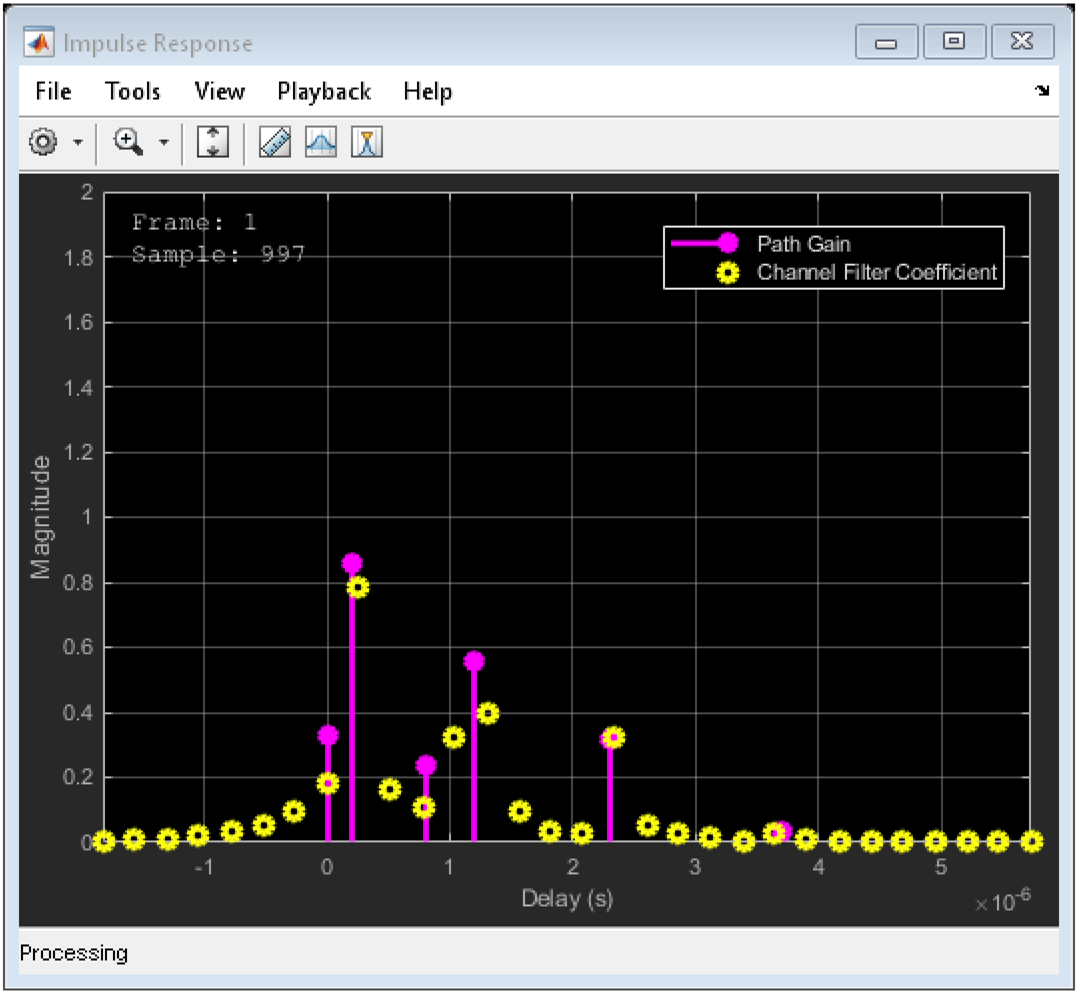
```
fs = 3.84e6;                                    % Hz
pathDelays = [0 200 800 1200 2300 3700]*1e-9;   % sec
avgPathGains = [0 -0.9 -4.9 -8 -7.8 -23.9];     % dB
fD = 50;                                        % Hz
```

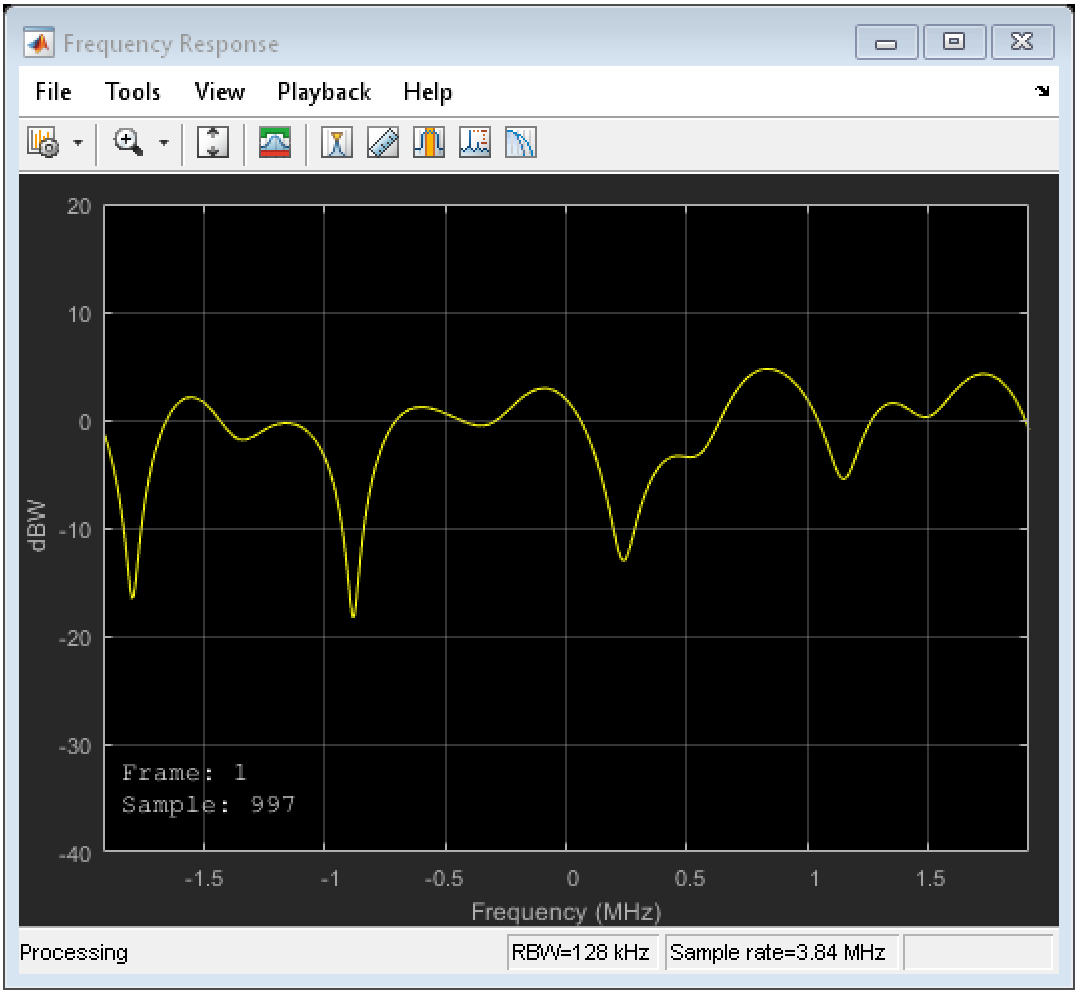
Create a Rayleigh channel System object with the previously defined parameters and set the `Visualization` property to `Impulse and frequency responses` using name-value pairs.

```
rchan = comm.RayleighChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the Rayleigh channel. The impulse response plot allows you to easily identify the individual paths and their corresponding filter coefficients. The frequency selective nature of the pedestrian B channel is shown by the frequency response plot.

```
x = randi([0 1],1000,1);
y = rchan(x);
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 997
Path Gain
Channel Filter Coefficient
Magnitude
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
-1
0
1
2
3
4
5
Delay (s)
$\times 10^{-6}$
Processing



Frame: 1
Sample: 997

RBW=128 kHz

Sample rate=3.84 MHz

Generate a Rayleigh Channel Using Sum-of-Sinusoids Technique

This example shows how to generate a Rayleigh channel using the sum-of-sinusoids technique.

Set the channel parameters.

```
fs = 1000;                % Sample rate (Hz)
pathDelays = [0 2.5e-3];  % Path delays (s)
pathPower = [0 -6];       % Path power (dB)
fD = 5;                   % Maximum Doppler shift (Hz)
numSamples = 1000;        % Number of samples
```

Create a Rayleigh channel object using a name-value pair to set the `FadingTechnique` property to `Sum of sinusoids`.

```
rchan = comm.RayleighChannel('SampleRate',fs, ...
'PathDelays',pathDelays,'AveragePathGains',pathPower, ...
'MaximumDopplerShift',fD,'FadingTechnique','Sum of sinusoids')
```

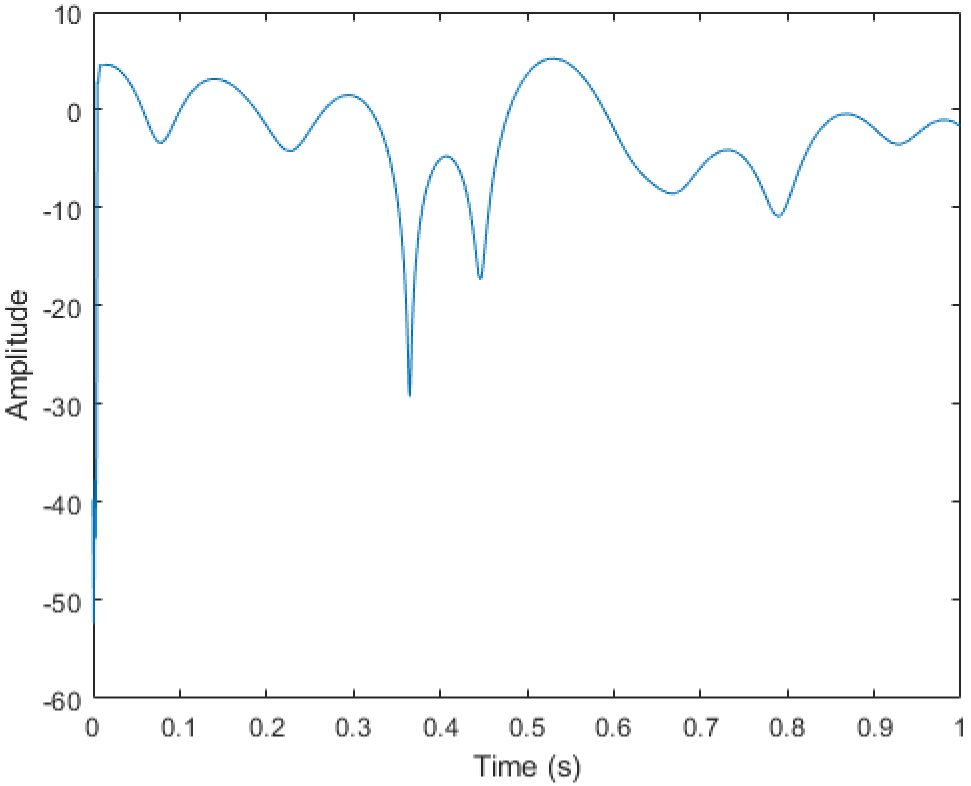
```
rchan = 
  comm.RayleighChannel with properties:

             SampleRate: 1000
             PathDelays: [0 0.0025]
       AveragePathGains: [0 -6]
     NormalizePathGains: true
    MaximumDopplerShift: 5
        DopplerSpectrum: [1x1 struct]

  Show all properties
```

Pass an all-ones vector through the Rayleigh channel.

```
y = rchan(ones(numSamples,1));
```

Plot the magnitude of the Rayleigh channel output.

```
t = (0:numSamples-1)'/fs;
plot(t,20*log10(abs(y)))
xlabel('Time (s)')
ylabel('Amplitude')
```



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.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. “Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms.” *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Algorithms

Cutoff Frequency Factor

The following information explains how the cutoff frequency factor, f_c, is determined for different Doppler spectrum types:

- For any Doppler spectrum type other than Gaussian and BiGaussian, f_c equals `1`.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation•sqrt(2•log(2))`.
- For a `doppler('BiGaussian')` spectrum type:

  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)•sqrt(2•log(2))`.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)•sqrt(2•log(2))`.
  - In all other cases, f_c equals `1`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AWGNChannel` | `comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.RicianChannel`

Introduced in R2013b

info

System object: comm.RayleighChannel
Package: comm

Display information about the `RayleighChannel` 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.

reset

System object: comm.RayleighChannel
Package: comm

Reset states of the `RayleighChannel` object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RayleighChannel` 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.

step

System object: comm.RayleighChannel
Package: comm

Filter input signal through multipath Rayleigh fading channel

Syntax

```
Y = step(H,X)
[Y,PATHGAINS] = step(H,X)
___ = step(H,X,INITIALTIME)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` filters input signal `X` through a multipath Rayleigh fading channel and returns the result in `Y`. Both the input `X` and the output signal `Y` are of size N_s–by–1, where N_s represents the number of samples. The input `X` can be of double- or single-precision data type with real or complex values. `Y` contains complex values with same precision as input signal.

`[Y,PATHGAINS] = step(H,X)` returns the channel path gains of the underlying Rayleigh 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, where N_p represents the number of paths, i.e., the length of the PathDelays property value of `H`. `PATHGAINS` contains complex values with same precision as input signal.

`___ = step(H,X,INITIALTIME)` passes data through the Rayleigh channel beginning at `INITIALTIME`, where `INITIALTIME` is a nonnegative real scalar measured in seconds. This syntax applies when the FadingTechnique property of `H` is set to `Sum of sinusoids` and the InitialTimeSource property of `H` is set to `Input port`. The `INITIALTIME` must be greater than the last frame end time. When `INITIALTIME` is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RBDSWaveformGenerator System object

Package: comm

Generate RDS/RBDS waveform

Description

The `comm.RBDSWaveformGenerator` System object generates configurable standard-compliant baseband RDS/RBDS waveforms in MATLAB. RDS/RBDS waveforms supplement FM radio stations with additional textual information, such as song title, artist name, and station description. The RDS/RBDS signal lies in the 57-kHz band of the baseband FM radio signal.

Use this object to generate a waveform containing RadioText Plus (RT+) information and register a custom encoding implementation for an Open Data Application (ODA). You can also specify the time, data, and the program type. The object supports short, scrolling 8-character text, and longer 32-character or 64-character text.

To generate baseband RDS/RBDS waveforms:

1. Create a `comm.RBDSWaveformGenerator` object and set the properties of the object.
2. Call `step` to generate the waveform.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`rbdsgen = comm.RBDSWaveformGenerator` creates an RDS/RBDS waveform generator object, `rbdsgen`, using the default properties.

`rbdsgen = comm.RBDSWaveformGenerator(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
rbdsgen = comm.RBDSWaveformGenerator('GroupsPerFrame',20,'SamplesPerSymbol',10,...
                                      'SendRadioTextPlus',true);
```

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

SamplesPerSymbol — Number of samples per symbol

`10` (default) | positive even integer

Number of samples per symbol (bit), specified as a positive even integer. Half of the samples represent one amplitude level of Manchester coding. The other half of the samples represent the opposite level.

GroupsPerFrame — Number of groups per output frame

`10` (default) | scalar integer

Number of groups per output frame, specified as a scalar integer. Each group is 104 symbols (bits) long.

RadioText — Long text conveyed with type 2A groups

`'Long text'` (default) | character vector

Radio text conveyed with type 2A groups, specified as a character vector that is up to 64 characters long. The object transmits the specified text four characters at a time, using type 2A groups.

Tunable: Yes

ProgramServiceName — Label of the program service

`'ShortTxt'` (default) | character vector

Label of the program service, specified as a character vector that is up to eight characters long. This information is conveyed as a short text with type 0A groups, two characters at a time.

Tunable: Yes

ProgramIdentificationCode — Program identification code

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] (default) | 16-bit row vector

Program identification (PI) code, specified as a 16-bit row vector. In North America, the PI code conveys the call letters of the station. Example call letters include 'WABC' and 'KXYZ'.

To generate North American PI codes for a station's call letters, use the callLettersToPICode method.

ProgramType — Program type

'No program type or undefined' (default) | character vector

Program type, specified as a character vector containing one of the 31 values allowed by the RDS/RBDS standard. For a list of program types that the RDS/RBDS standard allows in North America, see [1].

Tunable: Yes

ProgramTypeName — Program type name

'' (default) | character vector

Program type name, specified as a character vector that is up to eight characters long. This text further characterizes the program type, such as 'Football' for the program type 'Sports'. The object conveys the program type name using type 10A groups. If this property is empty, then no 10A groups are generated.

Tunable: Yes

SendDateTime — Option to advertise date and time

false (default) | true

Option to advertise the date and time, specified as either false or true. When you set this property to true, one 4A group is periodically generated every 685 groups (once a minute).

AlternativeFrequencies — Alternative frequencies

[] (default) | numeric row vector

Alternative frequencies, specified as a numeric row vector in MHz. This information is conveyed with type 0A groups. It indicates other transmitters broadcasting the same program in the same or adjacent reception areas. With this information, receivers can switch to a different frequency with better reception.

SendRadioTextPlus — Option to send RT+ information
`false` (default) | `true`

Option to transmit RadioText Plus (RT+) information, specified as a scalar logical. When you set this property to `true`, the RT+ ODA information is advertised with type 3A groups. In addition, the RT+ content types, specified in `RadioTextType1`, `RadioTextType2`, and the two RT+ substrings indexed by `RadioTextIndices` are conveyed with the open-format type 11A group.

RadioTextType1 — Content type of first RT+ substring
`'Item.Artist'` (default) | character vector

Content type of the first RT+ substring, specified as a character vector. Allowed values are the class names specified in the RT+ standard. For more details, see [2].

Tunable: Yes

RadioTextType2 — Content type of second RT+ substring
`'Item.Title'` (default) | character vector

Content type of the second RT+ substring, specified as a character vector. Allowed values are the class names specified in the RT+ standard. For more details, see [2].

Tunable: Yes

RadioTextIndices — Start and end indices of RT+ substrings
`[1 2; 3 4]` (default) | 2-by-2 matrix of positive integers

Start and end indices of RT+ substrings, specified as a 2-by-2 matrix of positive integers. The first column indexes the beginning of each RT+ substring. The second column indexes the end of each substring.

Tunable: Yes

Methods

callLettersToPICode	Convert North-American call letters to binary PI code
registerODA	Register a custom encoding implementation for an ODA
reset	Reset states of RBDS waveform generator object
step	Generate RDS/RBDS waveform

Examples

Generate a Basic RBDS Waveform

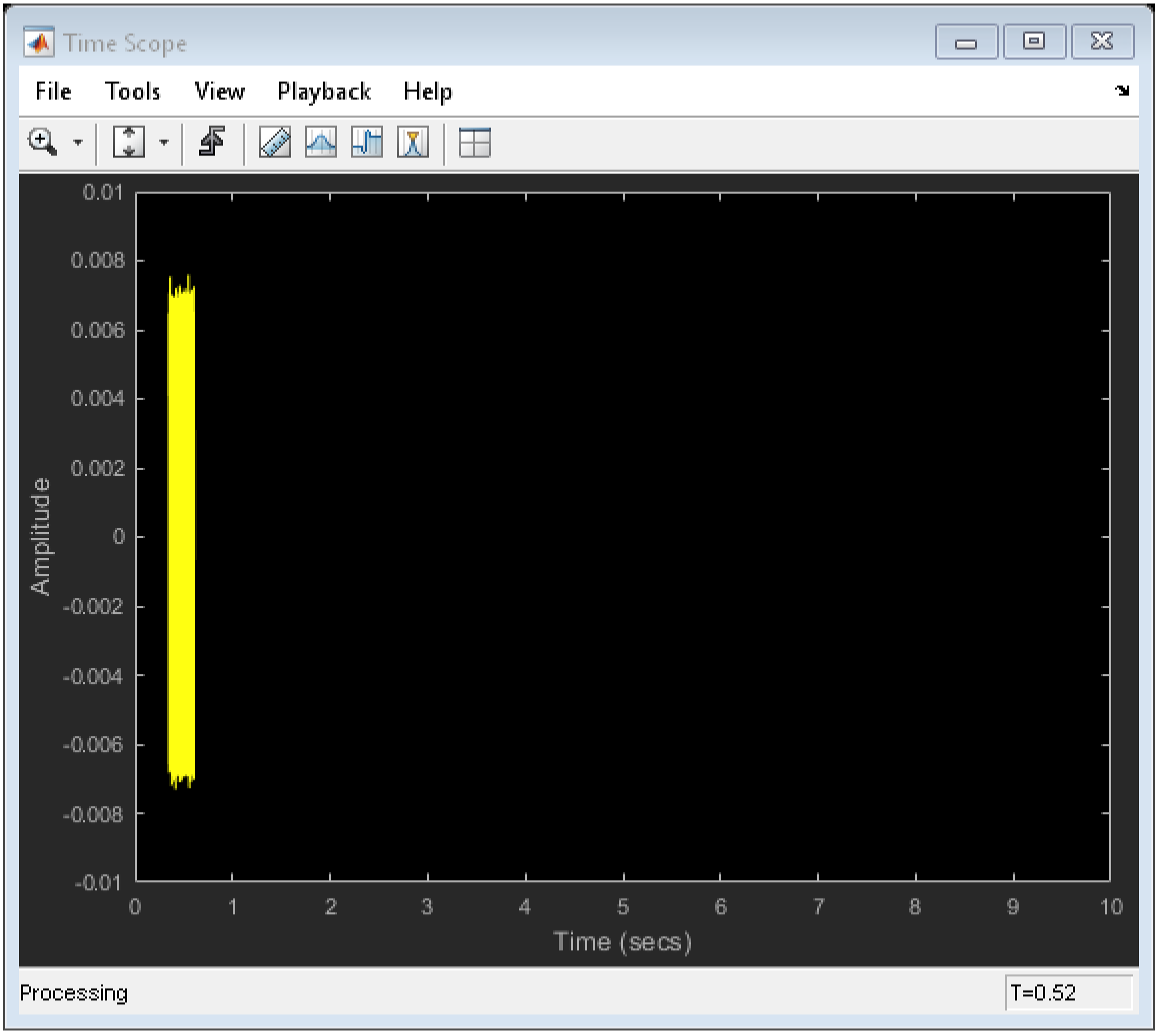
Generate a basic RBDS waveform, FM modulate the waveform with an audio signal, and then demodulate the waveform.

Each frame of the RBDS waveform contains 19 groups, with a group length of 104 bits (symbols) each. Set the number of samples per RBDS symbol to 10. Therefore, the number of samples in each frame of RBDS waveform is 104 x 10 x 19 = 19,760. According to the RBDS standard, the bit rate is 1187.5 Hz. So, the RBDS sample rate = 1187.5 x samples per RBDS symbol. Set the audio frame rate to 40 x 1187.5 = 47,500.

```
groupLen = 104;
sps = 10;
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRa
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sp

fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = dsp.TimeScope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the audio input and generate the RBDS waveform. FM modulate the stereo audio with the RBDS waveform, add noise, and FM demodulate the audio and RBDS waveforms. View the demodulated RBDS waveform in the time scope.

```
for idx = 1:7
    input = afr();                          % get current audio input
    rbdsWave = rbds();                      % generate RBDS info at the same configured
    yFM = fmMod([input input], rbdsWave);  % FM modulate stereo audio with RBDS info
    rcv = awgn(yFM, 40);                    % add noise
    [audioRcv, rbdsRcv] = fmDemod(rcv);    % FM demodulate the audio and RBDS waveforms
    scope(rbdsRcv);
end
```



Generate RBDS Waveform with RadioText Plus Information

Create a `comm.RBDSWaveformGenerator` System object™ with 20 groups per frame and 10 samples per symbol. Add the Radio Text plus (RT+) information, such as artist name and song, title, to the waveform. Indicate the start and end of the RT+ substrings by using the RadioTextIndices property.

```
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',20,'SamplesPerSymbol',10,...
    'SendRadioTextPlus', true);
rbds.RadioText = 'MyArtist - MySongTitle';
rbds.RadioTextType1 = 'Item.Artist';
rbds.RadioTextType2 = 'Item.Title';
rbds.RadioTextIndices = [1 8; 12 22];
for idx = 1:10
    rbds.step();
end
```

Register a Custom Encoding Implementation

Register a custom encoding implementation for an Open Data Application (ODA) by using the registerODA method of the comm.RBDSWaveformGenerator System object™. Set the ODA ID to 'CD46', which is the ID for the traffic message channel. The allocated group type is 8A.

```
rbds = comm.RBDSWaveformGenerator();
odaID = 'CD46';
allocatedGroupType = '8A';
```

This example uses the following templates as a starting point for custom encoding implementation.

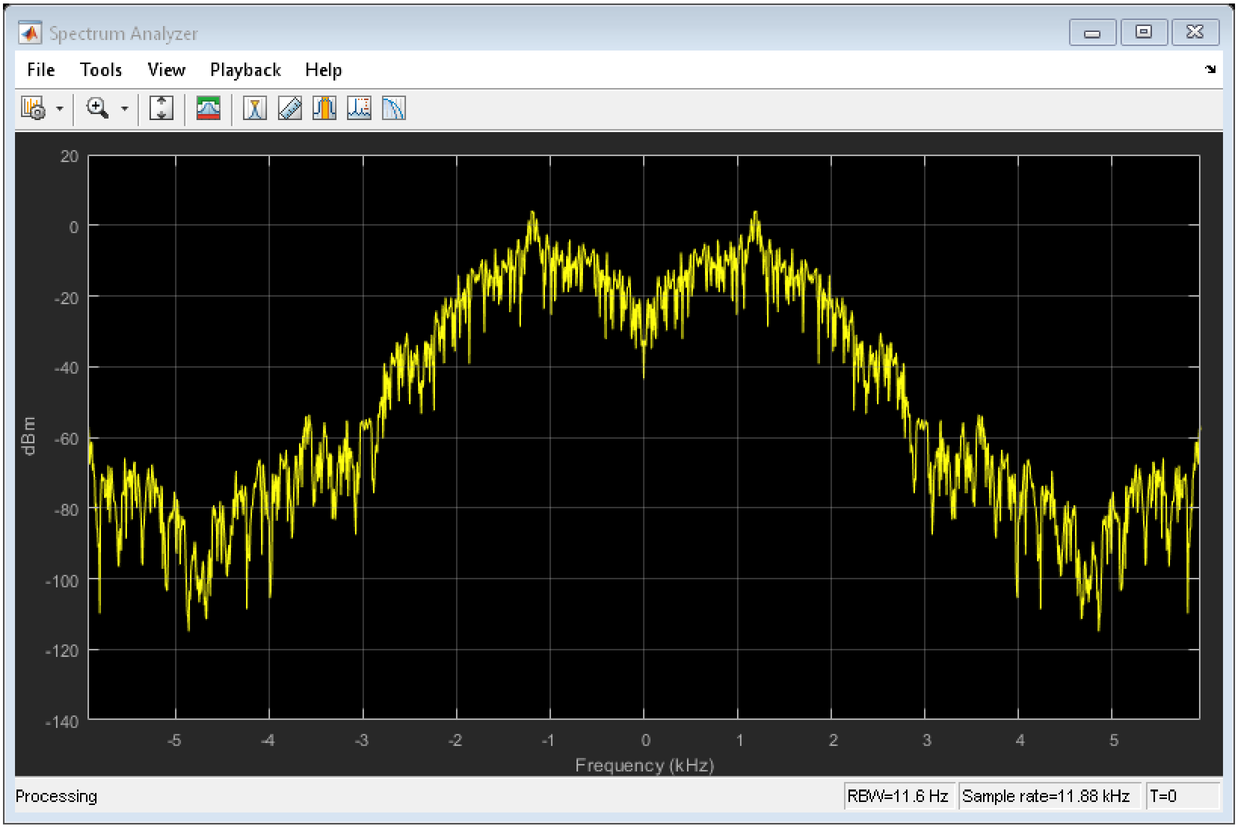
```
mainProcessingFcn = @CustomODAEncodingMain;
fcn3A             = @CustomODAEncoding3A;
registerODA(rbds,odaID,allocatedGroupType,mainProcessingFcn,fcn3A);
s = info(rbds);
s.ODAMap
```

```
ans = 2x1 struct array with fields:
    ID
    GroupType
    FunctionMain
    Function3A
```

Configure RBDS Waveforms with Date and Time Information

Generate RBDS waveform with date and time information, the program type, and alternative frequencies. The `comm.RBDSWaveformGenerator` object uses type 4A groups for date and time information, type 10A groups for the program type information, and type 0A groups for alternative frequencies. View the waveform in a spectrum analyzer.

