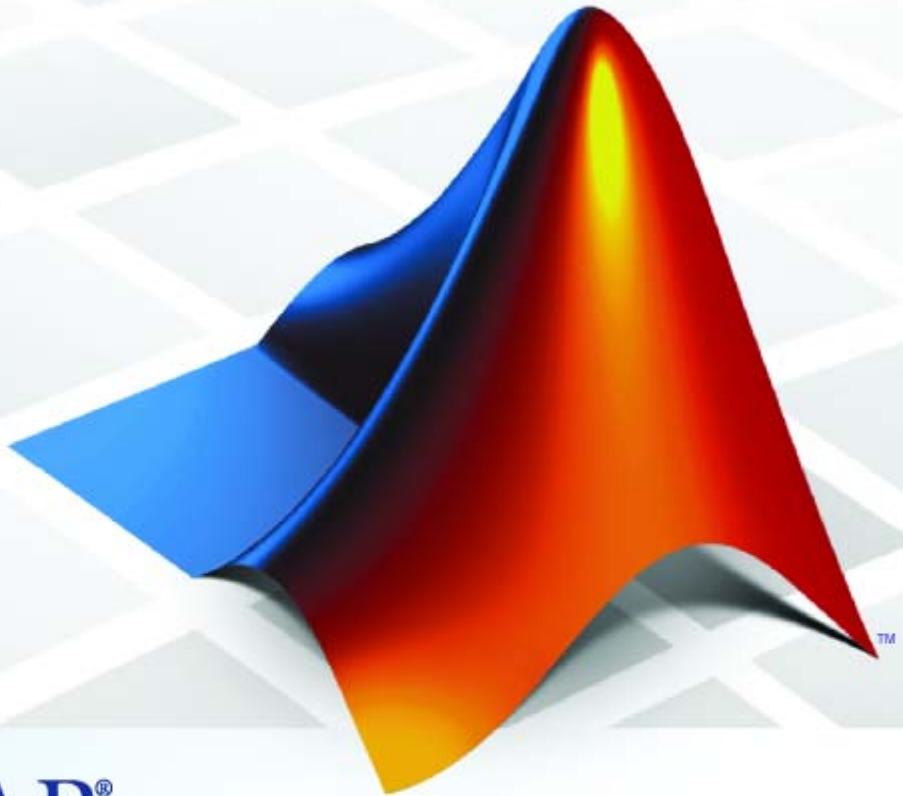


Communications Toolbox™ 4

Reference



MATLAB®

How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Communications Toolbox™ Reference

© COPYRIGHT 1996–2009 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

April 1996	First printing	Version 1.0
May 1997	Second printing	Revised for Version 1.1 (MATLAB 5.0)
September 2000	Third printing	Revised for Version 2.0 (Release 12)
May 2001	Online only	Revised for Version 2.0.1 (Release 12.1)
July 2002	Fourth printing	Revised for Version 2.1 (Release 13)
June 2004	Fifth printing	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.1 (Release 14SP2)
September 2005	Online only	Revised for Version 3.2 (Release 14SP3)
October 2005	Reprint	Version 3.0 (Notice updated)
March 2006	Online only	Revised for Version 3.3 (Release 2006a)
September 2006	Sixth printing	Revised for Version 3.4 (Release 2006b)
March 2007	Online only	Revised for Version 3.5 (Release 2007a)
September 2007	Online only	Revised for Version 4.0 (Release 2007b)
March 2008	Online only	Revised for Version 4.1 (Release 2008a)
October 2008	Online only	Revised for Version 4.2 (Release 2008b)
March 2009	Online only	Revised for Version 4.3 (Release 2009a)

Function Reference

1

Signal Sources	1-2
Performance Evaluation	1-2
Source Coding	1-3
Error-Control Coding	1-4
Interleaving/Deinterleaving	1-5
Analog Modulation/Demodulation	1-6
Digital Modulation/Demodulation	1-7
Pulse Shaping	1-9
Filters	1-9
Lower-Level Filters	1-9
Channels	1-9
Equalizers	1-11
Galois Field	1-11
MATLAB Functions and Operators	1-12
Galois Fields of Odd Characteristic	1-13
Utilities	1-14
MATLAB Utilities	1-16