```
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',1000);
scope = dsp.SpectrumAnalyzer('SampleRate',1187.5*rbds.SamplesPerSymbol,'YLimits',[-140
rbds.SendDateTime = true;          % send type 4A groups
rbds.ProgramType = 'Sports';
rbds.ProgramTypeName = 'Football'; % send type 10A groups
rbds.AlternativeFrequencies = [99.1 102.5]; % info sent in type 0A groups
wave = rbds.step();
scope(wave)
```



Algorithms

`comm.RBDSWaveformGenerator` generates waveforms according to the RDS/RBDS standard [1]. The RDS/RBDS standard consists of three layers: physical layer, data-link layer, and session and application layer.

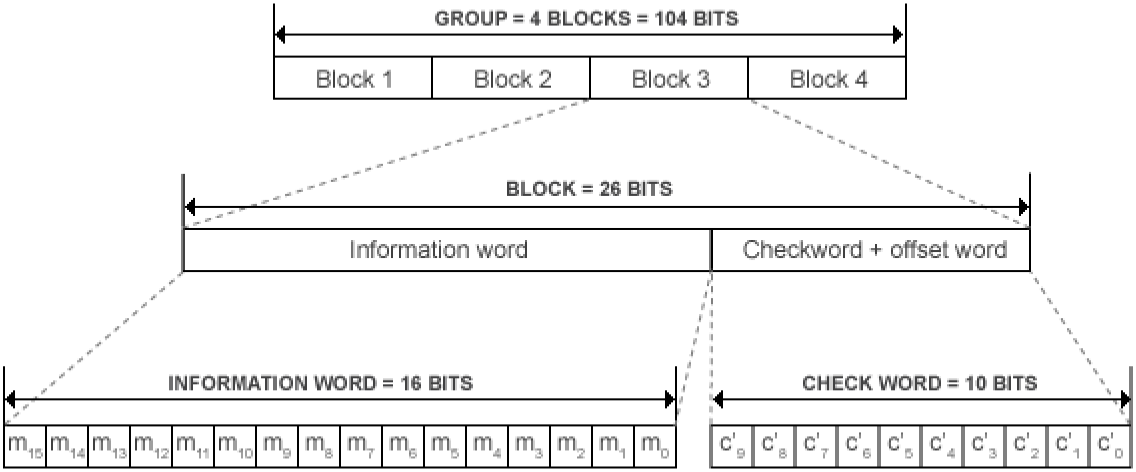
Physical Layer

The physical layer (first layer) converts the data-link bits to an analog waveform by conducting differential encoding and biphase symbol encoding (Manchester encoding) and pulse-shaping filtering.

Data-Link Layer

The data-link layer (second layer) performs (26,16) cyclic encoding shortened from (341,331) encoding [1]. The second layer is responsible for error detection, error correction, and the establishment of group-level synchronization. Each group of RDS/RBDS frames contains four blocks of 26 bits (that is 104 bits) each. Each block contains an information word and a check word. Each information word contains 16 bits, and each check word contains 10 bits.

Here is the baseband coding structure for the RDS/RBDS waveform. For more details, see [1].


GROUP = 4 BLOCKS = 104 BITS
Block 1
Block 2
Block 3
Block 4
BLOCK = 26 BITS
Information word
Checkword + offset word
INFORMATION WORD = 16 BITS
m_{15} m_{14} m_{13} m_{12} m_{11} m_{10} m_9 m_8 m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0
CHECK WORD = 10 BITS
c'_9 c'_8 c'_7 c'_6 c'_5 c'_4 c'_3 c'_2 c'_1 c'_0

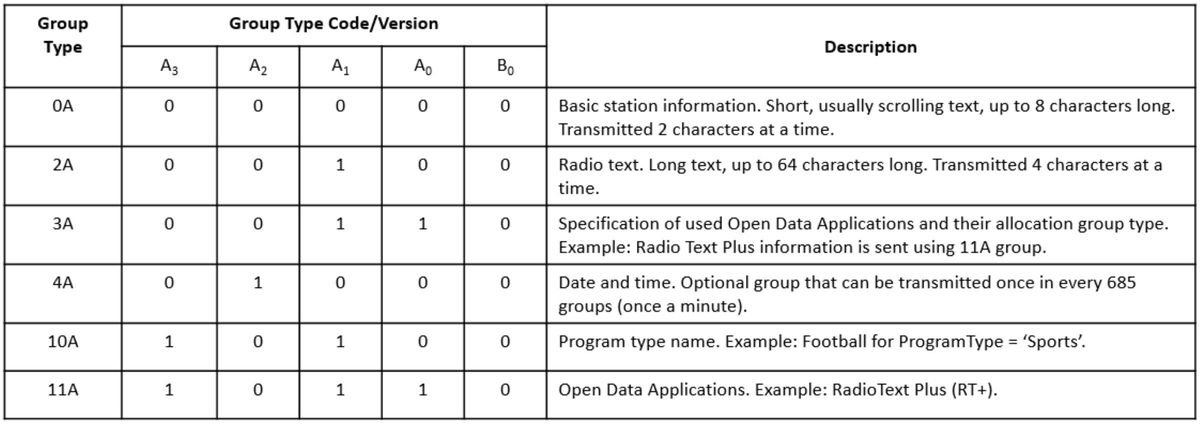
For each block, a unique offset word is modulo-2 added to the checkword bits. The added offset word provides a group and block synchronization system in the receiver (decoder). Because the addition of the offset is reversible in the decoder, the normal additive error-correcting and detecting properties of the basic code are unaffected.

Session and Application Layer

The first block in every group contains a program identification (PI) code. The first four bits of the second block of every group are allocated to a four-bit code. This code specifies the application of the group. Groups are referred to as types 0–15 according to the binary weighting $A_3 = 8$, $A_2 = 4$, $A_1 = 2$, $A_0 = 1$. The fifth bit of the second block, B_0, defines the version of the application. If $B_0 = 0$, the version of the group is A. The PI code in this version is inserted into block 1 only. Example group types include 0A, 1A, 2A, 3A, and 4A.

The Program Type code and Traffic Program Identification (PI) occupy fixed locations in block 2 of every group.

Group Types



Group Type	Group Type Code/Version					Description
	A_3	A_2	A_1	A_0	B_0	
0A	0	0	0	0	0	Basic station information. Short, usually scrolling text, up to 8 characters long. Transmitted 2 characters at a time.
2A	0	0	1	0	0	Radio text. Long text, up to 64 characters long. Transmitted 4 characters at a time.
3A	0	0	1	1	0	Specification of used Open Data Applications and their allocation group type. Example: Radio Text Plus information is sent using 11A group.
4A	0	1	0	0	0	Date and time. Optional group that can be transmitted once in every 685 groups (once a minute).
10A	1	0	1	0	0	Program type name. Example: Football for ProgramType = ‘Sports’.
11A	1	0	1	1	0	Open Data Applications. Example: RadioText Plus (RT+).

References

[1] National Radio Systems Committee. *United States RBDS Standard: Specification of the radio broadcast data system (RBDS)*. Electronic Industries Association and National Association of Broadcasters. April 9, 1998.

[2] Westdeutscher Rundfunk WDR, Nokia, and Institut für Rundfunktechnik IRT. *RadioText Plus (RT+) Specification, Version 2.1.* 2006.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

In addition, the following limitations apply when you generate code that contains this System object or when you use this object in a MATLAB function block.

- The group type 4A cannot be transmitted in the generated code.
- The `registerODA` method is not supported.
- The `ProgramType` property is not tunable.

See Also

System Objects

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator`

Introduced in R2017a

callLettersToPICode

System object: comm.RBDSWaveformGenerator
Package: comm

Convert North-American call letters to binary PI code

Syntax

```
picode = callLettersToPICode(rbdsgen, callLetters)
```

Description

`picode = callLettersToPICode(rbdsgen, callLetters)` returns the 16-bit program identification (PI) code that corresponds to `callLetters`. Acceptable call letter formats are 3-character or 4-character vectors beginning with `'K'` or `'W'`.

Introduced in R2017a

registerODA

System object: comm.RBDSWaveformGenerator
Package: comm

Register a custom encoding implementation for an ODA

Syntax

```
registerODA(rbdsgen,odaID,group,handleMain,handle3A)
```

Description

`registerODA(rbdsgen,odaID,group,handleMain,handle3A)` associates the Open Data Application (ODA) specified by the hexadecimal ID `odaID`, with the type `group` groups generated by `rbdsgen`. The four 16-bit information words of these groups are generated by the function handle `handleMain`. The third information word of type 3A groups, which is ODA-specific, is generated by the function handle `handle3A`.

Introduced in R2017a

reset

System object: comm.RBDSWaveformGenerator
Package: comm

Reset states of RBDS waveform generator object

Syntax

```
reset(rbdsgen)
```

Description

`reset(rbdsgen)` resets the states of the `RBDSWaveformGenerator` object, `rbdsgen`.

Introduced in R2017a

step

System object: comm.RBDSWaveformGenerator
Package: comm

Generate RDS/RBDS waveform

Syntax

```
y = step(rbdsgen)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(rbdsgen)` outputs a frame of the baseband RDS/RBDS waveform in column vector `y`. The waveform contains the number of 104-bit groups, specified in the `GroupsPerFrame` property of the object. Each symbol is oversampled according to the `SamplesPerSymbol` property. Thus, the output length is `SamplesPerSymbol` × 104 × `GroupsPerFrame` samples. The object uses an internal scheduler to determine the order and frequency of the transmitted group types.

Note `rbdsgen` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2017a

comm.RectangularQAMDemodulator System object

Package: comm

Demodulate using rectangular QAM signal constellation

Description

The `RectangularQAMDemodulator` object demodulates a signal that was modulated using quadrature amplitude modulation with a constellation on a rectangular lattice.

To demodulate a signal that was modulated using quadrature amplitude modulation:

1. Define and set up your rectangular QAM demodulator object. See "Construction" on page 3-1357.
2. Call `step` to demodulate the signal according to the properties of `comm.RectangularQAMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0) 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` on page 3-0) 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `true` and the `DecisionMethod` on page 3-0 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` on page 3-0 property to `true`, the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the `VarianceSource` on page 3-0 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` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 property to `Hard decision`. In this case, when the `OutputDataType` on page 3-0 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 “Fixed-Point Support for MATLAB System Objects in DSP System Toolbox” (DSP System Toolbox).

`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 on page 3-0 property to false, or when you set the BitOutput property to true and the DecisionMethod on page 3-0 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 on page 3-0 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 on page 3-0 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 on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 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 on page 3-0 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 on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 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` on page 3-0 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` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 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` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 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` on page 3-0 property to false, when you set the `BitOutput` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 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 on page 3-0 property to false or when you set the SumDataType on page 3-0 property Custom.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using rectangular QAM method

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate and Demodulate Data Using 16-QAM

This example shows how to modulate and demodulate data using 16-QAM modulation.

Create rectangular QAM modulator and demodulator objects with the modulation order set to 16.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',16);
hDemod = comm.RectangularQAMDemodulator('ModulationOrder',16);
```

Create an AWGN channel object.

```
hAWGN = comm.AWGNChannel('EbNo',2,'BitsPerSymbol',4);
```

To track the number of errors, create an error rate counter object.

```
hError = comm.ErrorRate;
```

Set the random number generator to its default state to ensure repeatability.

```
rng default
```

Generate random data symbols and apply 16-QAM modulation.

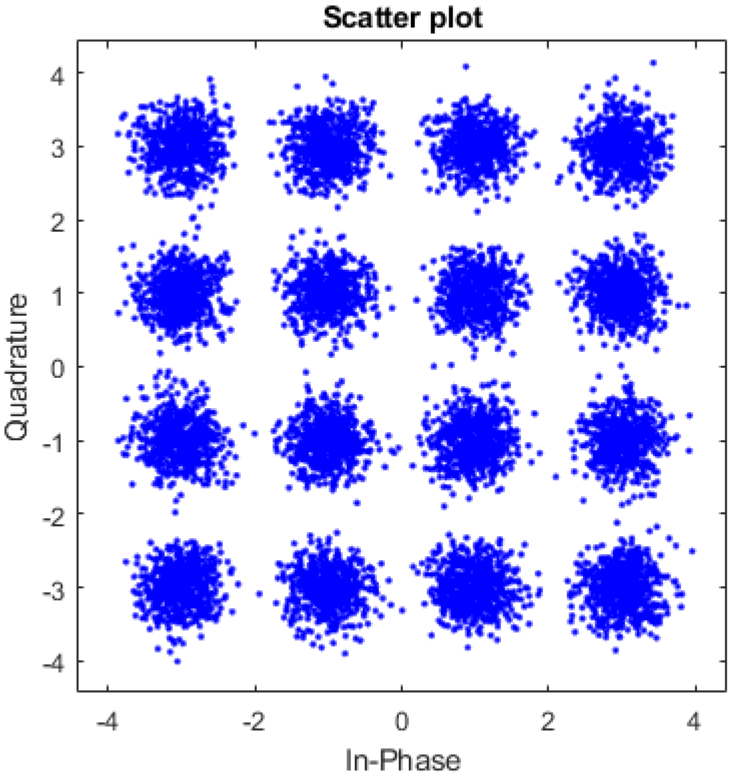
```
dataIn = randi([0 15],10000,1);
txSig = step(hMod,dataIn);
```

Pass the modulated data through the AWGN channel.

```
rxSig = step(hAWGN,txSig);
```

Display the noisy constellation using the `scatterplot` function.

```
scatterplot(rxSig)
```



Demodulate the received data symbols.

```
dataOut = step(hDemod,rxSig);
```

Using the `step` function of `hError`, calculate the error statistics.

```
errorStats = step(hError,dataIn,dataOut);
```

Display the error statistics, where you can observe that 8 errors were recorded in 10,000 transmitted symbols.

```
fprintf('\nError rate = %f\nNumber of errors = %d\nNumber of symbols = %d\n', ...
errorStats)
```

```
Error rate = 0.000800
Number of errors = 8
Number of symbols = 10000
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMDemodulator` | `comm.RectangularQAMModulator`

Introduced in R2012a

constellation

System object: comm.RectangularQAMDemodulator
Package: comm

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

Plot Rectangular QAM Reference Constellation

Create a rectangular QAM modulator.

```
mod = comm.RectangularQAMModulator;
```

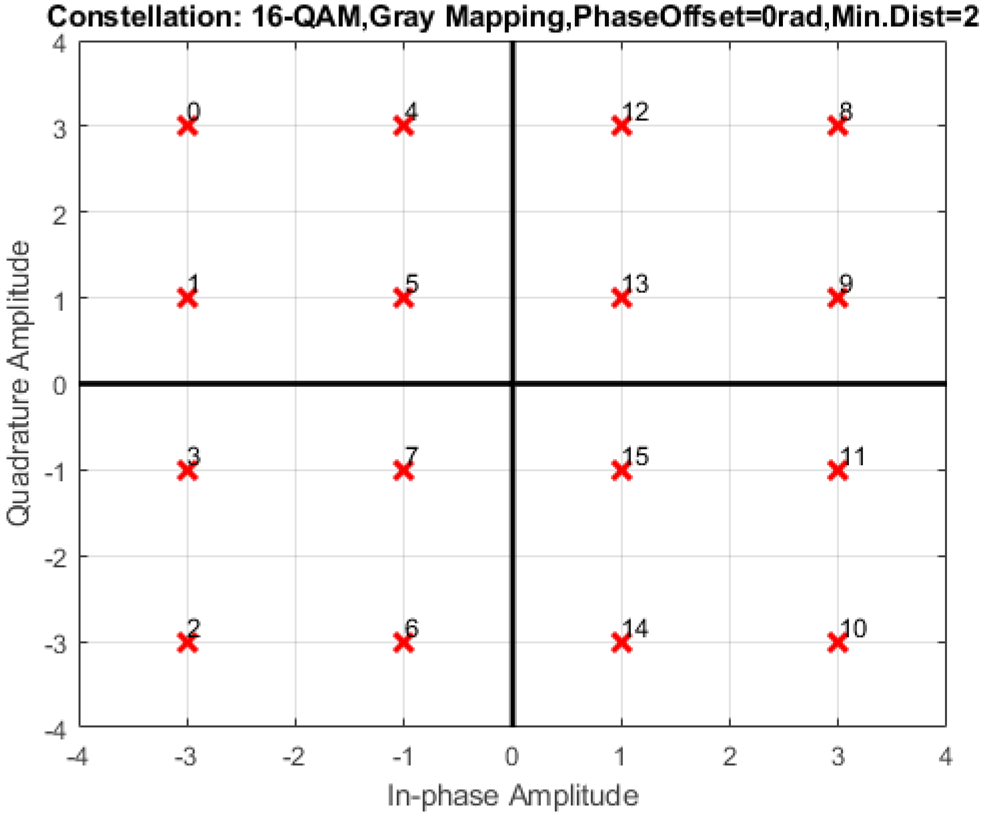
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 16×1 complex

  -3.0000 + 3.0000i
  -3.0000 + 1.0000i
  -3.0000 - 1.0000i
  -3.0000 - 3.0000i
  -1.0000 + 3.0000i
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
  -1.0000 - 3.0000i
   1.0000 + 3.0000i
   1.0000 + 1.0000i
       ⋮
```

Plot the constellation.

```
constellation(mod)
```

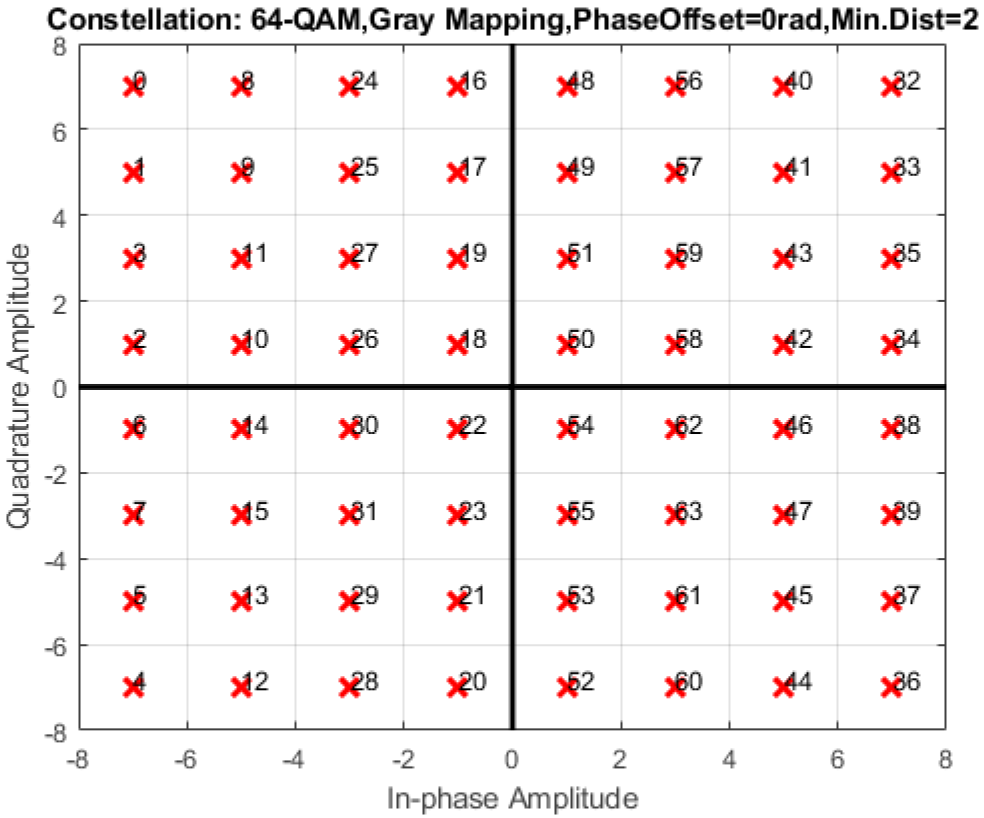


Create a QAM demodulator having a modulation order of 64.

```
demod = comm.RectangularQAMDemodulator(64);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



step

System object: comm.RectangularQAMDemodulator
Package: comm

Demodulate using rectangular QAM method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Modulate using rectangular QAM signal constellation

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.

To modulate a signal using quadrature amplitude modulation:

1. Define and set up your rectangular QAM modulator object. See “Construction” on page 3-1372.
2. Call `step` to modulate the signal according to the properties of `comm.RectangularQAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 on page 3-0). 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 on page 3-0) 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 on page 3-0 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 on page 3-0 . 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 on page 3-0 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 on page 3-0 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 on page 3-0 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` on page 3-0 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` on page 3-0 property to `Custom`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using rectangular QAM method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data with 64-QAM

This example shows how to modulate binary data with a 64-QAM System object and to view the resultant constellation.

Generate random binary data. As there are 6 bits/symbol in 64-QAM, the number of bits input to the modulator must be a multiple of 6.

```
data = randi([0 1],6000,1);
```

Create a 64-QAM modulator object that accepts binary input.

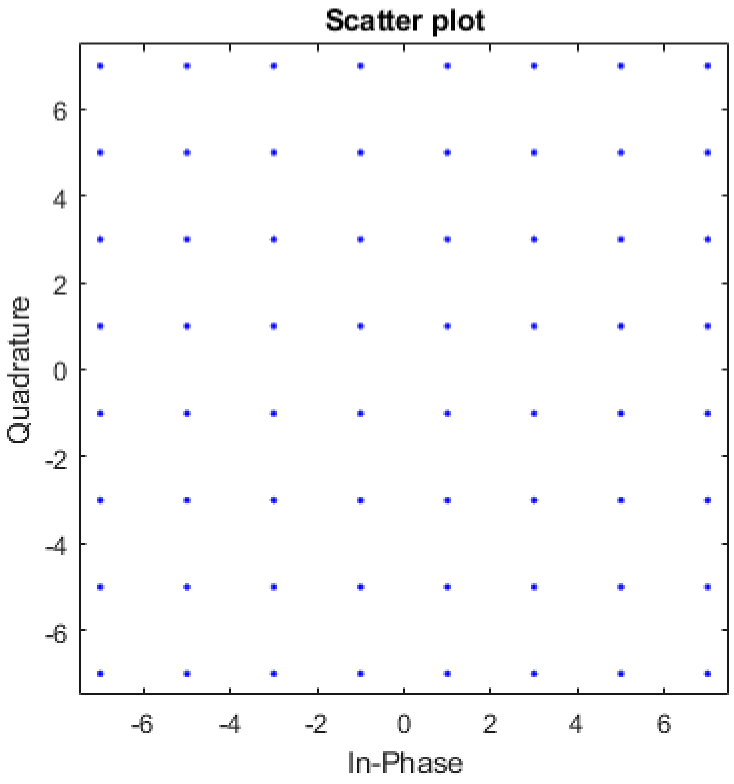
```
hMod = comm.RectangularQAMModulator('ModulationOrder',64,'BitInput',true);
```

Modulate the data using the `step` function.

```
dataMod = step(hMod,data);
```

Plot the modulated data using the `scatterplot` function.

```
scatterplot(dataMod)
```



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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMModulator` | `comm.RectangularQAMDemodulator`

Introduced in R2012a

constellation

System object: comm.RectangularQAMModulator
Package: comm

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

Plot Rectangular QAM Reference Constellation

Create a rectangular QAM modulator.

```
mod = comm.RectangularQAMModulator;
```

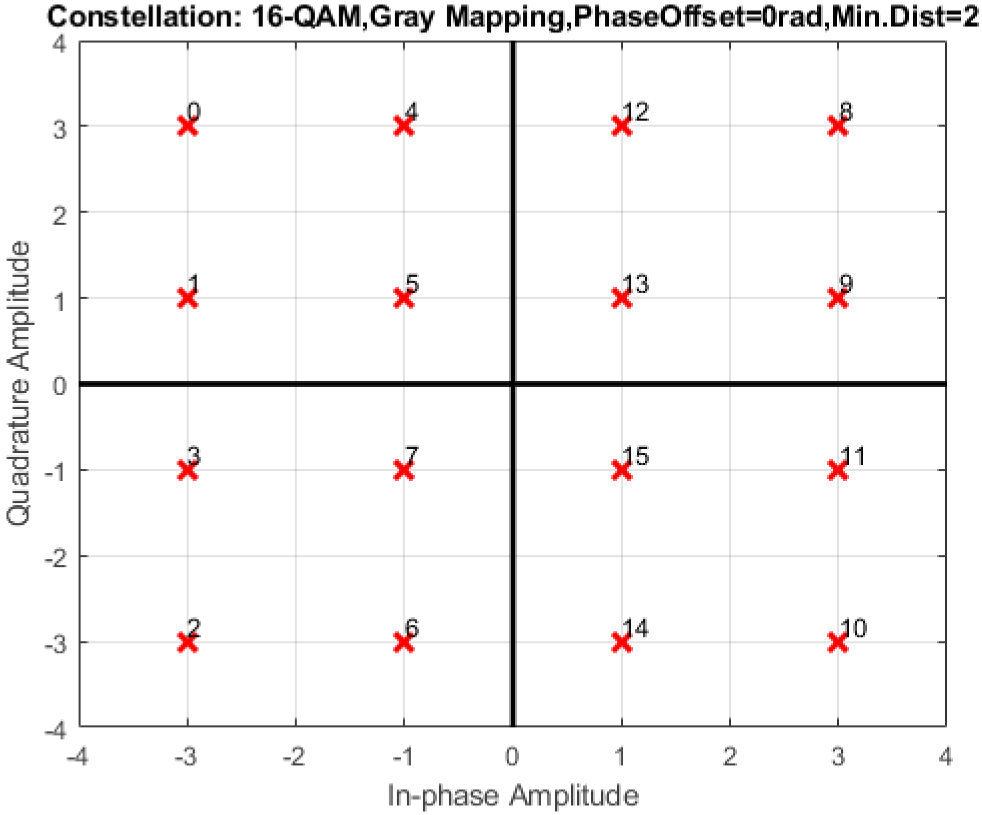
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 16×1 complex

  -3.0000 + 3.0000i
  -3.0000 + 1.0000i
  -3.0000 - 1.0000i
  -3.0000 - 3.0000i
  -1.0000 + 3.0000i
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
  -1.0000 - 3.0000i
   1.0000 + 3.0000i
   1.0000 + 1.0000i
        ⋮
```

Plot the constellation.

```
constellation(mod)
```

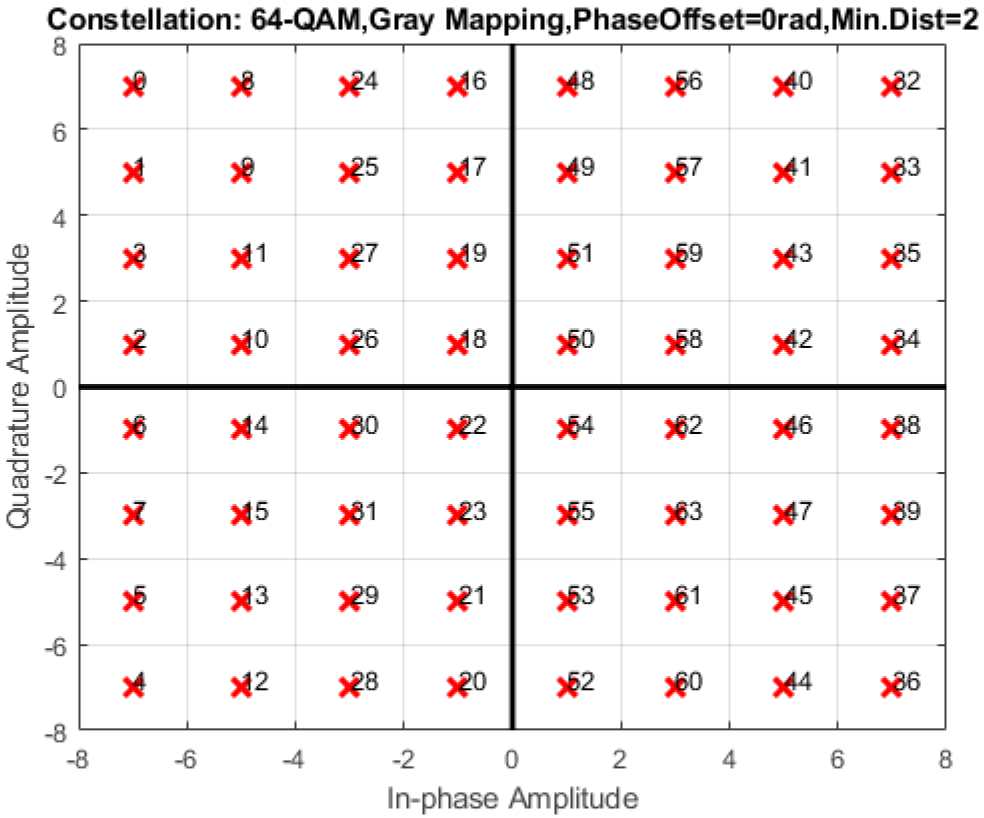


Create a QAM demodulator having a modulation order of 64.

```
demod = comm.RectangularQAMDemodulator(64);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



step

System object: comm.RectangularQAMModulator
Package: comm

Modulate using rectangular QAM method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Demodulate convolutionally encoded data mapped to rectangular QAM signal 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.

To demodulate convolutionally encoded data mapped to a rectangular QAM signal constellation:

1. Define and set up your rectangular QAM TCM demodulator object. See “Construction” on page 3-1383.
2. Call `step` to demodulate the signal according to the properties of `comm.RectangularQAMTCMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

reset	Reset states of the rectangular QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate and Demodulate Using Rectangular 16-QAM TCM

Modulate and demodulate data using 16-QAM TCM in an AWGN channel. Estimate the BER.

Create QAM TCM modulator and demodulator System objects™.

```
hMod = comm.RectangularQAMTCMModulator;
hDemod = comm.RectangularQAMTCMDemodulator('TracebackDepth',16);
```

Create an AWGN channel object.

```
hAWGN = comm.AWGNChannel('EbNo',5);
```

Determine the delay through the QAM TCM demodulator. The demodulator uses the Viterbi algorithm to decode the TCM signal that was modulated using rectangular QAM. To accurately calculate the bit error rate, the delay through the decoder must be known.

```
bitsPerSymbol = log2(hDemod.TrellisStructure.numInputSymbols);
delay = hDemod.TracebackDepth*bitsPerSymbol;
```

Create an error rate calculator object with the `ReceiveDelay` property set to `delay`.

```
hErrorCalc = comm.ErrorRate('ReceiveDelay',delay);
```

Generate binary data and modulate with 16-QAM TCM. Pass the signal through an AWGN channel and demodulate. Calculate the error statistics. The loop runs until either 100 bit errors are encountered or 1e7 total bits are transmitted.

```
% Initialize the error results vector.
errStats = [0 0 0];

while errStats(2) < 100 && errStats(3) < 1e7
    % Transmit frames of 200 3-bit symbols
    txData = randi([0 1],600,1);
    % Modulate
    txSig = step(hMod,txData);
    % Pass through AWGN channel
    rxSig = step(hAWGN,txSig);
    % Demodulate
    rxData = step(hDemod,rxSig);
    % Collect error statistics
    errStats = step(hErrorCalc,txData,rxData);
end
```

Display the error data.

```
fprintf('Error rate = %4.2e\nNumber of errors = %d\n', ...
    errStats(1),errStats(2))
```

```
Error rate = 1.94e-03
Number of errors = 100
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMTCMDemodulator` | `comm.RectangularQAMTCMModulator` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: comm.RectangularQAMTCMDemodulator
Package: comm

Reset states of the rectangular QAM TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RectangularQAMTCMDemodulator` object, `H`.

step

System object: comm.RectangularQAMTCMDemodulator
Package: comm

Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RectangularQAMTCMModulator System object

Package: comm

Convolutionally encode binary data and map using rectangular QAM signal 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.

To convolutionally encode binary data and map the result using a rectangular QAM constellation:

1. Define and set up your rectangular QAM TCM modulator object. See “Construction” on page 3-1392.
2. Call `step` to modulate the signal according to the properties of `comm.RectangularQAMTCMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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` on page 3-0 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` on page 3-0 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

reset	Reset states of the rectangular QAM TCM modulator object
step	Convolutionally encode binary data and map using rectangular QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data Using Rectangular QAM TCM

Modulate data using rectangular 16-QAM TCM modulation and display the scatter plot.

Generate random binary data. The length of the data vector must be an integer multiple of the number of input streams into the encoder, log2(8) = 3.

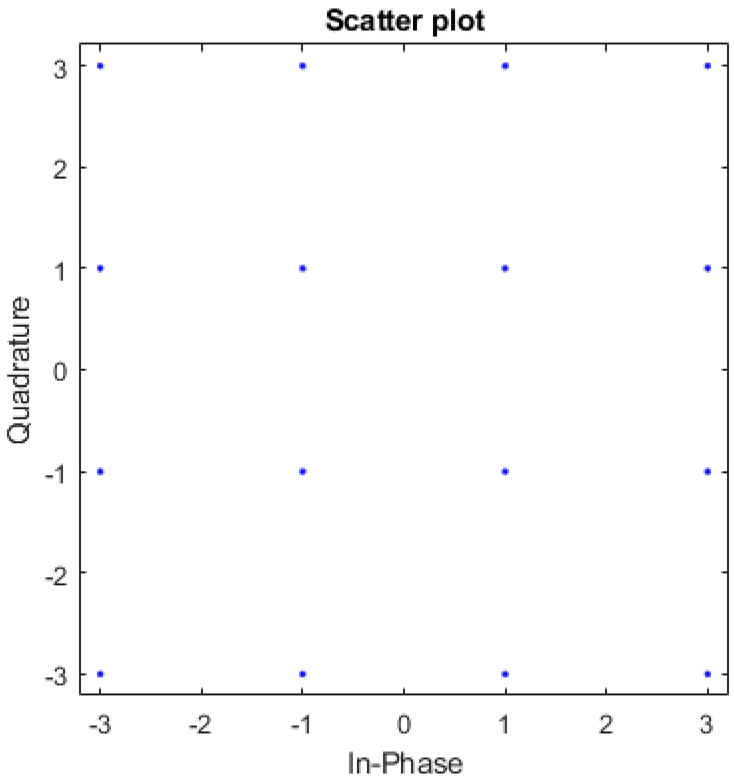
```
data = randi([0 1],3000,1);
```

Create a modulator System object™ and use its `step` function to modulate the data.

```
hMod = comm.RectangularQAMTCMModulator;
modData = step(hMod,data);
```

Plot the modulated 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMDemodulator`

Introduced in R2012a

reset

System object: comm.RectangularQAMTCMModulator
Package: comm

Reset states of the rectangular QAM TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RectangularQAMTCMModulator` object, `H`.

step

System object: comm.RectangularQAMTCMModulator
Package: comm

Convolutionally encode binary data and map using rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RicianChannel System object

Package: comm

Filter input signal through a Rician fading channel

Description

The `RicianChannel` System object filters an input signal through a Rician multipath fading channel. The fading processing per link is described in Methodology for Simulating Multipath Fading Channels.

To filter an input signal using a Rician multipath fading channel:

1. Define and set up your Rician channel object. See "Construction" on page 3-1401.
2. Call `step` to filter the input signal through a Rician multipath fading channel according to the properties of `comm.Ricianhannel`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RicianChannel` creates a frequency-selective or frequency-flat multipath Rician fading channel System object, `H`. This object filters a real or complex input signal through the multipath channel to obtain the channel impaired signal.

`H = comm.RicianChannel(Name,Value)` creates a multipath Rician 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 `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 channel is frequency flat.

When you set `PathDelays` to a vector, the 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.

`NormalizePathGains`

Normalize average path gains to 0 dB

When you set this property to true, the object normalizes the fading processes so that the total power of the path gains, averaged over time, is 0dB. The default value of this property is true.

`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`. The default value of this property is `3`.

If KFactor is a scalar, then 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.

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 is a row vector, 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 phase(s) 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.

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 is a row vector, 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.

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 bi-Gaussian, f_c is `1`. For Gaussian and bi-Gaussian Doppler spectrum types, f_c is dependent on the Doppler spectrum object properties. Refer to the algorithm section of the `comm.MIMOChannel` for more details about how $\mathrm{f_c}$ is defined.

DopplerSpectrum

Doppler spectrum

Specify the Doppler spectrum shape for the path(s) of the channel. This property accepts a single Doppler spectrum structure returned from the `doppler` function or a row cell array of such structures. 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. The default value of this property is `doppler('Jakes')`.

If you assign a single Doppler spectrum structure to `DopplerSpectrum`, all paths have the same specified Doppler spectrum. If the `FadingTechnique` property is `Sum of sinusoids`, `DopplerSpectrum` must be `doppler('Jakes')`; otherwise, select from the following:

- `doppler('Jakes')`
- `doppler('Flat')`
- `doppler('Rounded', ...)`
- `doppler('Bell', ...)`
- `doppler('Asymmetric Jakes', ...)`
- `doppler('Restricted Jakes', ...)`
- `doppler('Gaussian', ...)`
- `doppler('BiGaussian', ...)`

If you assign a row cell array of different Doppler spectrum structures (which can be chosen from any of those on the previous list) to `DopplerSpectrum`, each path has the Doppler spectrum specified by the corresponding structure in the cell array. In this case, the length of `DopplerSpectrum` must be equal to the length of PathDelays.

To generate C code, specify this property to a single Doppler spectrum structure.

FadingTechnique

Fading technique used to model the channel

Select between Filtered Gaussian noise and Sum of sinusoids to specify the way in which the channel is modeled. The default value is Filtered Gaussian noise.

NumSinusoids

Number of sinusoids used to model the fading process

The NumSinuoids property is a positive integer scalar that specified the number of sinusoids used in modeling the channel and is available only when the FadingTechnique property is set to Sum of sinusoids. The default value is 48.

InitialTimeSource

Source to control the start time of the fading process

Specify the initial time source as either Property or Input port. This property is available when the FadingTechnique property is set to Sum of sinusoids. When InitialTimeSource is set to Input port, the start time of the fading process is specified using the INITIALTIME input to the step function. The input value can change in consecutive calls to the step function. The default value is Property.

InitialTime

Start time of the fading process (s)

Specify the time offset of the fading process as a real nonnegative scalar in seconds. This property applies when the FadingTechnique property is set to Sum of sinusoids and the InitialTimeSource property is set to Property. The default value is 0.

InitialTime must be greater than the last frame end time. When InitialTime is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

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 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.

`PathGainsOutputPort`

Output channel path gains

Set this property to `true` to output the channel path gains of the underlying fading process. The default value of this property is false.

`Visualization`

Enable channel visualization

Specify the type of channel visualization to display as one of `Off` | `Impulse response` | `Frequency response` | `Impulse and frequency responses` | `Doppler spectrum`. The default value of this property is `Off`.

`SamplesToDisplay`

Specify percentage of samples to display

You can specify the percentage of samples to display, since displaying fewer samples will result in better performance at the expense of lower accuracy. Specify the property as one of `10%` | `25%` | `50%` | `100%`. This applies when `Visualization` is set to `Impulse` response, Frequency response, or Impulse and frequency responses. The default value is 25%.

PathsForDopplerDisplay

Specify path for Doppler display

You can specify an integer scalar which selects the discrete path used in constructing a Doppler spectrum plot. The specified path must be an element of {1, 2, ..., N_p}, where N_p is the number of discrete paths per link specified in the object. This property applies when Visualization is set to Doppler spectrum. The default value is 1.

Methods

info	Characteristic information about Rician Channel
reset	Reset states of the RicianChannel object
step	Filter input signal through multipath Rician fading channel

Common to All System Objects	
release	Allow System object property value changes

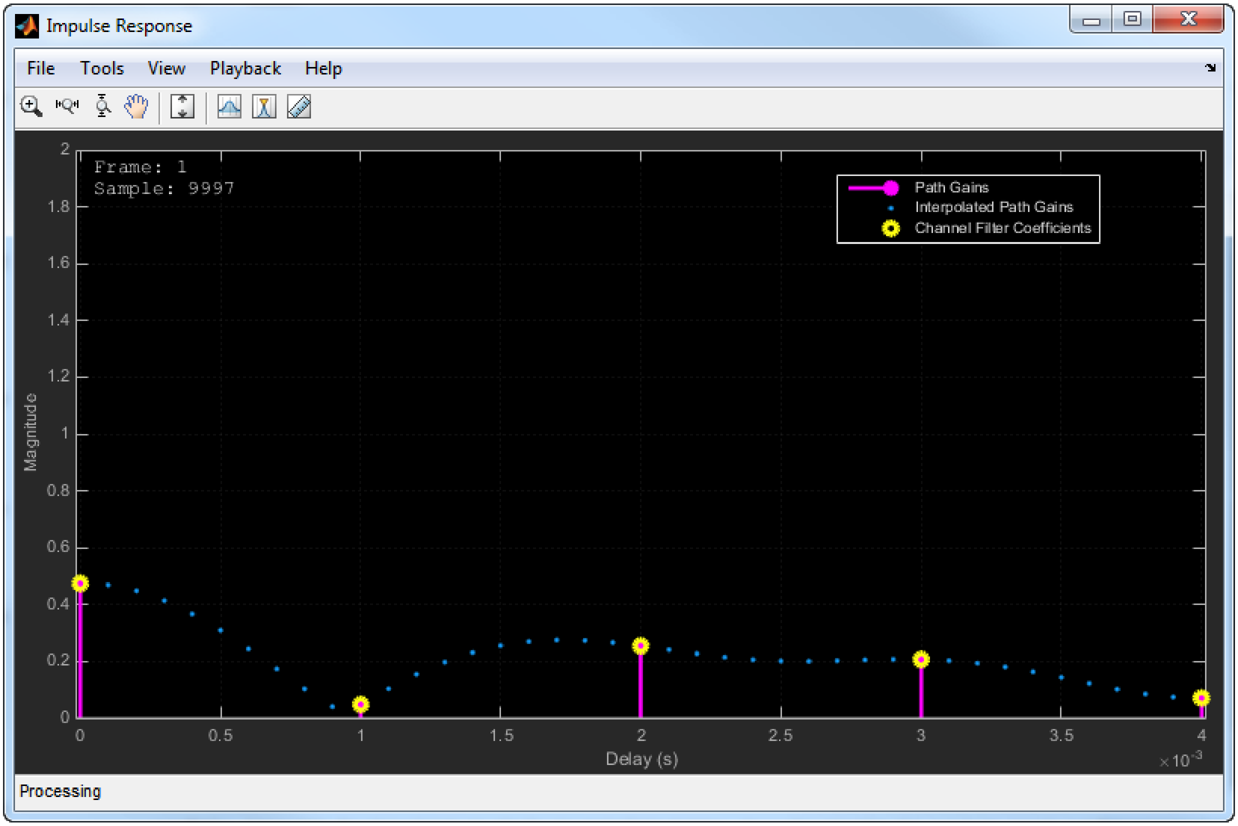
Visualization

Impulse Response

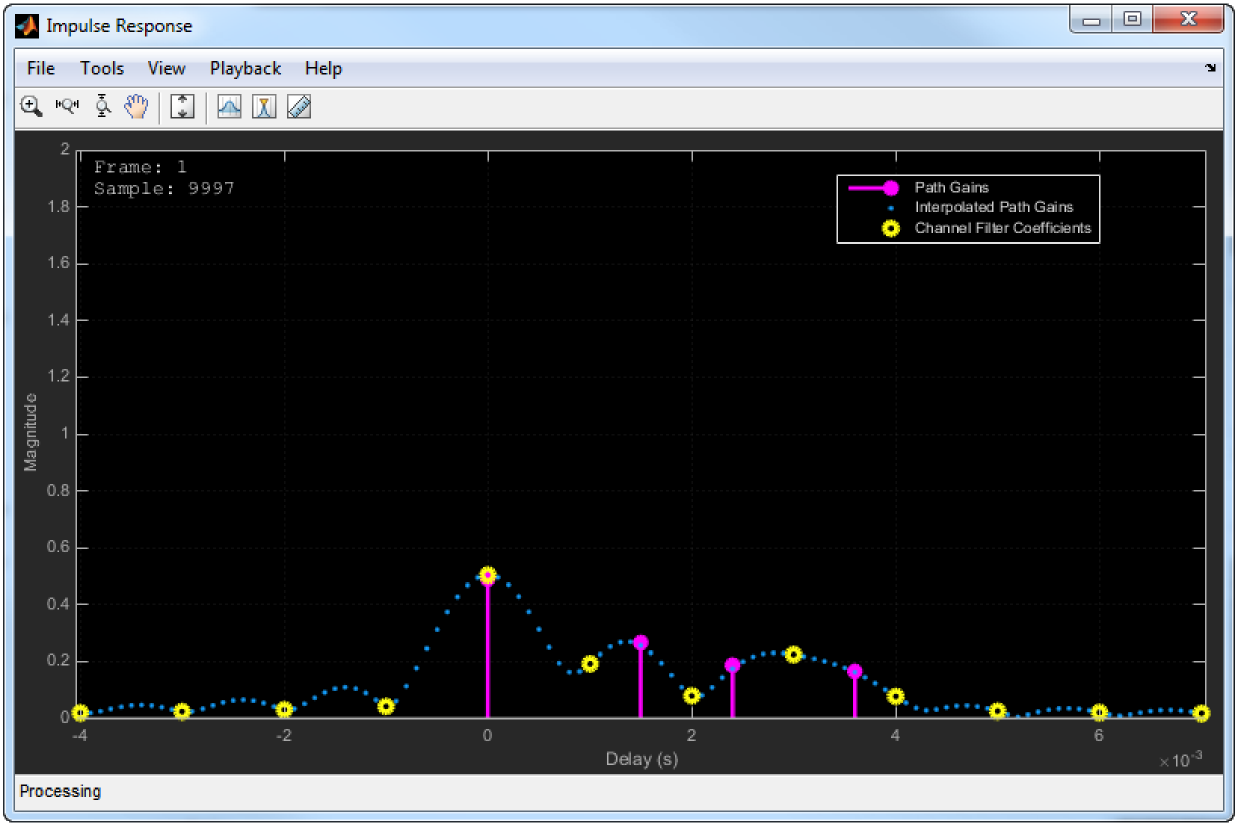
The impulse response plot displays the path gains, the channel filter coefficients, and the interpolated path gains. The path gains shown in magenta occur at time instances which correspond to the specified PathDelays property and may not be aligned with the input sampling time. The channel filter coefficients shown in yellow are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. In cases in which the path gains are aligned with the sampling time, they will overlap the filter coefficients. Sinc interpolation is used to connect the channel filter coefficients and is shown in blue. These points are used solely for display purposes and not used in subsequent channel filtering. For a flat fading channel (one path), the sinc interpolation curve is not displayed. For all impulse response plots, the frame and sample numbers are shown in the display’s upper left corner.

The impulse response plot shares the same toolbar and menus as the System object it was based on, `dsp.ArrayPlot`.

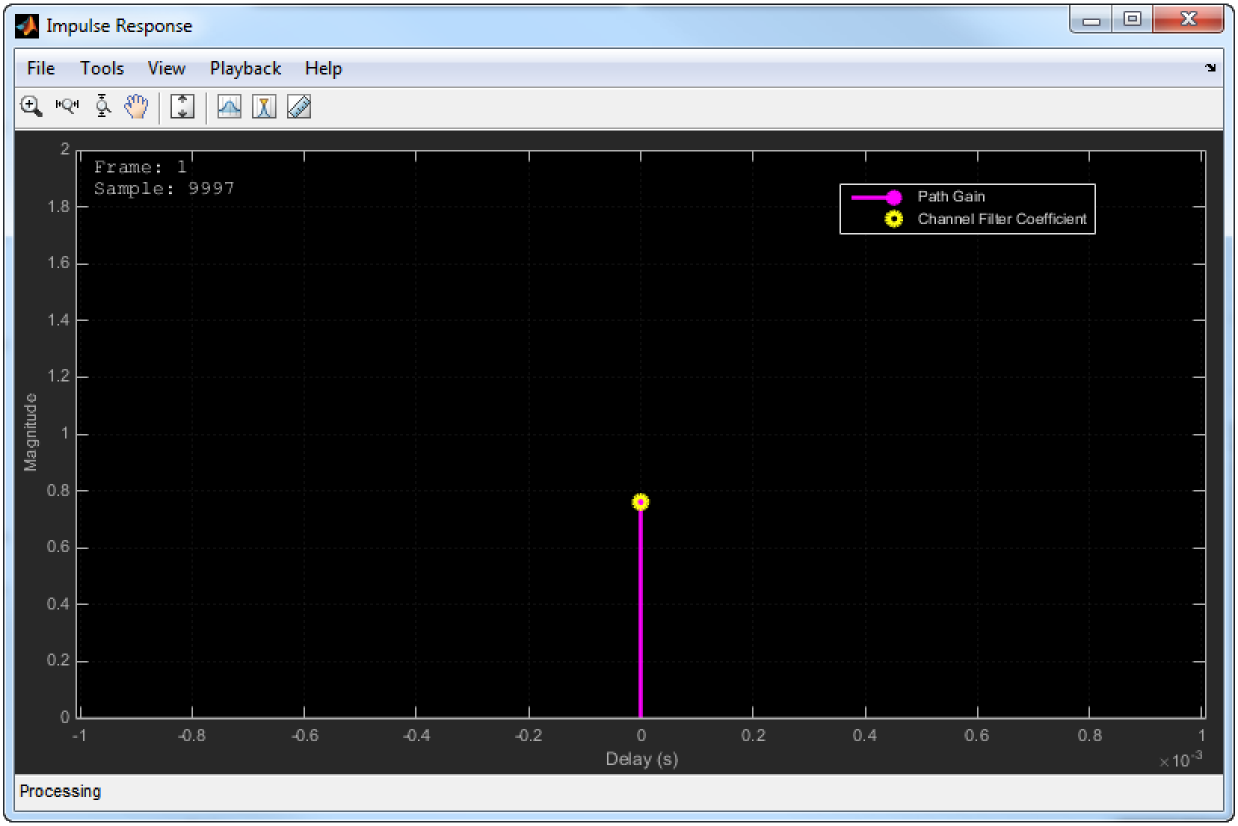
In the figure, the impulse response of a channel is shown for the case in which the path gains are aligned with the sample time. The overlap between the path gains and filter coefficients is evident.



The case in which the specified path gains are not aligned with the `SampleRate` property is shown below. Observe that the path gains and the channel filter coefficients do not overlap and that the filter coefficients are equally distributed.



The impulse response for a frequency flat channel is shown below. You can see that the interpolated path gains are not displayed.



Note

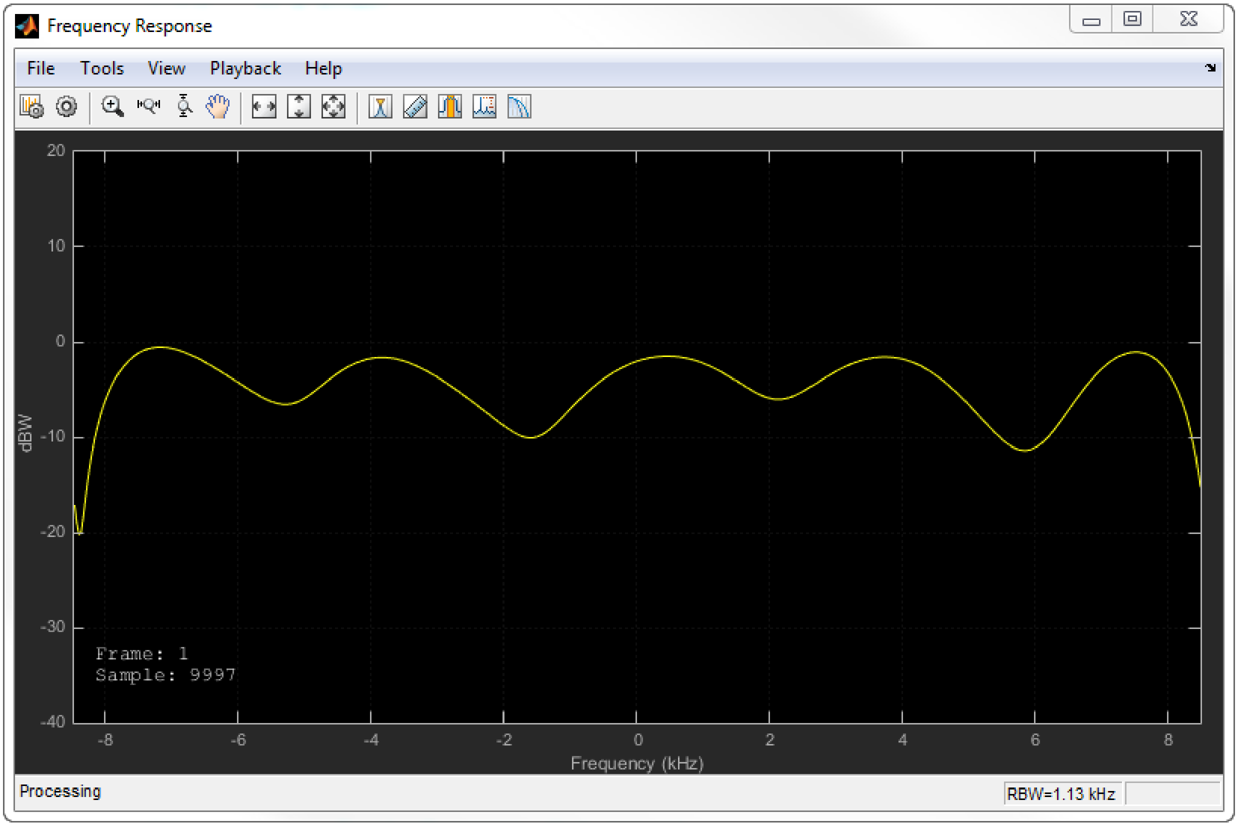
- The displayed and specified path gain locations can differ by as much as 5% of the input sample time.
- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Reduce Updates to Improve Performance** menu item. Reducing the percentage of samples to display and the enabling reduced updates will speed up the rendering of the impulse response.
- After the impulse response plots are manually closed, the `step` call for the Rician channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

Frequency Response

The frequency response plot displays the Rician channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. The frequency response plot shares the same toolbar and menus as the System object it was based on, `dsp.SpectrumAnalyzer`. The default parameter settings are shown below. These parameters can be changed from their default values by using the **View > Spectrum Settings** menu.

Parameter	Value
`Window`	`Rectangular`
`WindowLength`	Channel filter length
`FFTLength`	512
`PowerUnits`	`dBW`
`YLimits`	Based on `NormalizePathGains` and `AveragePathGains` properties

The frequency response plot for a frequency selective channel is shown.

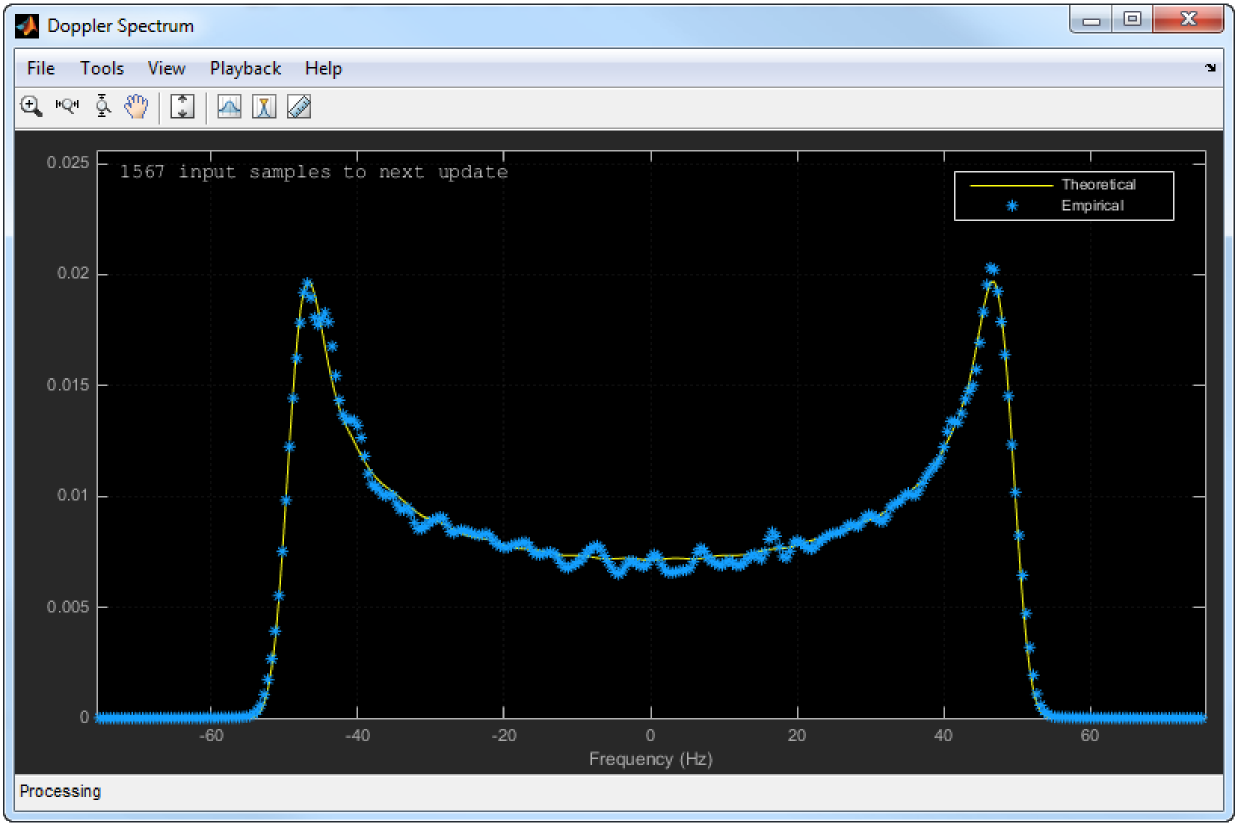


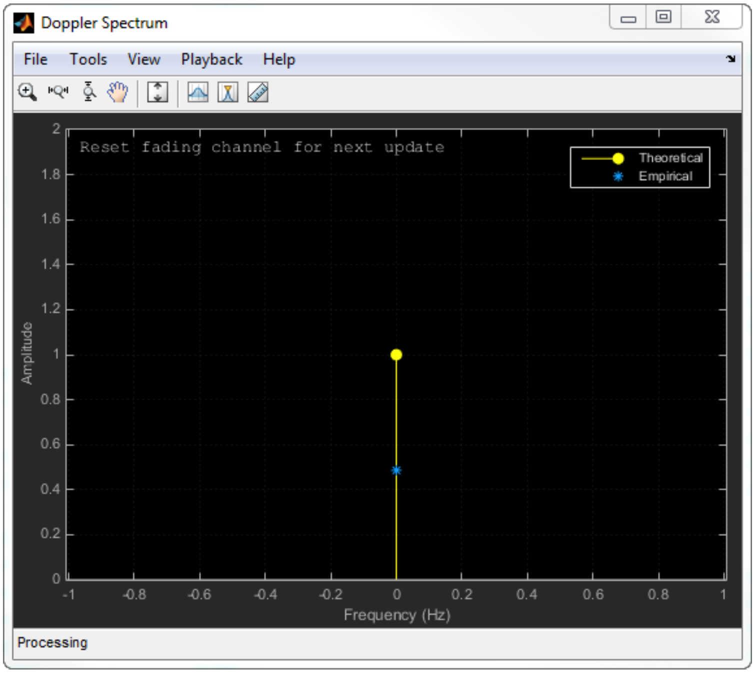
Note

- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Reduce Plot Rate to Improve Performance** menu item. Reducing the percentage of samples to display and the enabling reduced updates will speed up the rendering of the frequency response.
- After the frequency response plots are manually closed, the `step` call for the Rician channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

Doppler Spectrum

The Doppler spectrum plot displays both the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a yellow line for the case of non-static channels and as a yellow point for static channels, while the empirical data is shown in blue. There is an internal buffer which must be completely filled with filtered Gaussian samples before the empirical plot is updated. The empirical plot is the running mean of the spectrum calculated from each full buffer. For non-static channels, the number of input samples needed before the next update is displayed in the upper left hand corner. The samples needed is a function of the sample rate and the maximum Doppler shift. For static channels, the text `Reset fading channel for next update` is displayed.





Note

- After the Doppler spectrum plots are manually closed, the `step` call for the Rician channel object will be executed at its normal speed.
- Code generation is available only when the `Visualization` property is `Off`.

Examples

Produce Same Rician Channel Output Using Different Random Number Generation Methods

The Rician Channel System object™ has two methods for random number generation. You can use the current global stream or the mt19937ar algorithm with a specified seed. By interacting with the global stream, the object can produce the same outputs from the two methods.

Create a PSK Modulator System object to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
channelInput = pskModulator(randi([0 pskModulator.ModulationOrder-1],1024,1));
```