GUI 1-16

Functions — Alphabetical List

2

Index

Function Reference

Signal Sources (p. 1-2)	Sources of random signals
Performance Evaluation (p. 1-2)	Analyzing and visualizing performance of a communication system
Source Coding (p. 1-3)	Quantization, companders, and other kinds of source coding
Error-Control Coding (p. 1-4)	Block and convolutional coding
Interleaving/Deinterleaving (p. 1-5)	Block and convolutional interleaving
Analog Modulation/Demodulation (p. 1-6)	Passband amplitude, frequency, and phase modulation
Digital Modulation/Demodulation (p. 1-7)	Baseband digital modulation
Pulse Shaping (p. 1-9)	Oversampling and shaping a signal
Filters (p. 1-9)	Raised cosine and Hilbert filters
Channels (p. 1-9)	Channel models for real, complex, and binary signals
Equalizers (p. 1-11)	Adaptive and MLSE equalizers
Galois Field (p. 1-11)	Manipulating elements of finite fields of even order
Galois Fields of Odd Characteristic (p. 1-13)	Manipulating elements of finite fields of odd order
Utilities (p. 1-14)	Miscellaneous relevant functions
GUI (p. 1-16)	Bit error rate analysis tool

Signal Sources

<code>commsrc.pattern</code>	Construct pattern generator object
<code>randerr</code>	Generate bit error patterns
<code>randint</code>	Generate matrix of uniformly distributed random integers
<code>randsrc</code>	Generate random matrix using prescribed alphabet
<code>wgn</code>	Generate white Gaussian noise

Performance Evaluation

<code>berawgn</code>	Bit error rate (BER) for uncoded AWGN channels
<code>bercoding</code>	Bit error rate (BER) for coded AWGN channels
<code>berconfint</code>	Bit error rate (BER) and confidence interval of Monte Carlo simulation
<code>berfading</code>	Bit error rate (BER) for Rayleigh and Rician fading channels
<code>berfit</code>	Fit curve to nonsmooth empirical bit error rate (BER) data
<code>bersync</code>	Bit error rate (BER) for imperfect synchronization
<code>biterr</code>	Compute number of bit errors and bit error rate (BER)
<code>commmeasure.EVM</code>	Create EVM measurement object
<code>commmeasure.MER</code>	Create MER measurement object
<code>commscope</code>	Package of communications scope classes

<code>commscope.eyediagram</code>	Eye diagram analysis
<code>commscope.ScatterPlot</code>	Create Scatter Plot scope
<code>distspec</code>	Compute distance spectrum of convolutional code
<code>eyediagram</code>	Generate eye diagram
<code>EyeScope</code>	Launch eye diagram scope for eye diagram object H
<code>noisebw</code>	Equivalent noise bandwidth of filter
<code>scatterplot</code>	Generate scatter plot
<code>semianalytic</code>	Calculate bit error rate (BER) using semianalytic technique
<code>symerr</code>	Compute number of symbol errors and symbol error rate

Source Coding

<code>arithdeco</code>	Decode binary code using arithmetic decoding
<code>arithenco</code>	Encode sequence of symbols using arithmetic coding
<code>compand</code>	Source code mu-law or A-law compressor or expander
<code>dpcmdeco</code>	Decode using differential pulse code modulation
<code>dpcmenco</code>	Encode using differential pulse code modulation
<code>dpcmopt</code>	Optimize differential pulse code modulation parameters
<code>huffmandeco</code>	Huffman decoder

<code>huffmandict</code>	Generate Huffman code dictionary for source with known probability model
<code>huffmanenco</code>	Huffman encoder
<code>lloyds</code>	Optimize quantization parameters using Lloyd algorithm
<code>quantiz</code>	Produce quantization index and quantized output value