Create a Rician channel System object. Set the `RandomStream` property to `mt19937ar with seed` using a name-value pair. Set the random number seed to 73.

```
ricianChan = comm.RicianChannel(...
    'SampleRate',1e6,...
    'PathDelays',[0.0 0.5 1.2]*1e-6,...
    'AveragePathGains',[0.1 0.5 0.2],...
    'KFactor',2.8,...
    'DirectPathDopplerShift',5.0,...
    'DirectPathInitialPhase',0.5,...
    'MaximumDopplerShift',50,...
    'DopplerSpectrum',doppler('Bell', 8),...
    'RandomStream','mt19937ar with seed', ...
    'Seed',73, ...
    'PathGainsOutputPort',true);
```

Filter the modulated data using the Rician channel System object, `ricianChan`.

```
[RicianChanOut1, RicianPathGains1] = ricianChan(channelInput);
```

Set the object to use the global stream for random number generation.

```
release(ricianChan);
ricianChan.RandomStream = 'Global stream';
```

Set the global stream to use the same seed that was specified for `hRicianChan`.

```
rng(73)
```

Filter the modulated data using `hRicianChan` for the case where the channel uses the global random number generator.

```
[RicianChanOut2,RicianPathGains2] = ricianChan(channelInput);
```

Verify that the channel and path gain outputs are the same for both function calls.

```
isequal(RicianChanOut1,RicianChanOut2)
```

```
ans = logical
   1
```

```
isequal(RicianPathGains1,RicianPathGains2)

ans = logical
   1
```

Rician Channel Impulse and Frequency Responses

This example shows how to create a frequency selective Rician channel and display its impulse and frequency responses.

Set the sample rate to 3.84 MHz and specify path delays and gains using ITU pedestrian B channel parameters. Set the Rician K-factor to 10 and the maximum Doppler shift to 50 Hz.

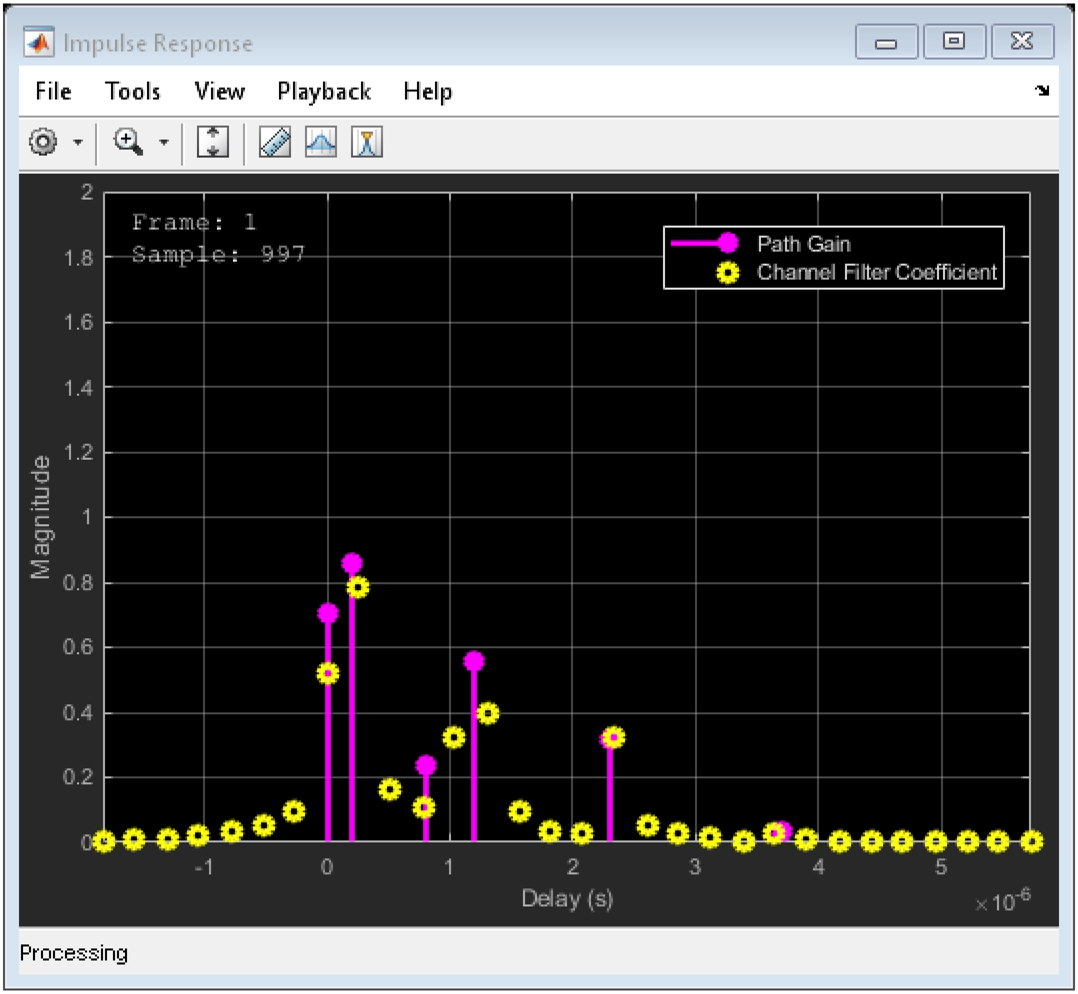
```
fs = 3.84e6;                                    % Hz
pathDelays = [0 200 800 1200 2300 3700]*1e-9;   % sec
avgPathGains = [0 -0.9 -4.9 -8 -7.8 -23.9];     % dB
fD = 50;                                        % Hz
```

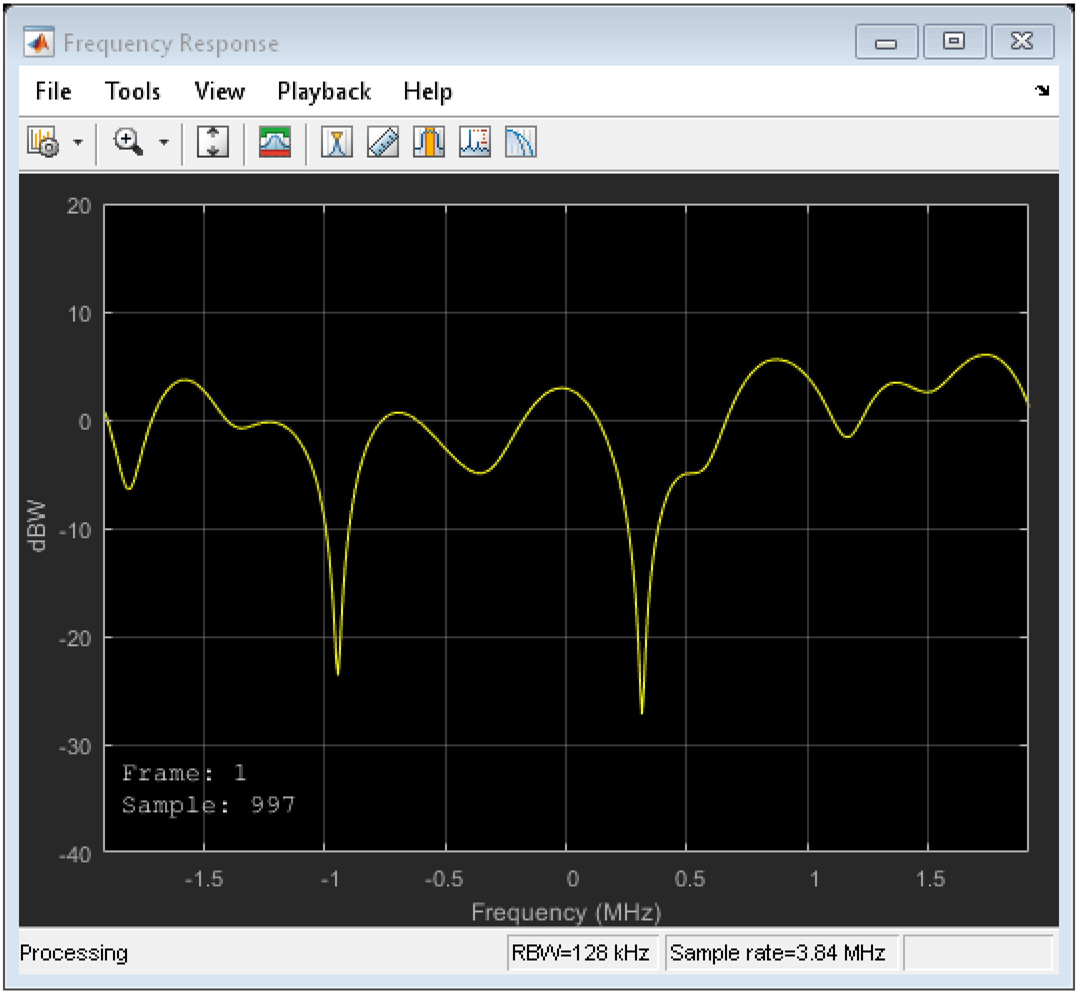
Create a Rician channel System object with the previously defined parameters and set the `Visualization` property to `Impulse and frequency responses` using name-value pairs.

```
ricianChan = comm.RicianChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'KFactor',10, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the Rician channel. The impulse response plot allows you to easily identify the individual paths and their corresponding filter coefficients. The frequency selective nature of the pedestrian B channel is shown by the frequency response plot.

```
x = randi([0 1],1000,1);
y = ricianChan(x);
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 997
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-1
0
1
2
3
4
5
Delay (s)
×10⁻⁶
Processing



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.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. “Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms.” *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AWGNChannel` | `comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.RayleighChannel`

Introduced in R2013b

info

System object: comm.RicianChannel
Package: comm

Characteristic information about Rician Channel

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.

reset

System object: comm.RicianChannel
Package: comm

Reset states of the `RicianChannel` object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RicianChannel` 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.

step

System object: comm.RicianChannel
Package: comm

Filter input signal through multipath Rician fading channel

Syntax

```
Y = step(H,X)
[Y,PATHGAINS] = step(H,X)
___ = step(H,X,INITIALTIME)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` filters input signal `X` through a multipath Rician fading channel and returns the result in `Y`. Both the input `X` and the output signal `Y` are of size N_s–by–1, where N_s represents the number of samples. The input `X` can be of double- or single-precision data type with real or complex values. `Y` is contains complex values with same precision as input signal.

`[Y,PATHGAINS] = step(H,X)` returns the channel path gains of the underlying Rician 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, where N_p represents the number of paths, i.e., the length of the PathDelays property value of `H`. PATHGAINS contains complex values with same precision as input signal.

`___ = step(H,X,INITIALTIME)` passes data through the Rician channel beginning at `INITIALTIME`, where `INITIALTIME` is a nonnegative real scalar measured in seconds. This syntax applies when the FadingTechnique property of `H` is set to `Sum of sinusoids` and the InitialTimeSource property of `H` is set to `Input port`. The `INITIALTIME` must be greater than the last frame end time. When `INITIALTIME` is not a multiple of 1/`SampleRate`, it is rounded up to the nearest sample position.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.RSDecoder System object

Package: comm

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 the property values in the corresponding RS Encoder object.

To decode data using a Reed-Solomon decoding scheme:

1. Define and set up your Reed-Solomon decoder object. See “Construction” on page 3-1424.
2. Call `step` to decode data according to the properties of `comm.RSDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dec = comm.RSDecoder` creates a block decoder System object, `dec`. This object performs Reed-Solomon (RS) decoding.

`dec = comm.RSDecoder(N,K)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N` and the `MessageLength` property set to `K`.

`dec = comm.RSDecoder(N,K,GP)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, and the `GeneratorPolynomial` property set to `GP`.

dec = comm.RSDecoder(N,K,GP,S) creates an RS decoder object, dec with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, and the ShortMessageLength property set to S.

dec = comm.RSDecoder(N,K,GP,S,Name,Value) creates an RS decoder object, dec with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, and each specified property Name set to the specified Value.

dec = comm.RSDecoder(Name,Value) creates an RS decoder object, dec, 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

Note The input and output signal lengths are listed in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111 on the comm.BCHDecoder reference page.

BitInput

Assume that 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 step method outputs an encoded data output vector. The output result is a column vector of integers. 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 on page 3-0 and PrimitivePolynomial on page 3-0 properties.

When you set this property to true, the input value must be a numeric, column vector of bits. The encoded data output result is a column vector of bits.

CodewordLength

Codeword length

Specify the codeword length of the RS code in symbols as a double-precision, positive, integer scalar value. The default is 7.

For a full-length RS code, the value of this property must be 2^M–1, where M is an integer such that $3 \le M \le 16$.

MessageLength

Message length

Specify the message length in symbols as a double-precision positive integer scalar value. The default is 3.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as Auto or Property. When this property is set to Auto, the RS code is defined by the CodewordLength on page 3-0 , MessageLength on page 3-0 , GeneratorPolynomial on page 3-0 , and PrimitivePolynomial on page 3-0 properties.

When ShortMessageLengthSource is set to Property, you must specify the ShortMessageLength on page 3-0 property, which is used with the other properties to define the RS code. The default is Auto.

ShortMessageLength

Shortened message length

Specify the length of the shortened message in symbols as a double-precision positive integer scalar whose value must be less than or equal to MessageLength on page 3-0 .

When ShortMessageLength < MessageLength, the RS code is shortened. The default is 3.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as `Auto` or `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` on page 3-0 property.

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 `PrimitivePolynomial` on page 3-0 property.

When you set this property to Property, you must specify a generator polynomial using the `GeneratorPolynomial` on page 3-0 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 `GeneratorPolynomialSource` on page 3-0 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 the specified generator polynomial is valid.

For larger codes, disabling the check accelerates processing time. As a best practice, perform the check at least once before setting this property to `false`. This property applies when you set `GeneratorPolynomialSource` on page 3-0 to `Property`.

`PrimitivePolynomialSource`

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto` or `Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil`(log2(`CodewordLength` on page 3-0 +1)).

When you set`PrimitivePolynomialSource` to `Property`, you must specify a polynomial using `PrimitivePolynomial` on page 3-0 .

`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. Specify this property as 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 `PrimitivePolynomialSource` on page 3-0 to `Property`.

`PuncturePatternSource`

Source of puncture pattern

Specify the source of the puncture pattern as `None` or `Property`. The default is `None`.

If you set this property to `None`, the object does not apply puncturing to the code. If you set it to `Property`, the object punctures the code based on a puncture pattern vector specified in `PuncturePattern` on page 3-0 .

`PuncturePattern`

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector of length (`CodewordLength` on page 3-0 -`MessageLength` on page 3-

0). 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 `PuncturePatternSource` on page 3-0 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.

The length of the erasures vector is explained in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111.

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`, or `logical`. The default is `Same as input`. This property applies when you set `BitInput` on page 3-0 to `true`.

Methods

step	Decode data using a Reed-Solomon decoder

Common to All System Objects	
release	Allow System object property value changes

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
    enc = comm.RSEncoder;
    mod = comm.DPSKModulator('BitInput',false);
    chan = comm.AWGNChannel(...
                'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
    demod = comm.DPSKDemodulator('BitOutput',false);
    hDdecec = comm.RSDecoder;
    errorRate = comm.ErrorRate('ComputationDelay',3);

    for counter = 1:20
      data = randi([0 7], 30, 1);
      encodedData = step(enc, data);
      modSignal = step(mod, encodedData);
      receivedSignal = step(chan, modSignal);
      demodSignal = step(demod, receivedSignal);
      receivedSymbols = step(hDdecec, demodSignal);
      errorStats = step(errorRate, data, receivedSymbols);
    end

  fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.115578
Number of errors = 69
```

Transmit a Shortened RS-encoded, 256-QAM-modulated Symbol Stream

Transmit a shortened RS-encoded, 256-QAM-modulated symbol stream through an AWGN channel. Then demodulate, decode, and count errors.

Set the parameters for the Reed-Solomon code, where N is the codeword length, K is the nominal message length, and S is the shortened message length. Set the modulation order, M, and the number of frames, L.

```
N = 255;
K = 239;
S = 188;
M = 256;
L = 50;
```

Create the QAM modulator, QAM demodulator, AWGN channel, and error rate System objects.

```
mod = comm.RectangularQAMModulator(M, ...
    'NormalizationMethod','Average power');
chan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'EbNo',15,'BitsPerSymbol',log2(M));
demod = comm.RectangularQAMDemodulator(M, ...
    'NormalizationMethod','Average power');
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Create the Reed-Solomon generator polynomial from the DVB-T standard.

```
gp = rsgenpoly(N,K,[],0);
```

Create a Reed-Solomon encoder and decoder pair using the shortened message length, S, and the DVB-T generator polynomial, gp.

```
enc = comm.RSEncoder(N,K,gp,S);
dec = comm.RSDecoder(N,K,gp,S);
```

Generate random symbol frames whose length equals one message block. Encode, modulate, apply AWGN, demodulate, decode, and collect statistics.

```
for counter = 1:L
    data = randi([0 1],S,log2(M));
    encodedData = step(enc,bi2de(data));
    modSignal = step(mod,encodedData);
    receivedSignal = step(chan,modSignal);
    demodSignal = step(demod,receivedSignal);
    receivedBits = step(dec,demodSignal);
    dataOut = de2bi(receivedBits);
    errorStats = step(errorRate,data(:),dataOut(:));
end
```

Display the error rate and number of errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 2.01e-02
Number of errors = 1509
```

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHDecoder` | `comm.RSEncoder` | `primpoly` | `rsdec` | `rsgenpoly`

Introduced in R2012a

step

System object: comm.RSDecoder
Package: comm

Decode data using a Reed-Solomon decoder

Syntax

```
[Y,ERR] = step(H,X)
Y = step(H,X)
Y = step(H,X,ERASURES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,ERR] = step(H,X)` decodes the encoded input data, `X`, into the output vector `Y` and returns the number of corrected symbols 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 input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111. 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`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

Encode data using Reed-Solomon encoder

Description

The `RSEncoder` object creates a Reed-Solomon code with message and codeword lengths you specify.

To encode data using a Reed-Solomon encoding scheme:

1. Define and set up your Reed-Solomon encoder object. See "Construction" on page 3-1435.
2. Call `step` to encode data according to the properties of `comm.RSEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`enc = comm.RSEncoder` creates a block encoder System object, `enc`. This object performs Reed-Solomon (RS) encoding.

`enc = comm.RSEncoder(N,K)` creates an RS encoder object, `enc`, with the `CodewordLength` property set to `N` and the `MessageLength` property set to `K`.

`enc = comm.RSEncoder(N,K,GP)` creates an RS encoder object, `enc`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, and the `GeneratorPolynomial` property set to `GP`.

`enc = comm.RSEncoder(N,K,GP,S)` creates an RS encoder object, `enc`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, the

GeneratorPolynomial property set to GP, and the ShortMessageLength property set to S.

enc = comm.RSEncoder(N,K,GP,S,Name,Value) creates an RS encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, the ShortMessageLength property set to S, and each specified property Name set to the specified Value.

enc = comm.RSEncoder(Name,Value) creates an RS encoder object, enc, 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

Note The input and output signal lengths are listed in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111 on the comm.BCHDecoder reference page.

BitInput

Assume that 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. 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 on page 3-0 and PrimitivePolynomial on page 3-0 properties.

When you set this property to true, the input value must be a numeric, column vector of bits. The encoded data output result is a column vector of bits.

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision positive integer scalar value. The default is `7`.

For a full-length RS code, the value of this property must be 2^M–1, where M is an integer such that $3 \leq M \leq 16$.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar value. The default is `3`.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as `Auto` or `Property`. When this property is set to `Auto`, the RS code is defined by the `CodewordLength` on page 3-0, `MessageLength` on page 3-0, `GeneratorPolynomial` on page 3-0, and `PrimitivePolynomial` on page 3-0 properties. When `ShortMessageLengthSource` is set to `Property`, you must specify the `ShortMessageLength` on page 3-0 property, which is used with the other properties to define the RS code. The default is `Auto`.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to `MessageLength` on page 3-0. When `ShortMessageLength` < `MessageLength`, the RS code is shortened. The default is `3`.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as `Auto` or `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 `PrimitivePolynomial` on page 3-0 property.

When you set `GeneratorPolynomialSource` to `Property`, you must specify a generator polynomial using the `GeneratorPolynomial` on page 3-0 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. The Galois row vector entries must be in the range from `0` to 2^M`-1`. These entries must represent a generator polynomial in descending order of powers. Each coefficient is an element of the Galois field `gf`(2^M), represented in integer format. The length of the generator polynomial must be `CodewordLength` on page 3-0 – `MessageLength` on page 3-0 + 1.

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 `GeneratorPolynomialSource` on page 3-0 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 the specified generator polynomial is valid. For larger codes, disabling the check speeds up processing. As a best practice, perform the check at least once before setting this property to `false`. This property applies when `GeneratorPolynomialSource` on page 3-0 is set to `Property`.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto` or `Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil`(log2(`CodewordLength` on page 3-0 +1)).

When you set this property to `Property`, you must specify a polynomial using the `PrimitivePolynomial` on page 3-0 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. Specify this property as a double-precision, binary row vector that represents a primitive polynomial over gf(2) of degree M in descending order of powers.

If CodewordLength on page 3-0 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 $x^3 + x + 1$.

This property applies when you set PrimitivePolynomialSource on page 3-0 to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as None or Property. The default is None.

If you set this property to None, the object does not apply puncturing to the code. If you set this property to Property, the object punctures the code based on a puncture pattern vector specified in the PuncturePattern on page 3-0 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 on page 3-0 – MessageLength on page 3-0). 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 on page 3-0 property to Property.

OutputDataType

Data type of output

Specify the output data type as `Same as input`, `double`, or `logical`. The default is `Same as input`. This property applies when you set the `BitInput` on page 3-0 property to `true`.

Methods

step	Encode data using a Reed-Solomon encoder

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
    enc = comm.RSEncoder;
    mod = comm.DPSKModulator('BitInput',false);
    chan = comm.AWGNChannel(...
                'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
    demod = comm.DPSKDemodulator('BitOutput',false);
    hDdecec = comm.RSDecoder;
    errorRate = comm.ErrorRate('ComputationDelay',3);

    for counter = 1:20
      data = randi([0 7], 30, 1);
      encodedData = step(enc, data);
      modSignal = step(mod, encodedData);
      receivedSignal = step(chan, modSignal);
      demodSignal = step(demod, receivedSignal);
      receivedSymbols = step(hDdecec, demodSignal);
      errorStats = step(errorRate, data, receivedSymbols);
    end

  fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.115578
Number of errors = 69
```

Transmit a Shortened RS-encoded, 256-QAM-modulated Symbol Stream

Transmit a shortened RS-encoded, 256-QAM-modulated symbol stream through an AWGN channel. Then demodulate, decode, and count errors.

Set the parameters for the Reed-Solomon code, where `N` is the codeword length, `K` is the nominal message length, and `S` is the shortened message length. Set the modulation order, `M`, and the number of frames, `L`.

```
N = 255;
K = 239;
S = 188;
M = 256;
L = 50;
```

Create the QAM modulator, QAM demodulator, AWGN channel, and error rate System objects.

```
mod = comm.RectangularQAMModulator(M, ...
    'NormalizationMethod','Average power');
chan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'EbNo',15,'BitsPerSymbol',log2(M));
demod = comm.RectangularQAMDemodulator(M, ...
    'NormalizationMethod','Average power');
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Create the Reed-Solomon generator polynomial from the DVB-T standard.

```
gp = rsgenpoly(N,K,[],0);
```

Create a Reed-Solomon encoder and decoder pair using the shortened message length, `S`, and the DVB-T generator polynomial, `gp`.

```
enc = comm.RSEncoder(N,K,gp,S);
dec = comm.RSDecoder(N,K,gp,S);
```

Generate random symbol frames whose length equals one message block. Encode, modulate, apply AWGN, demodulate, decode, and collect statistics.

```
for counter = 1:L
    data = randi([0 1],S,log2(M));
    encodedData = step(enc,bi2de(data));
    modSignal = step(mod,encodedData);
    receivedSignal = step(chan,modSignal);
    demodSignal = step(demod,receivedSignal);
    receivedBits = step(dec,demodSignal);
    dataOut = de2bi(receivedBits);
    errorStats = step(errorRate,data(:),dataOut(:));
end
```

Display the error rate and number of errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 2.01e-02
Number of errors = 1509
```

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHEncoder` | `comm.RSDecoder` | `primpoly` | `rsenc` | `rsgenpoly`

Introduced in R2012a

step

System object: comm.RSEncoder
Package: comm

Encode data using a Reed-Solomon encoder

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-111.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.Scrambler System object

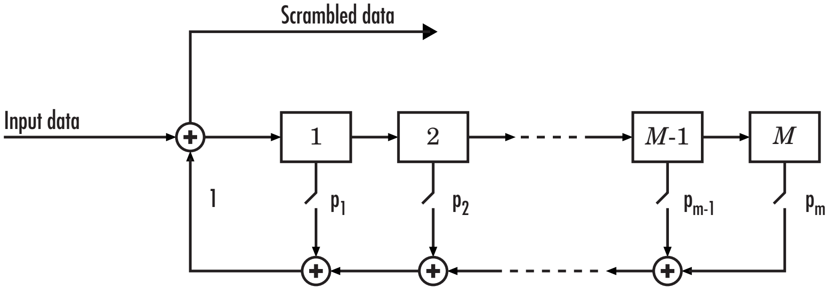
Package: comm

Scramble input signal

Description

The comm.Scrambler object scrambles a scalar or column vector input signal.

This schematic shows the scrambler operation. The adders operate modulo *N*, where *N* is the value specified by the Calculation base property.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Polynomial property, you specify the on or off state for each switch in the scrambler.

To scramble an input signal:

1. Create the comm.Scrambler object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
scrambler = comm.Scrambler
scrambler = comm.Scrambler(base,poly,cond)
scrambler = comm.Scrambler( ___ ,Name,Value)
```

Description

`scrambler = comm.Scrambler` creates a scrambler System object. This object scrambles the input data by using a linear feedback shift register that you specify with the Polynomial property.

`scrambler = comm.Scrambler(base,poly,cond)` creates the scrambler object with the CalculationBase property set to `base`, the Polynomial property set to `poly`, and the InitialConditions property set to `cond`.

Example: `comm.Scrambler(8,'1 + z^-2 + z^-3 + z^-5 + z^-7',[0 3 2 2 5 1 7])` sets the calculation base to `8`, and the scrambler polynomial and initial conditions as specified.

`scrambler = comm.Scrambler( ___ ,Name,Value)` sets properties using one or more name-value pairs and either of the previous syntaxes. Enclose each property name in single quotes.

Example: `comm.Scrambler('CalculationBase',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`CalculationBase` — Range of input data

4 (default) | nonnegative integer

Range of input data used in the scrambler for modulo operations, specified as a nonnegative integer. The input and output of this object are integers from `0` to `CalculationBase` – `1`.

Data Types: `double`

`Polynomial` — Connections for linear feedback shift registers

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Connections for linear feedback shift registers in the scrambler, specified as a character vector, integer vector, or binary vector. The `Polynomial` property defines if each switch in the scrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the scrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + \ldots$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that have a coefficient of 1. In this case, the order of the scramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Data Types: `double` | `char`

`InitialConditionsSource` — Initial conditions source

`'Property'` (default) | `'Input port'`

- `'Property'` – Specify scrambler initial conditions by using the InitialConditions property.
- `'Input port'` – Specify scrambler initial conditions by using an additional input argument, initcond, when calling the object.

Data Types: `char`

`InitialConditions` — Initial conditions of scrambler registers

`[0 1 2 3]` (default) | nonnegative integer vector

Initial conditions of scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of `InitialConditions` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

`ResetInputPort` — Scrambler state reset port
`false` (default) | `true`

Scrambler state reset port, specified as `false` or `true`. If `ResetInputPort` is `true`, you can reset the scrambler object by using an additional input argument, reset, when calling the object.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

Usage

Syntax

```
scrambledOut = scrambler(signal)
scrambledOut = scrambler(signal,initcond)
scrambledOut = scrambler(signal,reset)
```

Description

`scrambledOut = scrambler(signal)` scrambles the input signal. The output is the same data type and length as the input vector.

`scrambledOut = scrambler(signal,initcond)` provides an additional input with values specifying the initial conditions of the linear feedback shift register.

This syntax applies when you set the InitialConditionsSource property of the object to `'Input port'`.

`scrambledOut = scrambler(signal,reset)` provides an additional input indicating whether to reset the state of the scrambler.

This syntax applies when you set InitialConditionsSource to `'Property'` and ResetInputPort to `true`.

Input Arguments

`signal` — Input signal

column vector

Input signal, specified as a column vector.

Example: `scrambledOut = scrambler([0 1 1 0 1])`

Data Types: `double` | `logical`

`initcond` — Initial register conditions

nonnegative integer column vector

Initial scrambler register conditions when the simulation starts, specified as a nonnegative integer column vector. The length of `initcond` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Example: `scrambledOut = scrambler(signal,[0 1 1 0])` corresponds to possible initial register states for a scrambler with a polynomial order of 4 and a calculation base of 2 or higher.

Data Types: `double`

`reset` — Reset initial state of scrambler

scalar

Reset the initial state of the scrambler when the simulation starts, specified as a scalar. When the value of `reset` is nonzero, the object is reset before it is called.

Example: `scrambledOut = scrambler(signal,0)` scrambles the input signal without resetting the scrambler states.

Data Types: `double`

Output Arguments

out — Scrambled output
column vector

Scrambled output, returned as a column vector with the same data type and length as signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Scramble and Descramble Data

Scramble and descramble 8-ary data using `comm.Scrambler` and `comm.Descrambler` System objects™ having a calculation base of 8.

Create scrambler and descrambler objects, specifying the calculation base, polynomial, and initial conditions using input arguments. The scrambler and descrambler polynomials are specified with different but equivalent data formats.

```
N = 8;
scrambler = comm.Scrambler(N,'1 + z^-2 + z^-3 + z^-5 + z^-7', ...
    [0 3 2 2 5 1 7]);
descrambler = comm.Descrambler(N,[1 0 1 1 0 1 0 1], ...
    [0 3 2 2 5 1 7]);
```

Scramble and descramble random integers. Display the original data, scrambled data, and descrambled data sequences.

```
data = randi([0 N-1],5,1);
scrData = scrambler(data);
deScrData = descrambler(scrData);
[data scrData deScrData]
```

```
ans = 5×3

     6     7     6
     7     5     7
     1     7     1
     7     0     7
     5     3     5
```

Verify that the descrambled data matches the original data.

```
isequal(data,deScrData)
```

```
ans = logical
   1
```

Scramble and Descramble Data with Changing Initial Conditions

Scramble and descramble quaternary data while changing the initial conditions between function calls.

Create scrambler and descrambler System objects having a calculation base of 4. Set the `InitialConditionsSource` property to `'Input port'` so you can set the initial conditions as an argument to the object.

```
N = 4;
scrambler = comm.Scrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
descrambler = comm.Descrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
```

Preallocate memory for the error vector which will be used to store errors output by the `symerr` function.

```
errVec = zeros(10,1);
```

Scramble and descramble random integers while changing the initial conditions, `initCond`, each time the loop executes. Use the `symerr` function to determine if the scrambling and descrambling operations result in symbol errors.

```
for k = 1:10
    initCond = randperm(3)';
    data = randi([0 N-1],5,1);
    scrData = scrambler(data,initCond);
    deScrData = descrambler(scrData,initCond);
    errVec(k) = symerr(data,deScrData);
end
```

Examine `errVec` to verify that the output from the descrambler matches the original data.

```
errVec
```

```
errVec = 10×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

System Objects
`comm.Descrambler` | `comm.PNSequence`

Blocks
Scrambler

Introduced in R2012a

comm.SphereDecoder System object

Package: comm

Decode input using sphere decoder

Description

The `Sphere Decoder` System object decodes the symbols sent over N_T antennas using the sphere decoding algorithm.

To decode input symbols using a sphere decoder:

1. Define and set up your sphere decoder object. See “Construction” on page 3-1454.
2. Call `step` to decode input symbols according to the properties of `comm.SphereDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Decode received symbols using sphere decoding algorithm

Common to All System Objects	
`release`	Allow System object property value changes

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. Decode using a sphere decoder with perfect channel knowledge.

Specify the modulation order, the number of transmitted bits, the Eb/No ratio, and the symbol mapping.

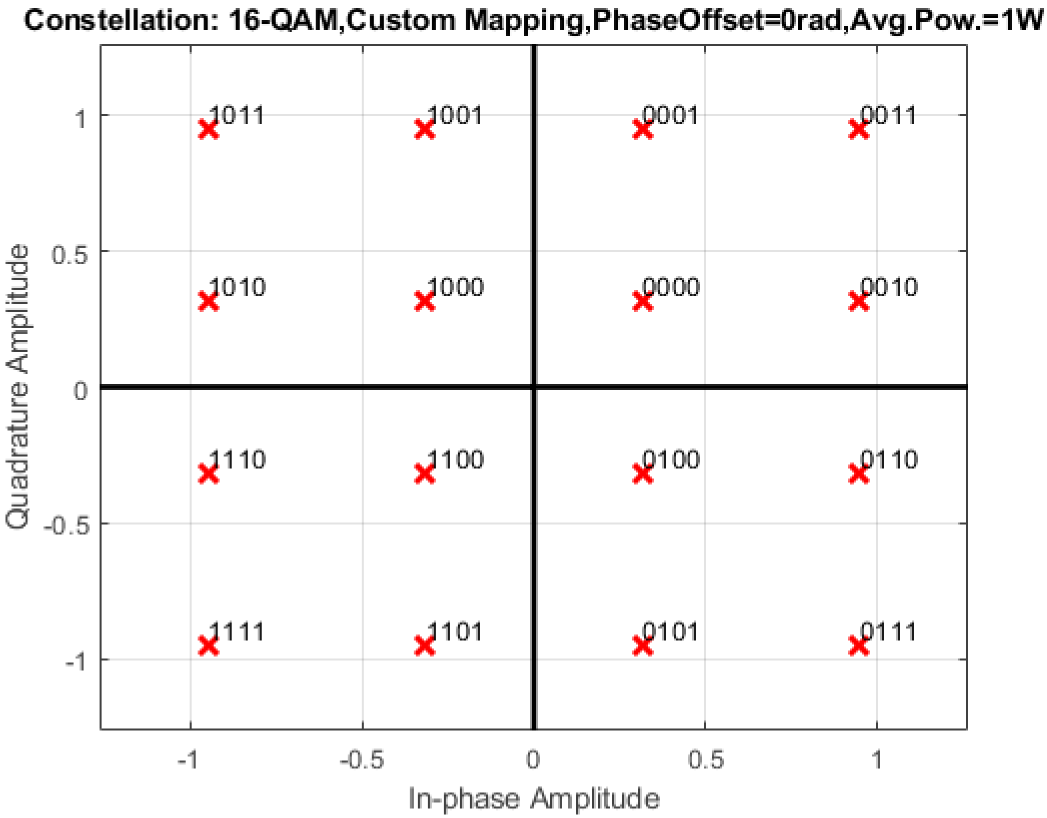
```
M = 16;
nBits = 1e3*log2(M);
ebno = 10;
symMap = [11 10 14 15 9 8 12 13 1 0 4 5 3 2 6 7];
```

Create a Rectangular QAM modulator System object™ whose properties are set using name-value pairs.

```
hMod = comm.RectangularQAMModulator(...
    'ModulationOrder',M, ...
    'BitInput',true, ...
    'NormalizationMethod','Average power', ...
    'SymbolMapping','Custom', ...
    'CustomSymbolMapping',symMap);
```

Display the symbol mapping of the 16-QAM modulator by using the `constellation` function.

```
constellation(hMod)
```



Convert the decimal value of the symbol map to binary bits using the left bit as the most significant bit (msb). The M-by-log2(M) matrix `bitTable` is used by the sphere decoder.

```
bitTable = de2bi(symMap,log2(M),'left-msb');
```

Create a 2x2 MIMO Channel System object with `PathGainsOutputPort` set to `true` to use the path gains as a channel estimate. To ensure the repeatability of results, set the object to use the global random number stream.

```
hMIMO = comm.MIMOChannel(...
    'PathGainsOutputPort',true, ...
    'RandomStream','Global stream');
```

Create an AWGN Channel System object.

```
hAWGN = comm.AWGNChannel('EbNo',ebno,'BitsPerSymbol',log2(M));
```

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;
```

Set the global random number generator seed.

```
rng(37)
```

Generate a random data stream.

```
data = randi([0 1],nBits,1);
```

Modulate the data and reshape it into two streams to be used with the 2x2 MIMO channel.

```
modData = step(hMod,data);
modData = reshape(modData,[],2);
```

Pass the modulated data through the MIMO fading channel and add AWGN.

```
[fadedSig,pathGains] = step(hMIMO,modData);
rxSig = step(hAWGN,fadedSig);
```

Decode the received signal using `pathGains` as a perfect channel estimate.

```
decodedData = step(hSpDec,rxSig,squeeze(pathGains));
```

Convert the decoded hard-decision data, which is a logical matrix, into a double column vector to enable the calculation of error statistics. Calculate and display the bit error rate and the number of errors.

```
dataOut = double(decodedData(:));
errorStats = step(hBER,data,dataOut);
errorStats(1:2)
```

```
ans = 2×1

    0.0380
  152.0000
```

Algorithm

This object implements a soft-output max-log a posteriori probability (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, expressed as:

$y = Hs + n$.

where y is the received symbols, H is the MIMO channel matrix, s is the transmitted symbol vector, and n is the thermal noise.

The MIMO detector seeks the maximum-likelihood (ML) solution, $\hat{s}_{ML}$, such that:

$$\hat{s}_{ML} = \underset{s \in O}{\arg\min} \left\| y - Hs \right\|^2$$

where O is the complex-valued constellation from which the N_T elements of s are chosen.

Soft detection also computes a log-likelihood ratio (LLR) for each bit that serves as a measure of the the reliabilty of the estimate for each bit. The LLR is calculated as using the max-log approximation:

$$L\left(x_{j,b}\right) = \underbrace{\min_{s \in x_{j,b}^{(0)}} \left\| y - Hs \right\|^2}_{\lambda^{ML}} - \underbrace{\min_{s \in x_{j,b}^{(1)}} \left\| y - Hs \right\|^2}_{\lambda_{j,b}^{\overline{ML}}}$$

where

- $L(x_{j,b})$ is the LLR estimate for each bit.
- $x_{j,b}$ is each sent bit, the *b*th bit of the *j*th symbol.
- $x_{j,b}^{(0)}$ 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 two λ symbols denotes the distance calculated as norm squared., specifically:
  - λ^{ML} is the distance $\hat{s}ML$.
  - $\lambda_{j,b}^{\overline{ML}}$ is the distance to the counter-hypothesis, which denotes the binary complement of the *b*th bit in the binary label of the *j*th entry of $\hat{s}ML$, specifically 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 $\hat{s}ML$.

Based on whether $x_{j,b}^{\left(x_{j,b}^{ML}\right)}$ is `0` or `1`, the LLR estimate for bit $x_{j,b}$ is computed as follows:

$$L(x_{j,b}) = \begin{cases} \lambda^{ML} - \lambda_{j,b}^{\overline{ML}}, & x_{j,b}^{ML} = 0 \\ \lambda_{j,b}^{\overline{ML}} - \lambda^{ML}, & x_{j,b}^{ML} = 1 \end{cases}$$

The design of a decoder strives to efficiently find $\hat{s}_{ML}$, 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:

$$\lambda^{ML} = \underset{s \in o}{\arg\min} \left\| \overline{y} - Rs \right\|^{2}$$

$$\lambda_{j,b}^{\overline{ML}} = \underset{s \in x_{j,b}^{\left(\overline{x_{j,b}^{ML}}\right)}}{\arg\min} \left\| \overline{y} - Rs \right\|^{2}$$

Using this reformulated problem statement, the triangular structure of R can be exploited to arrange a tree structure such that 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 goal 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. The sub-tree originating from a given node is searched 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 $d(x) \geq \lambda^{ML}$, 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 STS concludes once all of the tree nodes have been visited once or pruned.

Limitations

- The output LLR values are not scaled by the noise variance. For coded links employing iterative coding (LDPC or turbo) or MIMO OFDM with Viterbi decoding, the output LLR values should be scaled by the channel state information to achieve better performance.

Selected Bibliography

[1] Studer, C., A. Burg, and H. Bölcskei. “Soft-Output Sphere Decoding: Algorithms and VLSI Implementation”. *IEEE Journal of Selected Areas in Communications*. Vol. 26, No. 2, February 2008, pp. 290–300.

[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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Sphere Decoder | `comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.OSTBCCombiner`

Introduced in R2013a

step

System object: comm.SphereDecoder
Package: comm

Decode received symbols using sphere decoding algorithm

Syntax

```
Y = step(H, RXSYMBOLS, CHAN)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.SymbolSynchronizer System object

Package: comm

Correct symbol timing clock skew

Description

The `SymbolSynchronizer` System object corrects for the clock skew between a single-carrier transmitter and receiver.

To correct for symbol timing clock skew:

1. Define and set up the `SymbolSynchronizer` object. See “Construction” on page 3-1467.
2. Call `step` to correct for the clock skew between a transmitter and receiver according to the properties of `comm.SymbolSynchronizer`. The length of the output vector varies depending on the timing error and the synchronizer properties. The behavior of `step` is specific to each object in the toolbox.

The listed characteristics apply to the symbol synchronizer:

- PAM, PSK, and QAM modulation schemes are supported.
- The input operates on a sample rate basis, while the output signal operates on a symbol rate basis.
- The damping factor, normalized loop bandwidth, and detector gain properties are tunable so that they can be easily adjusted to improve synchronizer performance.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`S = comm.SymbolSynchronizer` creates a symbol synchronizer System object, `S`, that corrects for the clock skew between a single-carrier transmitter and receiver.

`S = comm.SymbolSynchronizer(Name,Value)` creates a symbol synchronizer object, `S`, 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

`Modulation`

Modulation type

Specify the modulation type as `'PAM/PSK/QAM'`, or `'OQPSK'`. The default value is `'PAM/PSK/QAM'`. This property is nontunable.

`TimingErrorDetector`

Timing error detector

Specify the timing error detection method as `Zero-Crossing (decision-directed)`, `Gardner (non-data-aided)`, `Early-Late (non-data-aided)`, or `Mueller-Muller (decision-directed)`. The selection controls the timing error detection scheme used in the synchronizer. The default value is `Zero-Crossing ( decision-directed)`. This property is nontunable.

`SamplesPerSymbol`

Samples per symbol

Specify the number of samples per symbol, s, as a real positive finite integer scalar, such that $s \geq 2$. The default value is `2`. This property is nontunable.

`DampingFactor`

Damping factor of the loop filter

Specify the damping factor of the loop filter as a positive real positive finite scalar. The default value is `1`. This property is tunable.

NormalizedLoopBandwidth

Normalized bandwidth of the loop filter

Specify the normalized loop bandwidth as a real scalar having a value between 0 and 1. The loop bandwidth is normalized by the sample rate of the input signal. The default value is `0.01`. This property is tunable.

Note It is recommended to set `NormalizedLoopBandwidth` to less than `0.1` to ensure that the symbol synchronizer locks.

DetectorGain

Gain of the phase detector

Specify the detector gain as a real positive finite scalar. The default value is `2.7`. This property is tunable.

Methods

reset	Reset states of the symbol synchronizer object
step	Correct symbol timing clock skew

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Correct QPSK Signal for Timing Error

Correct for a fixed symbol timing error on a noisy QPSK signal.

Create raised cosine transmit and receive filter System objects™.

```
txfilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',4);
```

```
rxfilter = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',4, ...
    'DecimationFactor',2);
```

Create a delay object to introduce a fixed timing error of 2 samples, which is equivalent to 1/2 symbols.

```
fixedDelay = dsp.Delay(2);
```

Create a `SymbolSynchronizer` object to eliminate the timing error.

```
symbolSync = comm.SymbolSynchronizer;
```

Generate random 4-ary symbols and apply QPSK modulation.

```
data = randi([0 3],5000,1);
modSig = pskmod(data,4,pi/4);
```

Filter the modulated signal through a raised cosine transmit filter and apply a fixed delay.

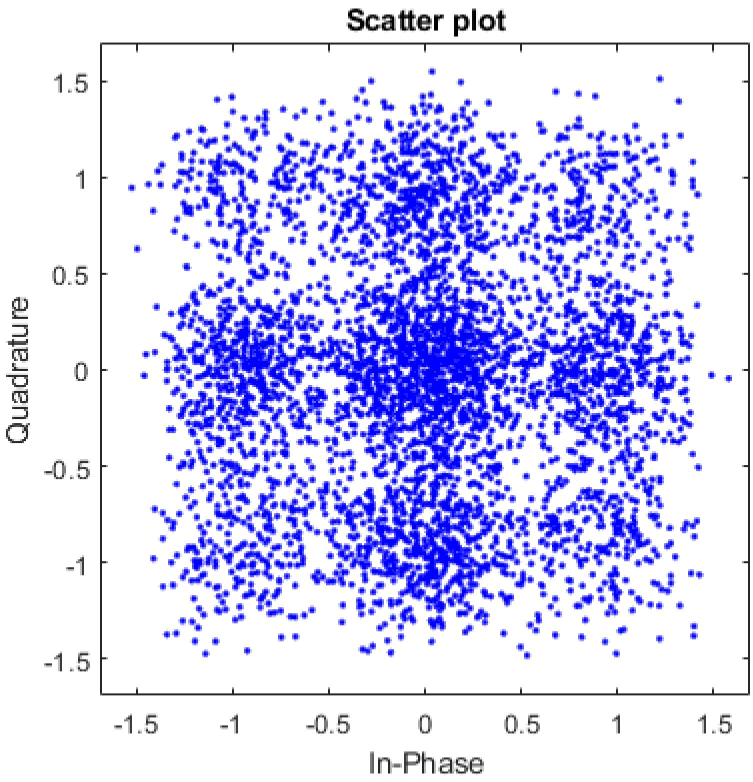
```
txSig = txfilter(modSig);
delaySig = fixedDelay(txSig);
```

Pass the delayed signal through an AWGN channel with a 15 dB signal-to-noise ratio.

```
rxSig = awgn(delaySig,15,'measured');
```

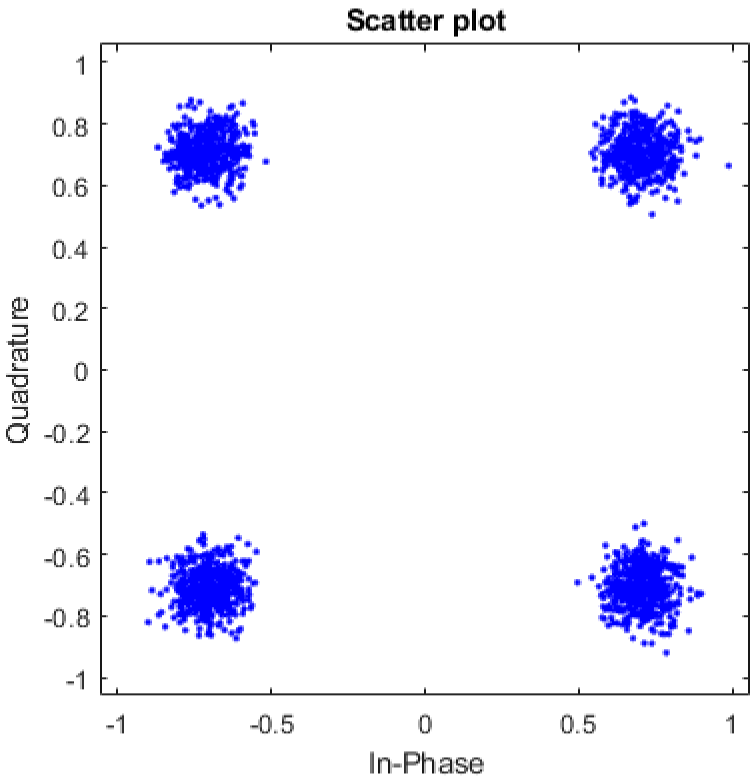
Filter the received signal and display its scatter plot. The scatter plot shows that the received signal does not align with the expected QPSK reference constellation due to the timing error.

```
rxSample = rxfilter(rxSig);
scatterplot(rxSample(1001:end),2)
```



Correct for the symbol timing error by using the symbol synchronizer object. Display the scatter plot and observe that the synchronized signal now aligns with the expected QPSK constellation.

```
rxSync = symbolSync(rxSample);
scatterplot(rxSync(1001:end),2)
```



Correct for Symbol Timing and Doppler Offsets

Recover from symbol timing and frequency offset errors by using the `comm.CarrierSynchronizer` and `comm.SymbolSynchronizer` objects.