Error-Control Coding

<code>bchdec</code>	BCH decoder
<code>bchenc</code>	BCH encoder
<code>bchgenpoly</code>	Generator polynomial of BCH code
<code>bchnumerr</code>	Number of correctable errors for BCH code
<code>convenc</code>	Convolutionally encode binary data
<code>cyclgen</code>	Produce parity-check and generator matrices for cyclic code
<code>cyclpoly</code>	Produce generator polynomials for cyclic code
<code>decode</code>	Block decoder
<code>dvbs2ldpc</code>	Low-density parity-check codes from DVB-S.2 standard
<code>encode</code>	Block encoder
<code>fec.bchdec</code>	Construct BCH decoder object
<code>fec.bchenc</code>	Construct BCH encoder object
<code>fec.ldpcdec</code>	Construct LDPC decoder object
<code>fec.ldpcenc</code>	Construct LDPC encoder object

<code>fec.rsdec</code>	Construct Reed-Solomon decoder object
<code>fec.rsenc</code>	Construct Reed-Solomon encoder object
<code>gen2par</code>	Convert between parity-check and generator matrices
<code>gfweight</code>	Calculate minimum distance of linear block code
<code>hammgen</code>	Produce parity-check and generator matrices for Hamming code
<code>rsdec</code>	Reed-Solomon decoder
<code>rsdecof</code>	Decode ASCII file encoded using Reed-Solomon code
<code>rsenc</code>	Reed-Solomon encoder
<code>rsencof</code>	Encode ASCII file using Reed-Solomon code
<code>rsgenpoly</code>	Generator polynomial of Reed-Solomon code
<code>syndtable</code>	Produce syndrome decoding table
<code>vitdec</code>	Convolutionally decode binary data using Viterbi algorithm

Interleaving/Deinterleaving

<code>algdeintrlv</code>	Restore ordering of symbols using algebraically derived permutation table
<code>algintrlv</code>	Reorder symbols using algebraically derived permutation table
<code>convdeintrlv</code>	Restore ordering of symbols using shift registers

<code>convintrlv</code>	Permute symbols using shift registers
<code>deintrlv</code>	Restore ordering of symbols
<code>heldeintrlv</code>	Restore ordering of symbols permuted using <code>helintrlv</code>
<code>helintrlv</code>	Permute symbols using helical array
<code>helscandeintrlv</code>	Restore ordering of symbols in helical pattern
<code>helscanintrlv</code>	Reorder symbols in helical pattern
<code>intrlv</code>	Reorder sequence of symbols
<code>matdeintrlv</code>	Restore ordering of symbols by filling matrix by columns and emptying it by rows
<code>matintrlv</code>	Reorder symbols by filling matrix by rows and emptying it by columns
<code>muxdeintrlv</code>	Restore ordering of symbols using specified shift registers
<code>muxintrlv</code>	Permute symbols using shift registers with specified delays
<code>randdeintrlv</code>	Restore ordering of symbols using random permutation
<code>randintrlv</code>	Reorder symbols using random permutation

Analog Modulation/Demodulation

<code>amdemod</code>	Amplitude demodulation
<code>ammod</code>	Amplitude modulation
<code>fmdemod</code>	Frequency demodulation
<code>fmmmod</code>	Frequency modulation

<code>pmdemod</code>	Phase demodulation
<code>pmmmod</code>	Phase modulation
<code>ssbdemod</code>	Single sideband amplitude demodulation
<code>ssbmod</code>	Single sideband amplitude modulation

Digital Modulation/Demodulation

<code>dpskdemod</code>	Differential phase shift keying demodulation
<code>dpskmod</code>	Differential phase shift keying modulation
<code>fskdemod</code>	Frequency shift keying demodulation
<code>fskmod</code>	Frequency shift keying modulation
<code>genqamdemod</code>	General quadrature amplitude demodulation
<code>genqammod</code>	General quadrature amplitude modulation
<code>modem</code>	Package of modem classes
<code>modem.dpskdemod</code>	Construct DPSK demodulator object
<code>modem.dpskmod</code>	Construct DPSK modulator object
<code>modem.genqamdemod</code>	Construct General QAM demodulator object
<code>modem.genqammod</code>	Construct General QAM modulator object
<code>modem.mskdemod</code>	Construct MSK demodulator object
<code>modem.mskmod</code>	Construct MSK modulator object