Configure Example

Specify the samples per symbol parameter. Create a matched pair of raised cosine filter objects.

```
sps = 8;
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',10, ...
    'OutputSamplesPerSymbol',sps,'Gain',sqrt(sps));
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',10, ...
    'InputSamplesPerSymbol',sps,'DecimationFactor',sps/2,'Gain',sqrt(1/sps));
```

Create a `PhaseFrequencyOffset` object to introduce a 100 Hz Doppler shift.

```
doppler = comm.PhaseFrequencyOffset('FrequencyOffset',100, ...
    'PhaseOffset',45,'SampleRate',1e6);
```

Create a variable delay object to introduce timing offsets.

```
varDelay = dsp.VariableFractionalDelay;
```

Create carrier and symbol synchronizer objects to correct for a Doppler shift and a timing offset, respectively.

```
carrierSync = comm.CarrierSynchronizer('SamplesPerSymbol',2);
symbolSync = comm.SymbolSynchronizer(...
    'TimingErrorDetector','Early-Late (non-data-aided)', ...
    'SamplesPerSymbol',2);
```

Create constellation diagram objects to view the results.

```
refConst = qammod(0:15,16,'UnitAveragePower',true);

cdReceive = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',sps,'Title','Received Signal');

cdDoppler = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',2,'Title','Frequency Corrected Signal');

cdTiming = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',2,'Title','Frequency and Timing Synchronized Signal');
```

Main Processing Loop

Perform the following operations:

- Generate random symbols and apply QAM modulation.
- Filter the modulated signal.
- Apply frequency and timing offsets.
- Pass the transmitted signal through an AWGN channel.
- Correct for the Doppler shift.

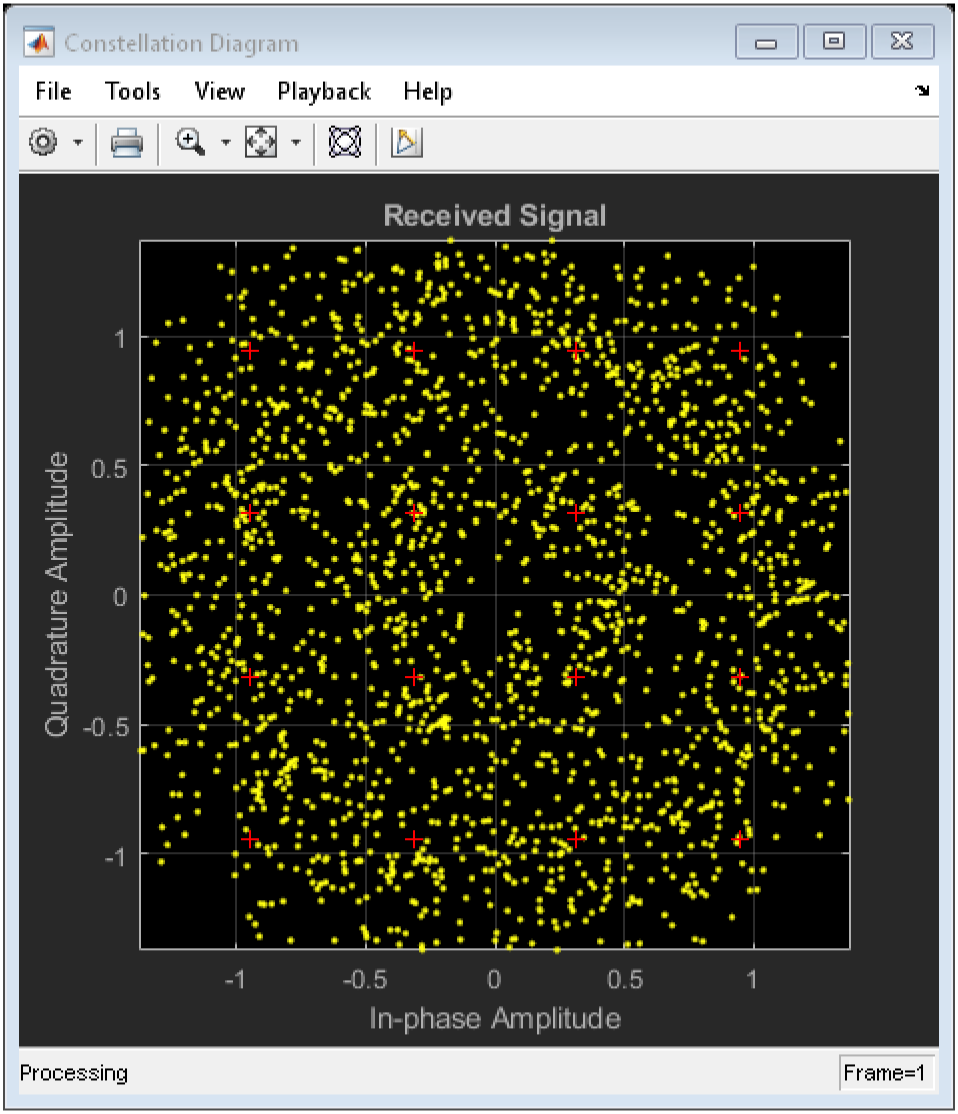
- Filter the received signal.
- Correct for the timing offset.

```
for k = 1:15
    data = randi([0 15],2000,1);
    modSig = qammod(data,16,'UnitAveragePower',true);
    txSig = txfilter(modSig);

    txDoppler = doppler(txSig);
    txDelay = varDelay(txDoppler,k/15);

    rxSig = awgn(txDelay,25);

    rxFiltSig = rxfilter(rxSig);
    rxCorr = carrierSync(rxFiltSig);
    rxData = symbolSync(rxCorr);
end
```

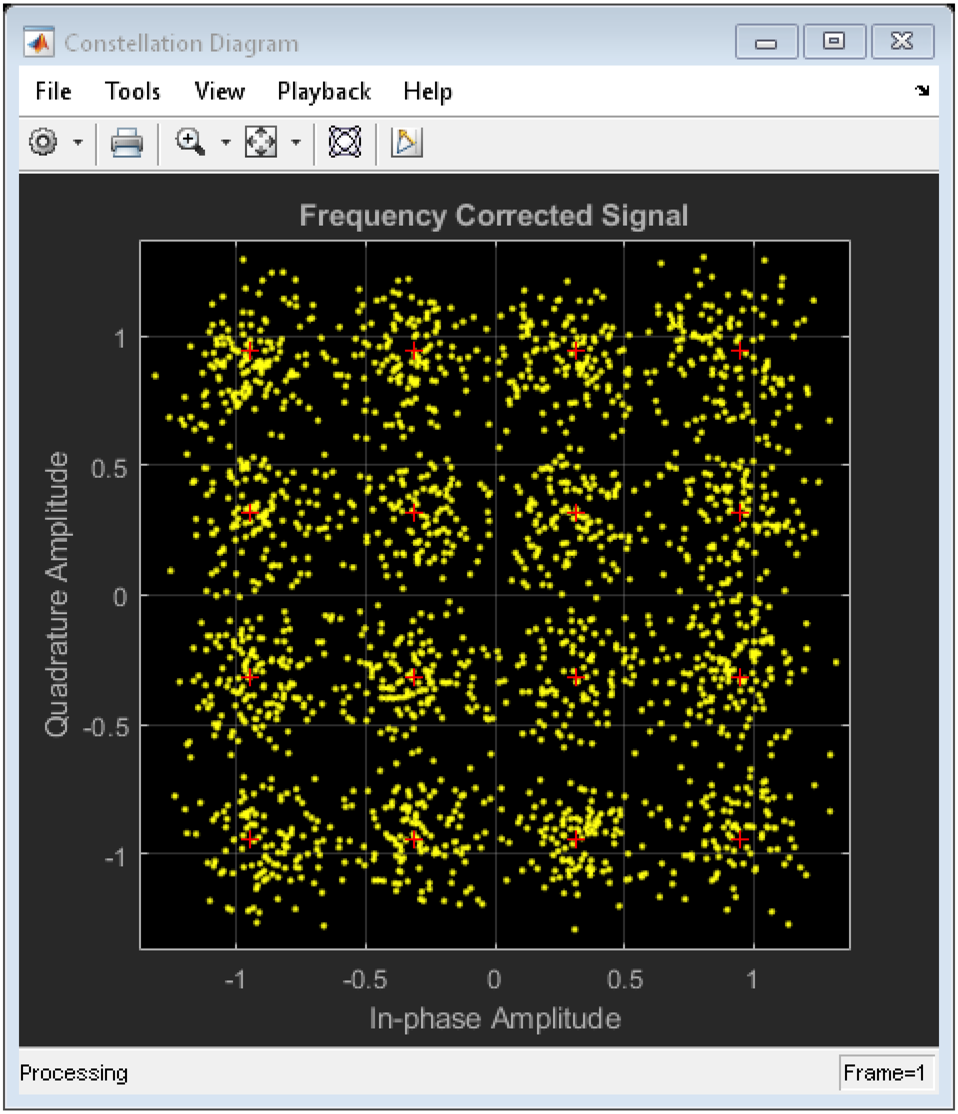
Visualization

Plot the constellation diagrams of the received signal, the frequency corrected signal, and the frequency and timing synchronized signal. While specific constellation points cannot be identified in the received signal and only partially identified in the frequency corrected signal, the timing and frequency synchronized signal aligns with the expected QAM constellation points.

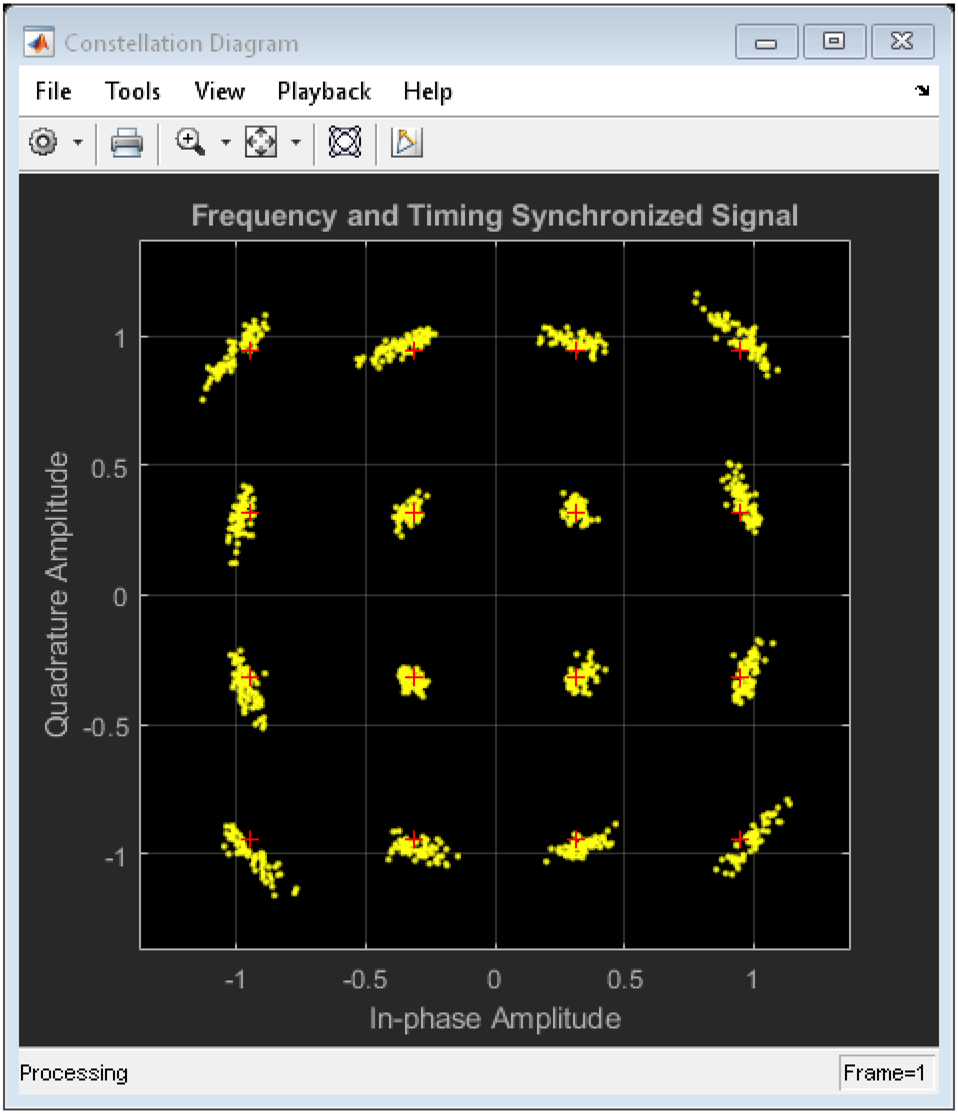
```
cdReceive(rxSig)
```



```
cdDoppler(rxCorr)
```



```
cdTiming(rxData)
```


Constellation Diagram
File
Tools
View
Playback
Help
Frequency and Timing Synchronized Signal
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Processing
Frame=1

Timing Error for Noisy 8-PSK Signal

Correct for a monotonically increasing symbol timing error on a noisy 8-PSK signal and display the normalized timing error.

Set example parameters.

```
fsym = 5000;                     % Symbol rate (Hz)
sps = 2;                         % Samples per symbol
fsamp = sps*fsym;                % Sample rate (Hz)
```

Create raised cosine transmit and receive filter System objects™.

```
txfilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',sps);
rxfilter = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',sps, ...
    'DecimationFactor',1);
```

Create a variable fractional delay object to introduce a monotonically increasing timing error.

```
varDelay = dsp.VariableFractionalDelay;
```

Create a `SymbolSynchronizer` object to eliminate the timing error.

```
symbolSync = comm.SymbolSynchronizer(...
    'TimingErrorDetector','Mueller-Muller (decision-directed)', ...
    'SamplesPerSymbol',sps);
```

Generate random 8-ary symbols and apply PSK modulation.

```
data = randi([0 7],5000,1);
modSig = pskmod(data,8,pi/8);
```

Filter the modulated signal through a raised cosine transmit filter and apply a monotonically increasing timing delay.

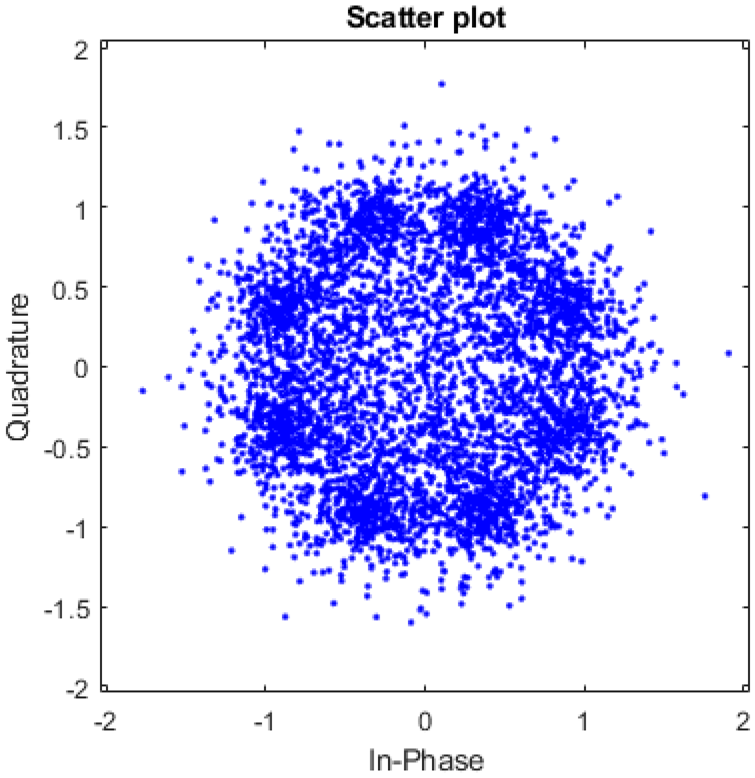
```
vdelay = (0:1/fsamp:1-1/fsamp)';
txSig = txfilter(modSig);
delaySig = varDelay(txSig,vdelay);
```

Pass the delayed signal through an AWGN channel with a 15 dB signal-to-noise ratio.

```
rxSig = awgn(delaySig,15,'measured');
```

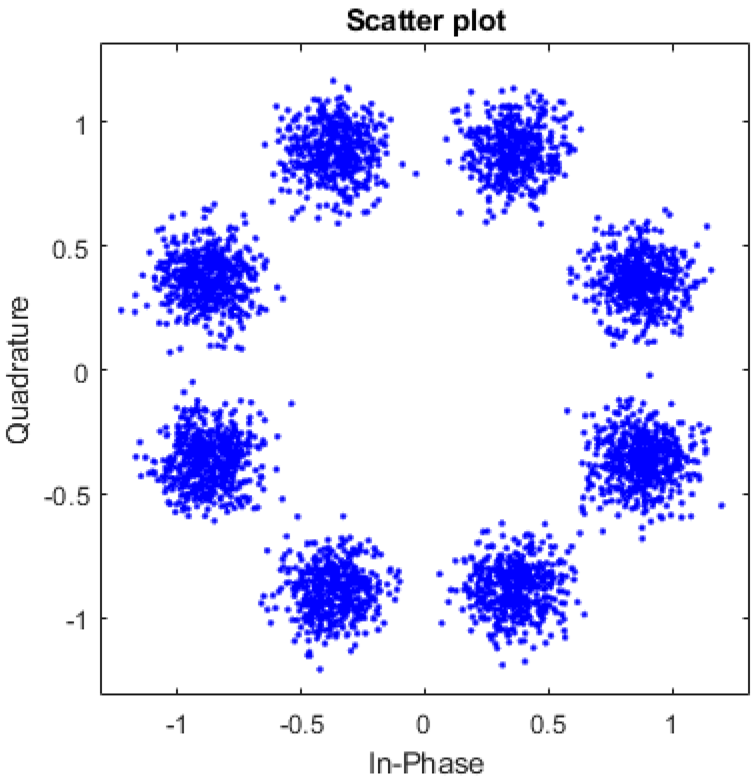
Filter the received signal and display its scatter plot. The scatter plot shows that the received signal does not align with the expected 8-PSK reference constellation due to the timing error.

```
rxSample = rxfilter(rxSig);
scatterplot(rxSample,sps)
```



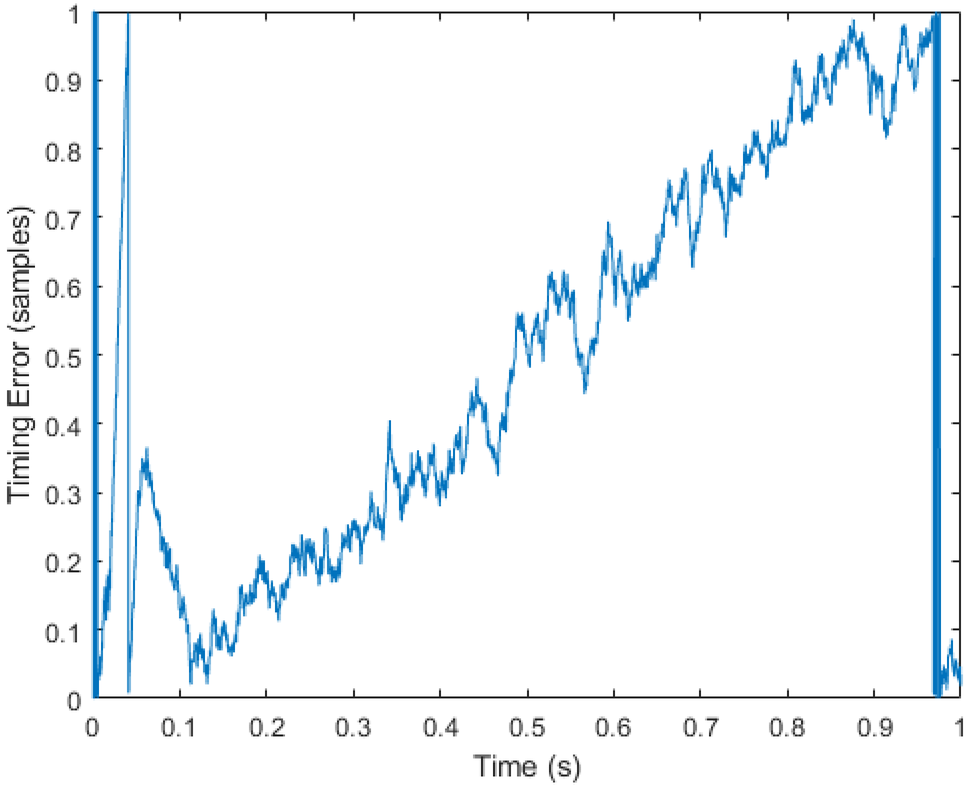
Correct for the symbol timing error by using the symbol synchronizer. Display the scatter plot and observe that the synchronized signal now aligns with the 8-PSK constellation.

```
[rxSym,tError] = symbolSync(rxSample);
scatterplot(rxSym(1001:end))
```



Plot the timing error estimate. You can see that the normalized timing error ramps up to 1 sample.

```
figure
plot(vdelay,tError)
xlabel('Time (s)')
ylabel('Timing Error (samples)')
```



Algorithms

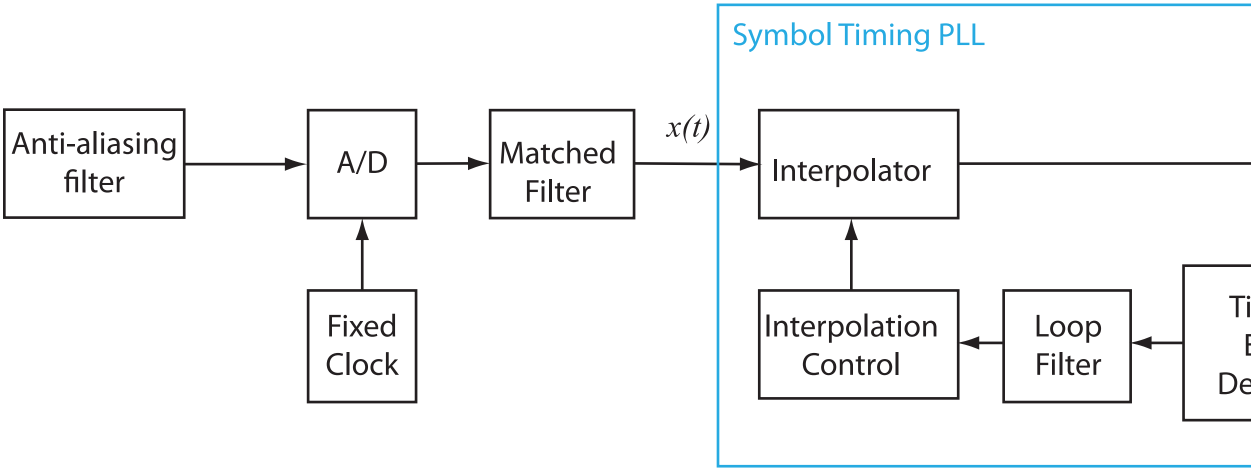
- “Overview” on page 3-1481
- “Timing Error Detection” on page 3-1481
- “Interpolator” on page 3-1483
- “Interpolation Control” on page 3-1484
- “Loop Filter” on page 3-1485

Overview

The symbol timing synchronizer is based on a phased lock loop (PLL) algorithm consisting of four components:

- A timing error detector (TED)
- An interpolator
- An interpolation controller
- A loop filter
- For OQPSK, the in-phase and quadrature signal components are first aligned (as in QPSK) using a buffer (state) to cache the last half symbol of the previous input. After initial alignment, the remaining synchronization processing is QPSK.

A generalized diagram of a timing synchronizer is shown below.



In the figure, *x(t)* is the received sample signal after the matched filter and $x(kT_s + \tau)$ is the symbol signal corrected for the clock skew between the transmitter and receiver.

Timing Error Detection

The `SymbolSynchronizer` object supports four types of timing error detection: Zero-Crossing, Gardner, Early-Late, and Mueller-Muller.

A non-data-aided timing error detector (TED) uses received samples without any knowledge of the transmitted signals and performing any estimation.

A decision-directed TED uses sign function to estimate the in-phase and quadrature components of received samples.

The `Zero-Crossing (decision-directed)` and `Mueller-Muller (decision-directed)` methods estimate the timing error based on the sign of the in-phase and quadrature components of signals passed to the synchronizer. As a result, the decision-directed methods are not recommended for constellations that have points with either a zero in-phase or quadrature component. For example, QPSK modulation with a zero phase offset having points at {`1+0i`, `0+1i`, `-1+0i`, and `0−1i`} would not be suitable for these methods.

In the table, $x(kT_s+\tau)$ and $y(kT_s+\tau)$ are the in-phase and quadrature components of the signals input to the timing error detector, where $\hat{\tau}$ is the estimated timing error. The coefficients $\hat{a}_0(k)$ and $\hat{a}_1(k)$ are the estimates of $x(kT_s+\tau)$ and $y(kT_s+\tau)$. These estimates are made by applying the `sign` function to the in-phase and quadrature components and are used only for the decision-directed TED modes.

TED Type	Expression
Zero-Crossing	$e(k)=x\left((k-1/2)T_s+\tau\right)\left[a_0(k-1)-a_0(k)\right]+y\left((k-1/2)T_s+\tau\right)\left[a_1(k-1)-a_1(k)\right]$
Gardner	$e(k)=x\left((k-1/2)T_s+\tau\right)\left[x\left((k-1)T_s+\tau\right)-x(kT_s+\tau)\right]+y\left((k-1/2)T_s+\hat{\tau}\right)\left[y\left((k-1)T_s+\hat{\tau}\right)-y(kT_s+\hat{\tau})\right]$
Early-Late	$e(k)=x(kT_s+\tau)\left[x\left((k+1/2)T_s+\tau\right)-x\left((k-1/2)T_s+\tau\right)\right]+y(kT_s+\tau)\left[y\left((k+1/2)T_s+\tau\right)-y\left((k-1/2)T_s+\tau\right)\right.$
Mueller-Muller	$e(k)=a_0(k-1)x(kT_s+\tau)-a_0(k)x\left((k-1)T_s+\tau\right)+a_1(k-1)y(kT_s+\tau)-a_1(k)y\left((k-1)T_s+\tau\right)$

The Zero-Crossing method is a decision-directed technique that requires two samples per symbol at the synchronizer’s input. It performs well in low SNR conditions for all values of excess bandwidth, and in moderate SNR conditions for moderate excess bandwidth factors (~0.4 - ~0.6).

The Gardner method is a non-data-aided feedback method that is independent of carrier phase recovery. It is suitable for both baseband systems and modulated carrier systems.

More specifically, this method is suitable for systems that use a linear modulation type with Nyquist pulses that have an excess bandwidth between approximately 40% and 100%. Examples of suitable systems are those that use pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), and that shape the signal using raised cosine filters whose rolloff factor is between 0.4 and 1. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

Gardner's method is similar to the early-late gate method.

The Early-Late method is also a non-data-aided feedback method. It is suitable for systems that use a linear modulation type, such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), with Nyquist pulses (for example, using a raised cosine filter). In the presence of noise, the performance of this timing recovery method improves as the pulse's excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

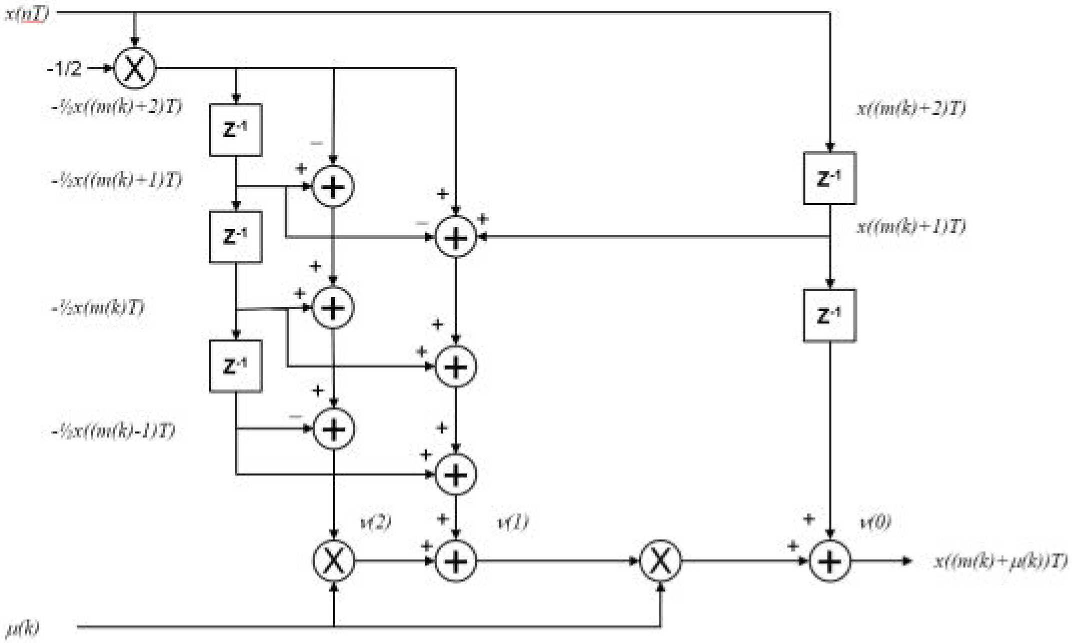
The Early-Late method is similar to the Gardner method. Compared to the Gardner method, the Early-Late method has higher self noise and thus does not perform as well as the Gardner method in systems with high SNR values.

The Mueller-Muller method is a decision-directed feedback method that requires prior recovery of the carrier phase.

When the input signal has Nyquist pulses (for example, using a raised cosine filter), this method has no self noise. In the presence of noise, the performance of the Mueller-Muller method improves as the pulse's excess bandwidth factor decreases, making the method a good candidate for narrowband signaling.

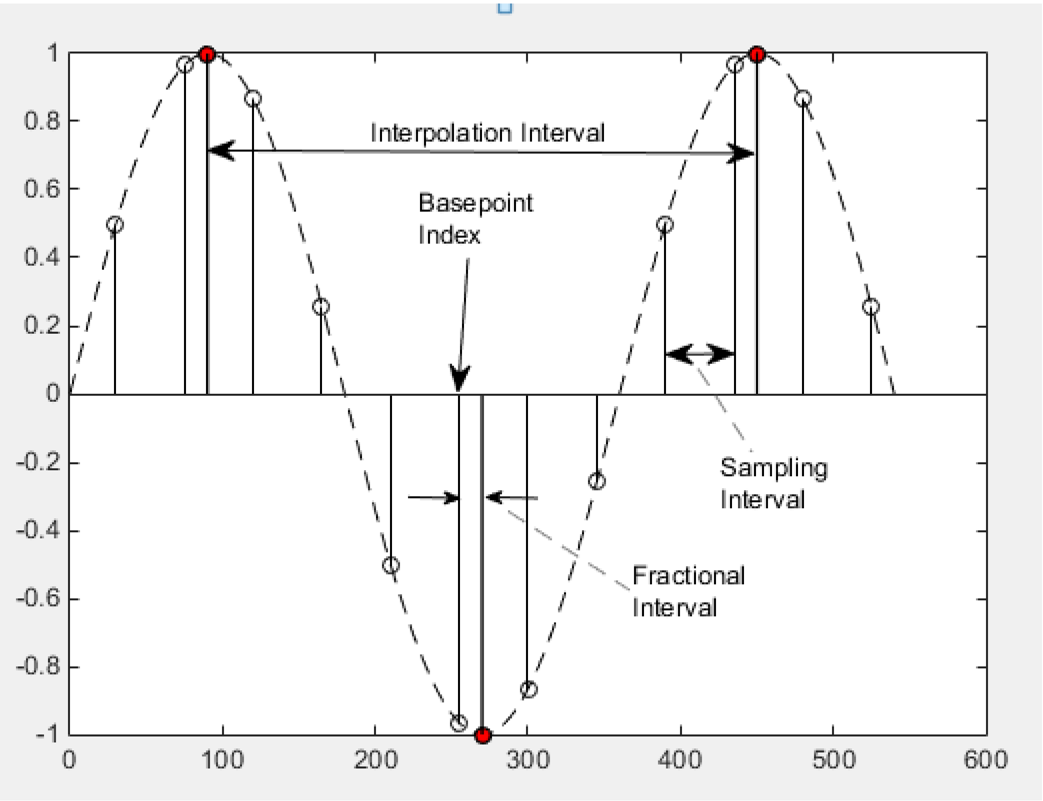
Interpolator

The time delay is estimated from fixed-rate samples of the matched filter, which are asynchronous with the symbol rate. As the resultant samples are not aligned with the symbol boundaries, an interpolator is used to “move” the samples. Because the time delay is unknown, the interpolator must be adaptive. Moreover, because the interpolant is a linear combination of the available samples, it can be thought of as the output of a filter. Consequently, a piecewise parabolic interpolator with a Farrow structure and coefficient α set to 1/2 as described in [1] is used by the System object.



Interpolation Control

Interpolation control provides the interpolator with the basepoint index and the fractional interval, where the basepoint index is the sample index nearest to the interpolant and the fractional interval is the ratio of the time between the interpolant and its basepoint index and the interpolation interval.



Interpolation is performed for every sample, where a strobe signal is used to determine if the interpolant is output. The object uses a modulo-1 counter interpolation control to provide the strobe and the fractional interval for use with the interpolator.

Loop Filter

A proportional-plus integrator (PI) loop filter of the form shown below is used. The proportional gain, K_1 and the integrator gain, K_2, are calculated by

$$K_1 = \frac{-4\zeta\theta}{\left(1 + 2\zeta\theta + \theta^2\right)K_p}$$

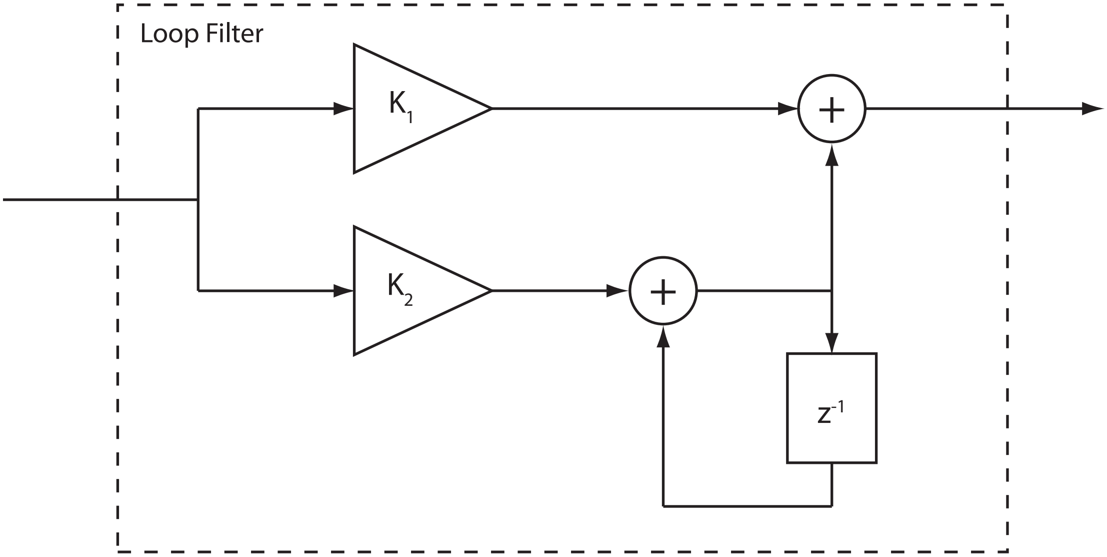
and

$$K_2 = \frac{-4\theta^2}{\left(1+2\zeta\theta+\theta^2\right)K_p} .$$

The interim term, θ, is given by

$$\theta = \frac{\frac{B_n T_s}{N}}{\zeta + \frac{1}{4\zeta}} ,$$

where N, ζ, B_nT_s, and K_p correspond to the `SamplesPerSymbol`, `DampingFactor`, `NormalizedLoopBandwidth`, and `DetectorGain` properties, respectively.



Selected Bibliography

[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 434–513.

[2] Mengali, Umberto and Aldo N. D’Andrea. *Synchronization Techniques for Digital Receivers*. New York: Plenum Press, 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer`

Introduced in R2015a

reset

System object: comm.SymbolSynchronizer
Package: comm

Reset states of the symbol synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `SymbolSynchronizer` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: comm.SymbolSynchronizer
Package: comm

Correct symbol timing clock skew

Syntax

```
Y = step(H,X)
[Y,P] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` corrects for symbol timing clock skew given input signal, `X`, and returns a corrected signal, `Y`. The input `X` is complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` is variable-sized and has the same data type as `X`. If `X` has dimensions of N_{samp}-by-1, `Y` will have dimensions of N_{sym}-by-1, where N_{sym} is approximately equal to N_{samp} divided by the `SamplesPerSymbol` property of `H`. The maximum output size is $\left\lceil \frac{N_{samp}}{N_{sps}} \times 1.1 \right\rceil$, where N_{sps} is equal to the `SamplesPerSymbol` property. The output is truncated if it exceeds that limit.

`[Y,P] = step(H,X)` returns a timing error vector, `P`, normalized by the input sample time. `P` is real and has the same data type and dimensions as `X` .

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ThermalNoise System object

Package: comm

Add thermal noise to signal

Description

The `ThermalNoise` object simulates the effects of thermal noise on a complex, baseband signal.

To add thermal noise to a complex, baseband signal:

1. Define and set up your thermal noise object. See “Construction” on page 3-1491.
2. Call `step` to add thermal noise according to the properties of `comm.ThermalNoise`.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`tn = comm.ThermalNoise` creates a receiver thermal noise System object, `H`. This object adds thermal noise to the complex, baseband input signal.

`tn = 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 used to set noise power

`'Noise temperature'` (default) | `'Noise figure'` | `'Noise factor'`

Method used to set the noise power, specified as `'Noise temperature'`, `'Noise figure'`, or `'Noise factor'`.

`NoiseTemperature` — Receiver noise temperature

`290` (default) | nonnegative real scalar

Receiver noise temperature, specified in degrees K as a nonnegative real scalar. This property is available when `NoiseMethod` is equal to `'Noise temperature'`. Noise temperature is typically used to characterize satellite receivers because the input noise temperature can vary and is often less than 290 K. Tunable.

`NoiseFigure` — Noise figure

`3.01` (default) | nonnegative real scalar

Noise figure, specified in dB as a nonnegative real scalar. This property is available when `NoiseMethod` is equal to `'Noise figure'`. Noise figure describes the performance of a receiver and does not include the effect of the antenna. It is defined only for an input noise temperature of 290 K. The noise figure is the dB equivalent of the noise factor. Tunable.

`NoiseFactor` — Noise factor

`2` (default) | real scalar ≥ 1

Noise factor, specified as a real scalar greater than or equal to 1. This property is available when `NoiseMethod` is equal to `'Noise factor'`. Noise factor describes the performance of a receiver and does not include the effect of the antenna. It is defined only for an input noise temperature of 290 K. The noise factor is the linear equivalent of the noise figure. Tunable.

`SampleRate` — Sample rate

`1` (default) | positive real scalar

Sample rate, specified as in Hz as a positive real scalar. The object computes the variance of the noise added to the input signal as $kT\times$`SampleRate`. The value k is Boltzmann's constant and T is the noise temperature specified explicitly or implicitly via one of the noise methods.

`Add290KAntennaNoise` — Add 290 K antenna noise

`false` (default) | `true`

Add 290 K antenna noise to the input signal, specified as a logical scalar. To add 290 K antenna noise, set this property to `true`. This property is available when `NoiseMethod` is equal to `'Noise factor'` or `'Noise figure'`.

The total noise applied to the input signal is the sum of the circuit noise and the antenna noise.

Methods

step	Add receiver thermal noise

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Add Thermal Noise to QPSK Signal

Create a thermal noise object having a noise temperature of 290 K and a sample rate of 5 MHz.

```
thNoise = comm.ThermalNoise('NoiseTemperature',290,'SampleRate',5e6);
```

Generate QPSK-modulated data having an output power of 20 dBm.

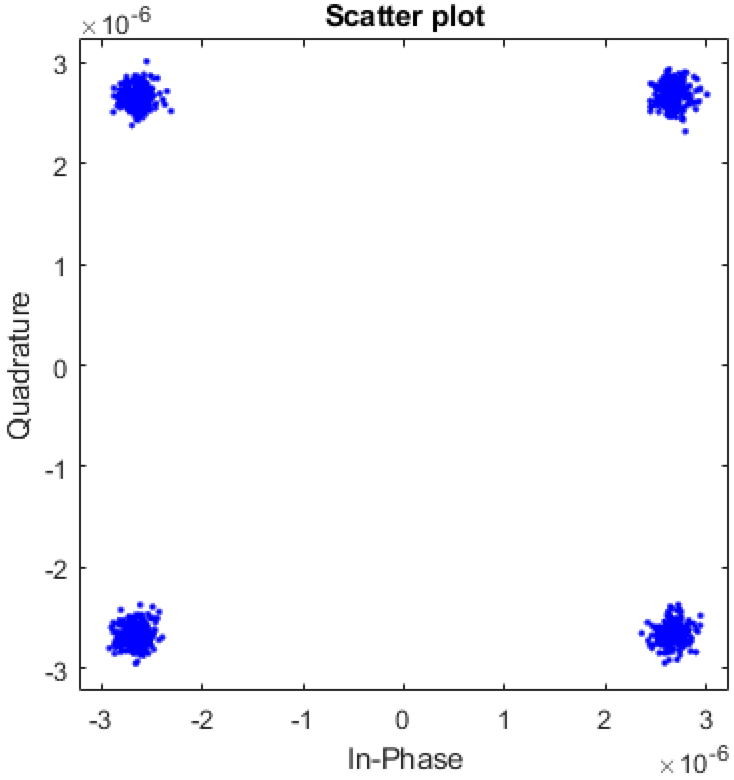
```
data = randi([0 3],1000,1);
modData = 0.3162*pskmod(data,4,pi/4);
```

Attenuate the signal by the free space path loss assuming a 1000 m link distance and a carrier frequency of 2 GHz.

```
fsl = (4*pi*1000*2e9/3e8)^2;
rxData = modData/sqrt(fsl);
```

Add thermal noise to the signal. Plot the noisy constellation.

```
noisyData = thNoise(rxData);
scatterplot(noisyData)
```



Add Antenna and Receiver Thermal Noise to 16-QAM Signal

Create a thermal noise object having a 5 dB noise figure and a 10 MHz sample rate. Specify that the 290 K antenna noise be included.

```
thermalNoise = comm.ThermalNoise('NoiseMethod','Noise figure', ...
    'NoiseFigure',5, ...
    'SampleRate',10e6, ...
    'Add290KAntennaNoise',true);
```

Generate QPSK-modulated data having a 1 W output power.

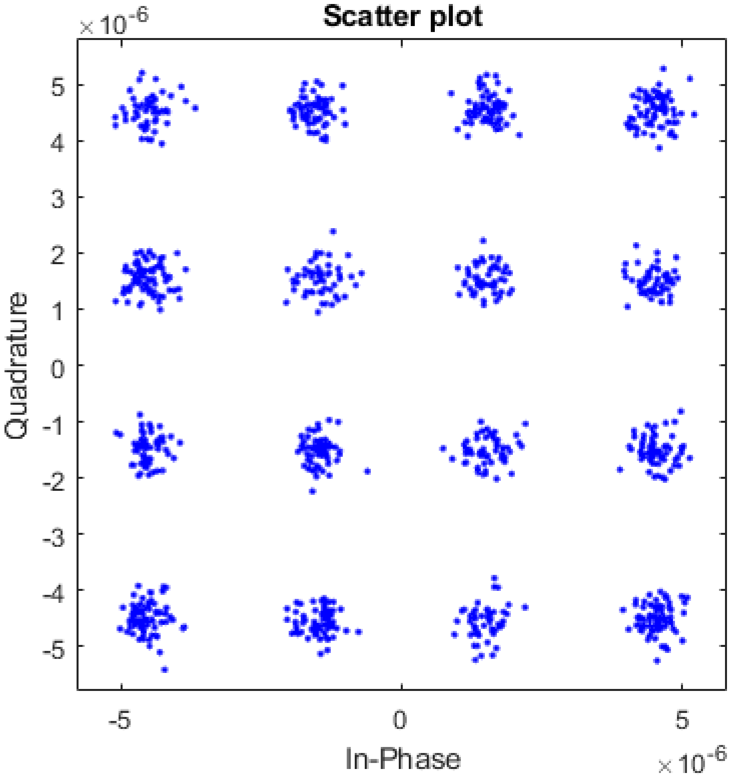
```
data = randi([0 15],1000,1);
modSig = qammod(data,16,'UnitAveragePower',true);
```

Attenuate the signal by the free space path loss assuming a 1 km link distance and a 5 GHz carrier frequency.

```
fsl = (4*pi*1000*5e9/3e8)^2;
rxSig = modSig/sqrt(fsl);
```

Add thermal noise to the signal and plot its constellation.

```
noisySig = thermalNoise(rxSig);
scatterplot(noisySig)
```



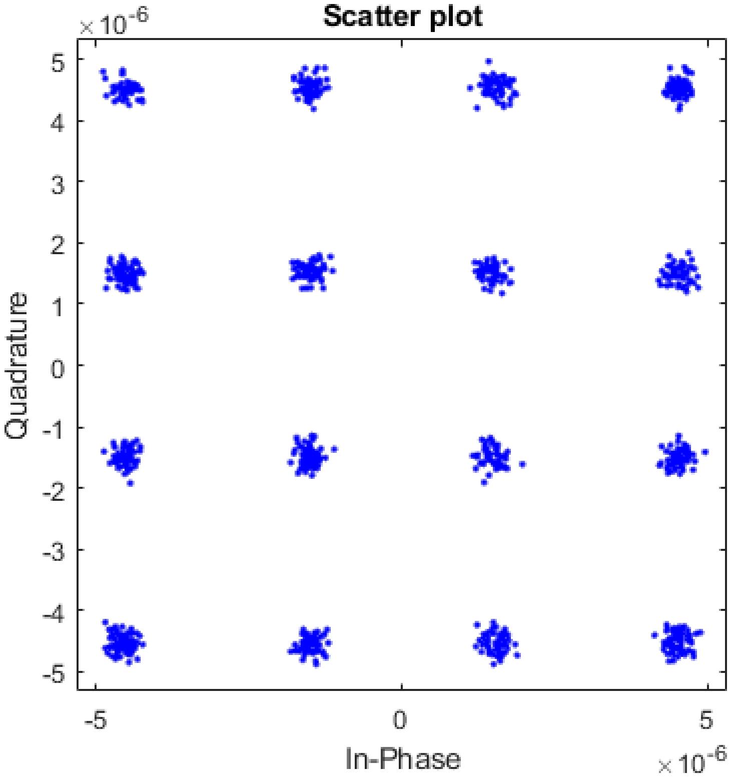
Estimate the SNR.

```
mer = comm.MER;
snrEst1 = mer(rxSig,noisySig)
```

```
snrEst1 = 22.6611
```

Decrease the noise figure to 0 dB, and plot the resultant received signal. Because antenna noise is included, the signal is not completely noiseless.

```
thermalNoise.NoiseFigure = 0;
noisySig = thermalNoise(rxSig);
scatterplot(noisySig)
```



Estimate the SNR. The SNR is 5 dB higher than in the first case, which is expected given the 5 dB decrease in the noise figure.

```
snrEst2 = mer(rxSig,noisySig)
```

```
snrEst2 = 27.8658
```

```
snrEst2 - snrEst1
```

```
ans = 5.2047
```

Algorithms

Wireless receiver performance is often expressed as a noise factor or figure. The noise factor is defined as the ratio of the input signal-to-noise ratio, S_i/N_i to the output signal-to-noise ratio, S_o/N_o, such that

$$F = \frac{S_i/N_i}{S_o/N_o}.$$

Given receiver gain G and receiver noise power N_{ckt}, the noise factor can be expressed as

$$\begin{aligned} F &= \frac{S_i/N_i}{GS_i/\left(N_{ckt}+GN_i\right)} \\ &= \frac{N_{ckt}+GN_i}{GN_i}. \end{aligned}$$

The IEEE defines the noise factor assuming that noise temperature at the input is T_0, where T_0 = 290 K. The noise factor is then

$$\begin{aligned} F &= \frac{N_{ckt}+GN_i}{GN_i} \\ &= \frac{GkBT_{ckt}+GkBT_0}{GkBT_0} \\ &= \frac{T_{ckt}+T_0}{T_0}. \end{aligned}$$

T_{ckt} is the equivalent input noise temperature of the receiver and is expressed as

$$T_{ckt} = T_0 (F - 1) \, .$$

The overall noise temperature of an antenna and receiver, T_{sys}, is

$$T_{sys} = T_{ant} + T_{ckt} \; ,$$

where T_{ant} is the antenna noise temperature.

The noise figure, *NF*, is the dB equivalent of the noise factor and can be expressed as

$$NF = 10 \log_{10} (F) \, .$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AWGNChannel`

Introduced in R2012a

step

System object: comm.ThermalNoise
Package: comm

Add receiver thermal noise

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.TurboDecoder System object

Package: comm

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.

To decode an input signal using a turbo decoding scheme:

1. Define and set up your turbo decoder object. See “Construction” on page 3-1500.
2. Call `step` to decode a binary signal according to the properties of `comm.TurboDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Decode input signal using parallel concatenated decoding scheme

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit and Receive Turbo-Encoded Data over a BPSK-Modulated AWGN Channel

Simulate the transmission and reception of BPSK data over an AWGN channel using turbo encoding and decoding.

Set the Eb/No (dB) and frame length parameters. Set the random number generator to its default state to ensure that the results are repeatable.

```
EbNo= -6;
frmLen = 256;
rng default
```

Calculate the noise variance from the Eb/No ratio. Generate random interleaver indices.

```
noiseVar = 10^(-EbNo/10);
intrlvrIndices = randperm(frmLen);
```

Create a turbo encoder and decoder pair using the trellis structure given by `poly2trellis(4,[13 15 17],13)` and `intrlvrIndices`.

```
hTEnc = comm.TurboEncoder('TrellisStructure',poly2trellis(4, ...
    [13 15 17],13),'InterleaverIndices',intrlvrIndices);

hTDec = comm.TurboDecoder('TrellisStructure',poly2trellis(4, ...
    [13 15 17],13),'InterleaverIndices',intrlvrIndices, ...
    'NumIterations',4);
```

Create a BPSK modulator and demodulator pair, where the demodulator outputs soft bits determined by using a log-likelihood ratio method.

```
hMod = comm.BPSKModulator;
hDemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
hChan = comm.AWGNChannel('EbNo',EbNo);
hError = comm.ErrorRate;
```

The main processing loop performs the following steps:

- Generate binary data
- Turbo encode the data

- Modulate the encoded data
- Pass the modulated signal through an AWGN channel
- Demodulate the noisy signal using LLR to output soft bits
- Turbo decode the demodulated data. Because the bit mapping from the demodulator is opposite that expected by the turbo decoder, the decoder input must use the inverse of demodulated signal.
- Calculate the error statistics

```
for frmIdx = 1:100
    data = randi([0 1],frmLen,1);
    encodedData = step(hTEnc,data);
    modSignal = step(hMod,encodedData);
    receivedSignal = step(hChan,modSignal);
    demodSignal = step(hDemod,receivedSignal);
    receivedBits = step(hTDec,-demodSignal);
    errorStats = step(hError,data,receivedBits);
end
```

Display the error data.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', ...
    errorStats)
```

```
Bit error rate = 2.34e-04
Number of errors = 6
Total bits = 25600
```

Turbo Coding with 16-QAM Modulation in an AWGN Channel

Simulate an end-to-end communication link employing 16-QAM using turbo codes in an AWGN channel. The frame sizes are randomly selected from a set of {500, 1000, 1500}. Because the frame size varies, provide the interleaver indices to the turbo encoder and decoder objects as an input to their associated `step` functions.

Set the modulation order and Eb/No (dB) parameters. Set the random number generator to its default state to be able to repeat the results.

```
M = 16;
EbNo= -2;
rng default
```

Calculate the noise variance from the Eb/No ratio and the modulation order.

```
noiseVar = 10^(-EbNo/10)*(1/log2(M));
```

Create a turbo encoder and decoder pair, where the interleaver indices are supplied by an input argument to the `step` function.

```
hTEnc = comm.TurboEncoder('InterleaverIndicesSource','Input port');

hTDec = comm.TurboDecoder('InterleaverIndicesSource','Input port', ...
    'NumIterations',4);
```

Create a 16-QAM modulator and demodulator pair, where the demodulator outputs soft bits determined by using a log-likelihood ratio method. The modulator and demodulator objects are normalized to use an average power of 1 W.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',M, ...
    'BitInput',true, ...
    'NormalizationMethod','Average power');
hDemod = comm.RectangularQAMDemodulator('ModulationOrder',M, ...
    'BitOutput',true, ...
    'NormalizationMethod','Average power', ...
    'DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
hChan = comm.AWGNChannel('EbNo',EbNo,'BitsPerSymbol',log2(M));
hError = comm.ErrorRate;
```

The processing loop performs the following steps:

- Select a random frame size and generate interleaver indices
- Generate random binary data
- Turbo encode the data
- Apply 16-QAM modulation
- Pass the modulated signal through an AWGN channel
- Demodulate the noisy signal using an LLR algorithm
- Turbo decode the data
- Calculate the error statistics

```
for frmIdx = 1:100
```

```
    % Randomly select one of three possible frame sizes
    frmLen = 500*randi([1 3],1,1);

    % Determine the interleaver indices given the frame length
    intrlvrIndices = randperm(frmLen);

    % Generate random binary data
    data = randi([0 1],frmLen,1);

    % Turbo encode the data
    encodedData = step(hTEnc,data,intrlvrIndices);

    % Modulate the encoded data
    modSignal = step(hMod,encodedData);

    % Pass the signal through the AWGN channel
    receivedSignal = step(hChan,modSignal);

    % Demodulate the received signal
    demodSignal = step(hDemod,receivedSignal);

    % Turbo decode the demodulated signal. Because the bit mapping from the
    % demodulator is opposite that expected by the turbo decoder, the
    % decoder input must use the inverse of demodulated signal.
    receivedBits = step(hTDec,-demodSignal,intrlvrIndices);

    % Calculate the error statistics
    errorStats = step(hError,data,receivedBits);
end
```

Display the error statistics.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', ...
    errorStats)
```

```
Bit error rate = 3.51e-04
Number of errors = 33
Total bits = 94000
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Turbo Decoder | `comm.APPDecoder` | `comm.TurboEncoder`

Introduced in R2012a

step

System object: comm.TurboDecoder
Package: comm

Decode input signal using parallel concatenated decoding scheme

Syntax

```
Y = step(H,X)
Y = step(H, X, INTERLVRINDICES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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 System object

Package: comm

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.

To encode an input signal using a turbo coding scheme:

1. Define and set up your turbo encoder object. See “Construction” on page 3-1510.
2. Call `step` to encode a binary signal according to the properties of `comm.TurboEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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

step	Encode input signal using parallel concatenated coding scheme

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit and Receive Turbo-Encoded Data over a BPSK-Modulated AWGN Channel

Simulate the transmission and reception of BPSK data over an AWGN channel using turbo encoding and decoding.