<code>modem.oqpskdemod</code>	Construct OQPSK demodulator object
<code>modem.oqpskmod</code>	Construct OQPSK modulator object
<code>modem.pamdemod</code>	Construct PAM demodulator object
<code>modem.pammod</code>	Construct PAM modulator object
<code>modem.pskdemod</code>	Construct PSK demodulator object
<code>modem.pskmod</code>	Construct PSK modulator object
<code>modem.qamdemod</code>	Construct QAM demodulator object
<code>modem.qammod</code>	Construct QAM modulator object
<code>modnorm</code>	Scaling factor for normalizing modulation output
<code>mskdemod</code>	Minimum shift keying demodulation
<code>mskmod</code>	Minimum shift keying modulation
<code>oqpskdemod</code>	Offset quadrature phase shift keying demodulation
<code>oqpskmod</code>	Offset quadrature phase shift keying modulation
<code>pamdemod</code>	Pulse amplitude demodulation
<code>pammod</code>	Pulse amplitude modulation
<code>pskdemod</code>	Phase shift keying demodulation
<code>pskmod</code>	Phase shift keying modulation
<code>qamdemod</code>	Quadrature amplitude demodulation
<code>qammod</code>	Quadrature amplitude modulation

Pulse Shaping

<code>intdump</code>	Integrate and dump
<code>rcosflt</code>	Filter input signal using raised cosine filter
<code>rectpulse</code>	Rectangular pulse shaping

Filters

<code>hank2sys</code>	Convert Hankel matrix to linear system model
<code>hilbiir</code>	Design Hilbert transform IIR filter
<code>rcosine</code>	Design raised cosine filter

Lower-Level Filters

<code>rcosfir</code>	Design raised cosine finite impulse response (FIR) filter
<code>rcosiir</code>	Design raised cosine infinite impulse response (IIR) filter

Channels

<code>awgn</code>	Add white Gaussian noise to signal
<code>bsc</code>	Model binary symmetric channel
<code>doppler</code>	Package of Doppler classes
<code>doppler.ajakes</code>	Construct asymmetrical Doppler spectrum object

<code>doppler.bell</code>	Construct bell-shaped Doppler spectrum object
<code>doppler.bigaussian</code>	Construct bi-Gaussian Doppler spectrum object
<code>doppler.flat</code>	Construct flat Doppler spectrum object
<code>doppler.gaussian</code>	Construct Gaussian Doppler spectrum object
<code>doppler.jakes</code>	Construct Jakes Doppler spectrum object
<code>doppler.rjakes</code>	Construct restricted Jakes Doppler spectrum object
<code>doppler.rounded</code>	Construct rounded Doppler spectrum object
<code>filter (channel)</code>	Filter signal with channel object
<code>mimochan</code>	Create MIMO fading channel object
<code>plot (channel)</code>	Plot channel characteristics with channel visualization tool
<code>rayleighchan</code>	Construct Rayleigh fading channel object
<code>reset (channel)</code>	Reset channel object
<code>ricianchan</code>	Construct Rician fading channel object
<code>stdchan</code>	Construct channel object from set of standardized channel models

Equalizers

<code>cma</code>	Construct constant modulus algorithm (CMA) object
<code>dfc</code>	Construct decision-feedback equalizer object
<code>equalize</code>	Equalize signal using equalizer object
<code>lineareq</code>	Construct linear equalizer object
<code>lms</code>	Construct least mean square (LMS) adaptive algorithm object
<code>mlseqq</code>	Equalize linearly modulated signal using Viterbi algorithm
<code>normlms</code>	Construct normalized least mean square (LMS) adaptive algorithm object
<code>reset (equalizer)</code>	Reset equalizer object
<code>rls</code>	Construct recursive least squares (RLS) adaptive algorithm object
<code>signlms</code>	Construct signed least mean square (LMS) adaptive algorithm object
<code>varlms</code>	Construct variable-step-size least mean square (LMS) adaptive algorithm object