Set the Eb/No (dB) and frame length parameters. Set the random number generator to its default state to ensure that the results are repeatable.

```
EbNo= -6;
frmLen = 256;
rng default
```

Calculate the noise variance from the Eb/No ratio. Generate random interleaver indices.

```
noiseVar = 10^(-EbNo/10);
intrlvrIndices = randperm(frmLen);
```

Create a turbo encoder and decoder pair using the trellis structure given by `poly2trellis(4,[13 15 17],13)` and `intrlvrIndices`.

```
hTEnc = comm.TurboEncoder('TrellisStructure',poly2trellis(4, ...
    [13 15 17],13),'InterleaverIndices',intrlvrIndices);

hTDec = comm.TurboDecoder('TrellisStructure',poly2trellis(4, ...
    [13 15 17],13),'InterleaverIndices',intrlvrIndices, ...
    'NumIterations',4);
```

Create a BPSK modulator and demodulator pair, where the demodulator outputs soft bits determined by using a log-likelihood ratio method.

```
hMod = comm.BPSKModulator;
hDemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
hChan = comm.AWGNChannel('EbNo',EbNo);
hError = comm.ErrorRate;
```

The main processing loop performs the following steps:

- Generate binary data
- Turbo encode the data
- Modulate the encoded data
- Pass the modulated signal through an AWGN channel
- Demodulate the noisy signal using LLR to output soft bits
- Turbo decode the demodulated data. Because the bit mapping from the demodulator is opposite that expected by the turbo decoder, the decoder input must use the inverse of demodulated signal.
- Calculate the error statistics

```
for frmIdx = 1:100
    data = randi([0 1],frmLen,1);
    encodedData = step(hTEnc,data);
    modSignal = step(hMod,encodedData);
    receivedSignal = step(hChan,modSignal);
    demodSignal = step(hDemod,receivedSignal);
    receivedBits = step(hTDec,-demodSignal);
    errorStats = step(hError,data,receivedBits);
end
```

Display the error data.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', ...
    errorStats)
```

```
Bit error rate = 2.34e-04
Number of errors = 6
Total bits = 25600
```

Turbo Coding with 16-QAM Modulation in an AWGN Channel

Simulate an end-to-end communication link employing 16-QAM using turbo codes in an AWGN channel. The frame sizes are randomly selected from a set of {500, 1000, 1500}. Because the frame size varies, provide the interleaver indices to the turbo encoder and decoder objects as an input to their associated `step` functions.

Set the modulation order and Eb/No (dB) parameters. Set the random number generator to its default state to be able to repeat the results.

```
M = 16;
EbNo= -2;
rng default
```

Calculate the noise variance from the Eb/No ratio and the modulation order.

```
noiseVar = 10^(-EbNo/10)*(1/log2(M));
```

Create a turbo encoder and decoder pair, where the interleaver indices are supplied by an input argument to the `step` function.

```
hTEnc = comm.TurboEncoder('InterleaverIndicesSource','Input port');

hTDec = comm.TurboDecoder('InterleaverIndicesSource','Input port', ...
    'NumIterations',4);
```

Create a 16-QAM modulator and demodulator pair, where the demodulator outputs soft bits determined by using a log-likelihood ratio method. The modulator and demodulator objects are normalized to use an average power of 1 W.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',M, ...
    'BitInput',true, ...
    'NormalizationMethod','Average power');
hDemod = comm.RectangularQAMDemodulator('ModulationOrder',M, ...
    'BitOutput',true, ...
    'NormalizationMethod','Average power', ...
    'DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
hChan = comm.AWGNChannel('EbNo',EbNo,'BitsPerSymbol',log2(M));
hError = comm.ErrorRate;
```

The processing loop performs the following steps:

- Select a random frame size and generate interleaver indices
- Generate random binary data
- Turbo encode the data
- Apply 16-QAM modulation
- Pass the modulated signal through an AWGN channel
- Demodulate the noisy signal using an LLR algorithm

- Turbo decode the data
- Calculate the error statistics

```
for frmIdx = 1:100

    % Randomly select one of three possible frame sizes
    frmLen = 500*randi([1 3],1,1);

    % Determine the interleaver indices given the frame length
    intrlvrIndices = randperm(frmLen);

    % Generate random binary data
    data = randi([0 1],frmLen,1);

    % Turbo encode the data
    encodedData = step(hTEnc,data,intrlvrIndices);

    % Modulate the encoded data
    modSignal = step(hMod,encodedData);

    % Pass the signal through the AWGN channel
    receivedSignal = step(hChan,modSignal);

    % Demodulate the received signal
    demodSignal = step(hDemod,receivedSignal);

    % Turbo decode the demodulated signal. Because the bit mapping from the
    % demodulator is opposite that expected by the turbo decoder, the
    % decoder input must use the inverse of demodulated signal.
    receivedBits = step(hTDec,-demodSignal,intrlvrIndices);

    % Calculate the error statistics
    errorStats = step(hError,data,receivedBits);
end
```

Display the error statistics.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', ...
    errorStats)
```

```
Bit error rate = 3.51e-04
Number of errors = 33
Total bits = 94000
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Turbo Encoder | `comm.ConvolutionalEncoder` | `comm.TurboDecoder`

Introduced in R2012a

step

System object: comm.TurboEncoder
Package: comm

Encode input signal using parallel concatenated coding scheme

Syntax

```
Y = step(H,X)
Y = step(H, X, INTERLVRINDICES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.ViterbiDecoder System object

Package: comm

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.

To decode input symbols and produce binary output symbols:

1. Define and set up your Viterbi decoder object. See “Construction” on page 3-1519.
2. Call `step` to decode input symbols according to the properties of `comm.ViterbiDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

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 1s and positive numbers to be 0s.

When you set this property to Hard, the input must be a vector of hard decision values, which are 0s or 1s. 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 0s and positive numbers to be 1s.

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 on page 3-0 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 on page 3-0 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 on page 3-0 property to Continuous, the decoding delay consists of TracebackDepth on page 3-0 zero symbols or TracebackDepth×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.

As a general estimate, a typical TracebackDepth property value is approximately two to three times $(k – 1)/(1 – r)$, where k is the constraint length of the code and r is the code rate [1]. For example:

- A rate 1/2 code has a TracebackDepth of $5(k – 1)$.
- A rate 2/3 code has a TracebackDepth of $7.5(k – 1)$.
- A rate 3/4 code has a TracebackDepth of $10(k – 1)$.

- A rate 5/6 code has a TracebackDepth of 15(k – 1).

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 on page 3-0 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 on page 3-0 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 on page 3-0 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 1s and 0s. The 0s 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 on page 3-0 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` on page 3-0 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` on page 3-0 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` on page 3-0 property to `Custom`.

Methods

reset	Reset states of the Viterbi decoder object
step	Decode convolutionally encoded data using Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Encode and Decode 8-DPSK Modulated Data

Transmit a convolutionally encoded 8-DPSK modulated bit stream through an AWGN channel. Then, demodulate and decode using a Viterbi decoder.

Create the necessary System objects.

```
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);
```

Process the data using the following steps:

1. Generate random bits
2. Convolutionally encode the data
3. Apply DPSK modulation
4. Pass the modulated signal through AWGN
5. Demodulate the noisy signal
6. Decode the data using a Viterbi algorithm
7. Collect error statistics

```
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
```

Display the number of errors.

```
errors(2)
```

```
ans = 3
```

Convolutional Encoding and Viterbi Decoding with a Puncture Pattern Matrix

Encode and decode a sequence of bits using a convolutional encoder and a Viterbi decoder with a defined puncture pattern. Verify that the input and output bits are identical

Define a puncture pattern matrix and reshape it into vector form for use with the Encoder and Decoder objects.

```
pPatternMat = [1 0 1;1 1 0];
pPatternVec = reshape(pPatternMat,6,1);
```

Create convolutional encoder and a Viterbi decoder in which the puncture pattern is defined by `pPatternVec`.

```
ENC = comm.ConvolutionalEncoder(...
    'PuncturePatternSource','Property', ...
    'PuncturePattern',pPatternVec);

DEC = comm.ViterbiDecoder('InputFormat','Hard', ...
    'PuncturePatternSource','Property',...
    'PuncturePattern',pPatternVec);
```

Create an error rate counter with the appropriate receive delay.

```
ERR = comm.ErrorRate('ReceiveDelay',DEC.TracebackDepth);
```

Encode and decode a sequence of random bits.

```
dataIn = randi([0 1],600,1);

dataEncoded = step(ENC,dataIn);

dataOut = step(DEC,dataEncoded);
```

Verify that there are no errors in the output data.

```
errStats = step(ERR,dataIn,dataOut);
errStats(2)
```

```
ans = 0
```

- “LLR vs. Hard Decision Demodulation”

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` on page 3-0 property.
- The **Operation mode** parameter corresponds to the `TerminationMethod` on page 3-0 property.

References

[1] Moision, B., “A Truncation Depth Rule of Thumb for Convolutional Codes,” *Information Theory and Applications Workshop*, pp. 555–557, 2008.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.APPDecoder` | `comm.ConvolutionalEncoder`

Topics

“LLR vs. Hard Decision Demodulation”

Introduced in R2012a

reset

System object: comm.ViterbiDecoder
Package: comm

Reset states of the Viterbi decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ViterbiDecoder` object, `H`.

step

System object: comm.ViterbiDecoder
Package: comm

Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,ERASURES)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.WalshCode System object

Package: comm

Generate Walsh code from orthogonal set of codes

Description

The `WalshCode` object generates a Walsh code from an orthogonal set of codes.

To generate a Walsh code:

1. Define and set up your Walsh code object. See “Construction” on page 3-1532.
2. Call `step` to encode the input signal according to the properties of `comm.WalshCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

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` on page 3-0 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` on page 3-0 property.

OutputDataType

Data type of output

Specify the output data type as `double` | `int8`. The default is `double`.

Methods

reset	Reset states of Walsh code generator object
step	Generate Walsh code from orthogonal set of codes

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Walsh Code Sequence

Generate 16 samples of a length-64 Walsh code sequence.

```
walsh = comm.WalshCode('SamplesPerFrame',16)
```

```
walsh = 
  comm.WalshCode with properties:

             Length: 64
              Index: 60
    SamplesPerFrame: 16
     OutputDataType: 'double'
```

```
seq = walsh()
```

```
seq = 16×1

     1
    -1
     1
    -1
     1
    -1
     1
    -1
     1
    -1
      ⋮
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HadamardCode` | `comm.OVSFCode`

Introduced in R2012a

reset

System object: comm.WalshCode
Package: comm

Reset states of Walsh code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `WalshCode` object, `H`.

step

System object: comm.WalshCode
Package: comm

Generate Walsh code from orthogonal set of codes

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`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 *N*x*N* 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 `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) 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.WINNER2Channel System object

Package: comm

Filter input signal through WINNER II fading channel

Download Required

To use `comm.WINNER2Channel`, first download the WINNER II Channel Model for Communications System Toolbox from the Add-On Explorer. For more information on downloading add-ons, see "Get Add-Ons" (MATLAB) and "Manage Your Add-Ons" (MATLAB).

Description

The `comm.WINNER2Channel` System object filters an input signal through a WINNER II fading channel. The object utilizes the basic model defined and provided by the WINNER II Channel Models [1].

To filter an input signal using a WINNER II fading channel:

1. Define and set up your WINNER II channel object. See "Construction" on page 3-1538.
2. Call `step` to filter the input signal through a WINNER II fading channel according to the properties of `comm.WINNER2Channel`.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`chan = comm.WINNER2Channel` creates a WINNER II fading channel System object to model single or multiple links. `chan` generates channel coefficients using the WINNER II spatial channel model (SCM). It also filters a real or complex input signal through the fading channel for each link.

`chan = comm.WINNER2Channel(Name,Value)` creates a WINNER II fading channel object, `chan`, that overrides default values using one or more `Name,Value` pair arguments. You can specify additional name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`chan = comm.WINNER2Channel(cfgModel)` creates a WINNER II fading channel object with the `ModelConfig` property set to `cfgModel`.

`chan = comm.WINNER2Channel(cfgModel,cfgLayout)` creates a WINNER II fading channel object with the `ModelConfig` property set to `cfgModel` and the `LayoutConfig` property set to `cfgLayout`.

Properties

ModelConfig

WINNER II model parameter configuration

WINNER II model parameter configuration, specified as a structure containing these fields:

NumTimeSamples

Number of time samples. The default value is 100.

Note If the number of samples in the input signal (N_S) does not match `NumTimeSamples`, `NumTimeSamples` is updated to match N_S.

FixedPdpUsed

Set to `'yes'` to use predefined path delays and powers for specific scenarios. The default value is `'no'`.

FixedAnglesUsed

Set to `'yes'` to use predefined angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios. The default value is `'no'`.

IntraClusterDsUsed

Set to `'yes'` to divide each of the two strongest clusters per link into three subclusters. The default value is `'yes'`.

PolarisedArrays

Set to `'yes'` to use dual polarized arrays. The default value is `'yes'`.

UseManualPropCondition

Set to `'yes'` to use the manually defined propagation conditions (LOS or NLOS) in the `LayoutConfig.PropagConditionVector` field. Set to `'no'` to draw propagation conditions from predefined LOS probabilities. The default value is `'yes'`.

UniformTimeSampling

Set to `'yes'` to sample all links at the same time instants. The default value is `'no'`.

SampleDensity

Number of time samples per half wavelength. The default value is `2e6`.

CenterFrequency

Center frequency of carrier. The default value `5.25e9` Hz.

DelaySamplingInterval

Sampling grid to which the path delays are rounded. The default value of `0` seconds indicates no rounding on path delays.

- `DelaySamplingInterval` specifies the input signal sample time.
- When performing channel filtering, the object uses `DelaySamplingInterval = 0` to obtain the original path delays. Any non-zero value of `DelaySamplingInterval` is ignored, specifically the path delays used are not rounded to be multiples of `DelaySamplingInterval` values that are non-zero.

ShadowingModelUsed

Set to `'yes'` to include shadow fading in the model. The default value is `'no'`.

PathLossModelUsed

Set to `'yes'` to include path loss in the model. The default value is `'no'`.

PathLossModel

Path loss model function name, specified as `'pathloss'`, which uses the internal `pathloss` function from the “WINNER II Channel” Add-On to model the path loss. The `PathLossModel` property is applicable only when `PathLossModelUsed` is `'yes'`. The default value is `'pathloss'`.

PathLossOption

Path loss option indicating the wall material for the NLOS path loss calculation of scenario A1, specified as one of `{'CR_light'`, `'CR_heavy'`, `'RR_light'`, `'RR_heavy'}`. The default value is `'CR_light'`. The `PathLossOption` property is applicable only when `PathLossModelUsed` is `'yes'`.

See `LayoutConfig.ScenarioVector` for the scenario number mapping.

RandomSeed

Seed for random number generators. To use the global random stream, set `RandomSeed` to empty, `[]`. The default is `[]`.

LayoutConfig

WINNER II layout parameter configuration

WINNER II layout parameter configuration, specified as a structure containing these fields:

Stations

Row vector of structures to describe antenna arrays for active stations. The row ordering specifies BS sectors first, followed by the MS. The default assigns two structures, one for BS and one for MS.

NofSect

Vector of number of sectors in each BS. The default is `1`.

Pairing

A 2-by-N_L matrix, where N_L specifies the number links to be modeled. The default is [1;2].

ScenarioVector

A 1-by-N_L vector of scenario numbers. The default is 1, which specifies scenario A1.

The scenarios numbers map as {1=A1, 2=A2, 3=B1, 4=B2, 5=B3, 6=B4, 10=C1, 11=C2, 12=C3, 13=C4, 14=D1, 15=D2a}.

For more information, see WINNER II Channel Models [1], Section 2.3.

PropagConditionVector

A 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link. The default is 1.

StreetWidth

A 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. StreetWidth is used for the path loss model of the B1 and B2 scenarios. The default is 20. See ScenarioVector for the scenario number mapping. All elements must have the same value. The StreetWidth property is applicable only when the ModelConfig.PathLossModelUsed property is 'yes'.

Dist1

A 1-by-N_L vector of distances from BS to the last LOS point. Dist1 is used for the path loss model of the B1 and B2 scenarios. The default value is NaN, which means the distance is randomly determined in path loss function. See ScenarioVector for the scenario number mapping. Dist1 is applicable only when the ModelConfig.PathLossModelUsed property is 'yes'.

For more information, see WINNER II Channel Models [1], Figure 4-3.

NumFloors

A 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The default value is 1. The NumFloors property is used for the path loss model of the A2 and B4 scenarios only. See ScenarioVector for the scenario number mapping. The

NumFloors property is applicable only when ModelConfig.PathLossModelUsed is 'yes'.

NumPenetratedFloors

A 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The default value is 0. The NumPenetratedFloors is used for the NLOS path loss model of the A1 scenario. See ScenarioVector for the scenario number mapping. The NumPenetratedFloors property is applicable only when PathLossModelUsed is 'yes'.

For more information, see WINNER II Channel Models [1], Table 4-4.

NormalizeChannelOutputs

Normalize channel outputs, specified as true or false. Set this property to true to normalize the channel outputs by the number of receive antennas at the mobile station (MS) for each link. The default value is true.

For more information, see “Channel Power” on page 3-1546.

Methods

info	Display information about WINNER2Channel object
reset	Reset states of WINNER2Channel object
step	Filter input signal through WINNER II fading channel

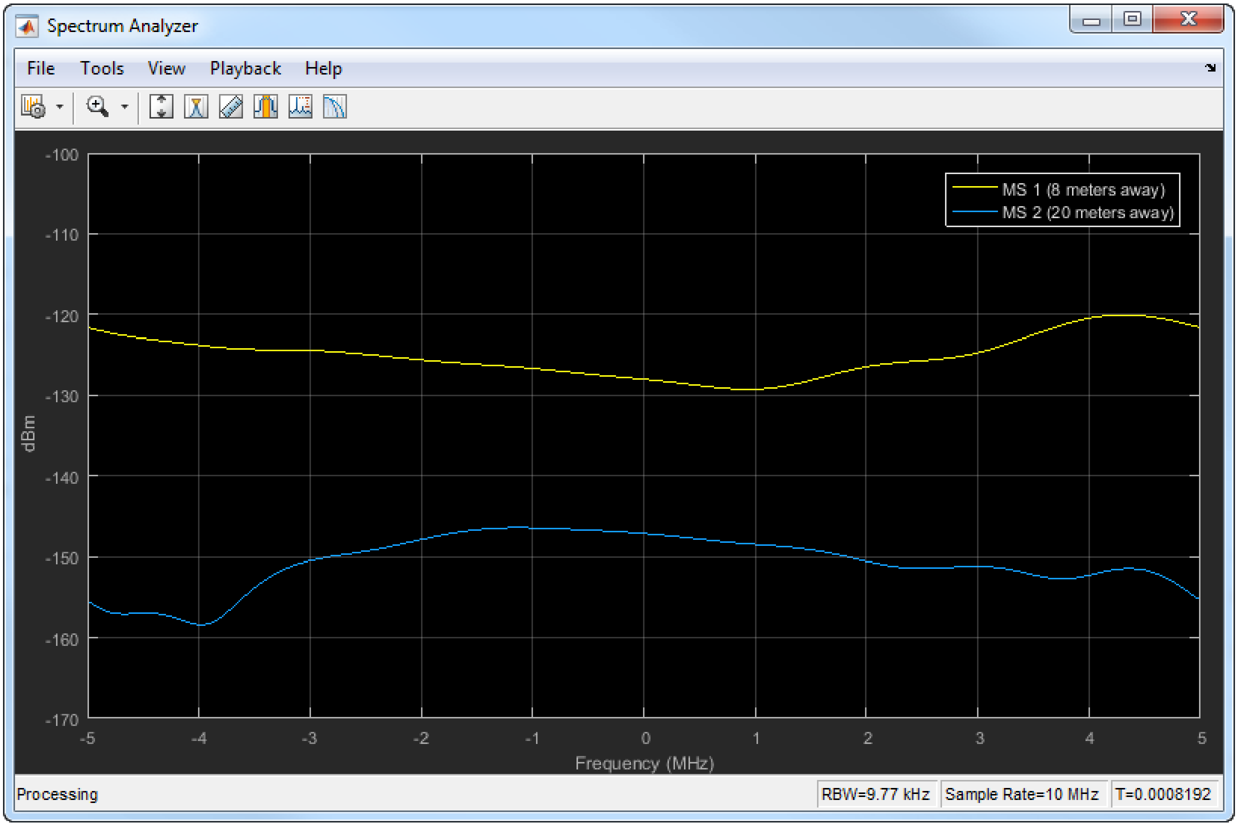
Common to All System Objects	
release	Allow System object property value changes

Examples

WINNER II Channel with Two Mobile Stations

```
%% WINNER II Channel with Two Mobile Stations
% Simulate a system that has two MS connected to one BS. One MS is 8 meters
% away from the BS; the other is 20 meters away from the BS. Impulse
% signals are sent through the two links. The spectrum of the received
% signals at MS shows frequency selectivity. It also shows the MS that is
% closer the BS has a larger average received power than the other MS.
%%
% Specify random number generator seed for repeatability.
rng(100);
%%
% Initial frame length and sample rate.
frmLen   = 1024;
%%
% Configure layout parameters.
BSAA  = winner2.AntennaArray('UCA', 8, 0.02);  % UCA-8 antenna array for BS
MSAA1 = winner2.AntennaArray('ULA', 2, 0.01);  % ULA-2 antenna array for MS
MSAA2 = winner2.AntennaArray('ULA', 4, 0.005); % ULA-4 antenna array for MS
MSIdx = [2 3]; BSIdx = {1}; NL = 2; maxRange = 100; rndSeed = 101;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx,NL, ...
   [BSAA,MSAA1,MSAA2],maxRange,rndSeed);
%%
% Adjust BS and MS positions.
cfgLayout.Stations(1).Pos(1:2) = [10, 10];
cfgLayout.Stations(2).Pos(1:2) = [18, 10];  % 8 meters away from BS
cfgLayout.Stations(3).Pos(1:2) = [22, 26];  % 20 meters away from BS
%%
% NLOS for both links
cfgLayout.Pairing = [1 1; 2 3];
cfgLayout.PropagConditionVector = [0 0];
%%
% Configure model parameters
cfgModel = winner2.wimparset;
cfgModel.NumTimeSamples     = frmLen; % Frame length
cfgModel.IntraClusterDsUsed = 'no';   % No cluster splitting
cfgModel.SampleDensity      = 2e5;    % For lower sample rate
cfgModel.PathLossModelUsed  = 'yes';  % Turn on path loss
cfgModel.ShadowingModelUsed = 'yes';  % Turn on shadowing
%%
% Create a WINNER II channel System object.
wimChan = comm.WINNER2Channel(cfgModel, cfgLayout);
%%
% Call the info method of the object to get some system information
chanInfo = info(wimChan)
numTx    = chanInfo.NumBSElements(1);
Rs       = chanInfo.SampleRate(1);
%%
% Create a Spectrum Analyzer System object.
```

```
SA = dsp.SpectrumAnalyzer('SampleRate', Rs, ...
    'YLimits', [-170, -100], 'ShowLegend', true, ...
    'ChannelNames', {'MS 1 (8 meters away)','MS 2 (20 meters away)'});
%%
% Pass impulse signals through the two links and show spectra of the
% received signals at the two MS.
for i = 1:10
    x = [ones(1,numTx); zeros(frmLen-1, numTx)];
    y = wimChan(x);
    SA([y{1}(:,1), y{2}(:,1)]);
end
```



More About

WINNER II Sampling Rate

The signal sample rate (R_S) for generating channel coefficients and performing channel filtering is calculated per link using the mobile station speed (V_{MS}), half wavelength distance, and sample density. The sample rate for each link is available as a field in the info method return.

$R_S = V_{MS} / (C / F_{center} / 2 / N_{SD})$,

- For the MS speed,V_{MS},
  - When `ModelConfig.UniformTimeSampling` is set to `'no'`, V_{MS} is the speed of the MS for the corresponding link, derived from the `LayoutConfig.Stations(i).Velocity` field.
  - When `ModelConfig.UniformTimeSampling` is set to `'yes'`, V_{MS} is the maximum speed of the MS for all links.
- C is the speed of light (`2.99792458e8` m/s).
- F_{center} is `ModelConfig.CenterFrequency`.
- N_{SD} is `ModelConfig.SampleDensity`.

Channel Power

These conditions apply to the channel power of the `comm.WINNER2Channel` object:

- When path loss and shadowing are off, path gains are normalized. Specifically, path gains are normalized when the ModelConfig.`ShadowingModelUsed` and ModelConfig.`PathLossModelUsed` parameters are set to `'no'`.
- When the NormalizeChannelOutputs property is `true`, the average gain of the channel is 0 dB.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4–027756 WINNER II, September 2007.

See Also

System Objects
`comm.AWGNChannel` | `comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Functions
`winner2.AntennaArray` | `winner2.layoutparset` | `winner2.wim` | `winner2.wimparset`

Introduced in R2016b

info

System object: comm.WINNER2Channel
Package: comm

Display information about `WINNER2Channel` object

Syntax

```
s = info(obj)
```

Description

`s = info(obj)` returns a structure containing information about the `Winner2Channel` System object characteristics. The information structure contains:

- `Numlinks` - Number of links in the system
- `NumBSElements` - Number of transmit antennas at the BS for each link
- `NumMSElements` - Number of receive antennas at the MS for each link
- `NumPaths` - Number of delay paths for each link
- `SampleRate` - Sample rate for each link
- `ChannelFilterDelay` - Channel filter delay per link, measured in samples
- `NumSamplesProcessed` - Number of samples the channel has processed since the last reset

Introduced in R2016b

reset

System object: comm.WINNER2Channel
Package: comm

Reset states of `WINNER2Channel` object

Syntax

```
reset(obj)
```

Description

`reset(obj)` resets the states of the `Winner2Channel` System object.

If the `ModelConfig.RandomSeed` property of `obj` is empty, the `reset` method resets the filters only. Otherwise, the `reset` method resets the filters and also reinitializes the random number stream to the value of the `ModelConfig.RandomSeed` property.

Introduced in R2016b

step

System object: comm.WINNER2Channel
Package: comm

Filter input signal through WINNER II fading channel

Syntax

```
y = step(obj,x)
[y,pathGains] = step(obj,x)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(obj,x)` filters input signal `x` through a WINNER II fading channel and returns the result in `y`. Both `x` and `y` are N_L-by-1 cell arrays, where N_L represents the number of links, as determined by the `LayoutConfig` property of `obj`. The `i`th element of `x` must be an N_S-by-N_T`(i)` matrix of doubles.

- N_S represents the number of samples to be generated and must be the same for all elements of `x`.
- N_T`(i)` is the number of transmit antennas at the base station (BS) for the `i`th link, determined by the `LayoutConfig` property of `obj`.

If the channel has only one link or if all links have the same number of transmit antennas, `x` can also be an N_S-by-N_T matrix of doubles. In this case, the same input signal is filtered through all the links. The `i`th element of `y` is an N_S-by-N_R`(i)` matrix of doubles. N_R`(i)` is the number of receive antennas at the mobile station (MS) for the `i`th link, as determined by the `LayoutConfig` property of `obj`.

`[y,pathGains] = step(obj,x)` also returns the channel coefficients of the underlying WINNER II fading process. `pathGains` is an N_L-by-1 cell array. The `i`th element of pathGains is an N_R(i)-by-N_T(i)-by-N_P(i)-by-N_S array of complex doubles. N_P(i) is the number of paths for the ith link, as determined by the LayoutConfig property of obj.

N_R, N_T, and N_P are link specific. N_S is the same for all the links.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) 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.

Introduced in R2016b