Galois Field

<code>convmtx</code>	Convolution matrix of Galois field vector
<code>cosets</code>	Produce cyclotomic cosets for Galois field

dftmtx	Discrete Fourier transform matrix in Galois field
fft	Discrete Fourier transform
filter (gf)	1-D digital filter over Galois field
gf	Create Galois field array
gftable	Generate file to accelerate Galois field computations
ifft	Inverse discrete Fourier transform
isprimitive	True for primitive polynomial for Galois field
log	Logarithm in Galois field
minpol	Find minimal polynomial of Galois field element
mldivide	Matrix left division \ of Galois arrays
primpoly	Find primitive polynomials for Galois field

MATLAB Functions and Operators

+ -	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
' .'	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

deconv	Deconvolution and polynomial division
det	Determinant of square Galois matrix
diag	Diagonal Galois matrices and diagonals of a Galois matrix
inv	Inverse of Galois matrix
isempty	True for empty Galois arrays
length	Length of Galois vector
lu	Lower-upper triangular factorization of Galois array
polyval	Evaluate polynomial in Galois field
rank	Rank of a Galois array
reshape	Reshape Galois array
roots	Find polynomial roots across a Galois field
size	Size of Galois array
tril	Extract lower triangular part of Galois array
triu	Extract upper triangular part of Galois array

Galois Fields of Odd Characteristic

gfadd	Add polynomials over Galois field
gfconv	Multiply polynomials over Galois field
gfcosets	Produce cyclotomic cosets for Galois field
gfdeconv	Divide polynomials over Galois field
gfdiv	Divide elements of Galois field
gffilter	Filter data using polynomials over prime Galois field

<code>gflinseq</code>	Find particular solution of $Ax = b$ over prime Galois field
<code>gfminpol</code>	Find minimal polynomial of Galois field element
<code>gfmul</code>	Multiply elements of Galois field
<code>gfpretty</code>	Polynomial in traditional format
<code>gfprimck</code>	Check whether polynomial over Galois field is primitive
<code>gfprimdf</code>	Provide default primitive polynomials for Galois field
<code>gfprimfd</code>	Find primitive polynomials for Galois field
<code>gfrank</code>	Compute rank of matrix over Galois field
<code>gfrepconv</code>	Convert one binary polynomial representation to another
<code>gfroots</code>	Find roots of polynomial over prime Galois field
<code>gfsub</code>	Subtract polynomials over Galois field
<code>gftrunc</code>	Minimize length of polynomial representation
<code>gftuple</code>	Simplify or convert Galois field element formatting

Utilities

<code>alignsignals</code>	Align two signals by delaying earliest signal
<code>bi2de</code>	Convert binary vectors to decimal numbers

<code>bin2gray</code>	Convert positive integers into corresponding Gray-encoded integers
<code>commsrc.pn</code>	Create PN sequence generator package
<code>de2bi</code>	Convert decimal numbers to binary vectors
<code>finddelay</code>	Estimate delay(s) between signals
<code>gray2bin</code>	Convert Gray-encoded positive integers to corresponding Gray-decoded integers
<code>iscatastrophic</code>	True for trellis corresponding to catastrophic convolutional code
<code>istrellis</code>	True for valid trellis structure
<code>marcumq</code>	Generalized Marcum Q function
<code>mask2shift</code>	Convert mask vector to shift for shift register configuration
<code>oct2dec</code>	Convert octal to decimal numbers
<code>poly2trellis</code>	Convert convolutional code polynomials to trellis description
<code>qfunc</code>	Q function
<code>qfuncinv</code>	Inverse Q function
<code>seqgen</code>	Sequence generator package
<code>seqgen.pn</code>	Construct default PN sequence generator object
<code>shift2mask</code>	Convert shift to mask vector for shift register configuration
<code>vec2mat</code>	Convert vector into matrix

MATLAB Utilities

erf	Error function
erfc	Complementary error function

GUI

bertool	Open bit error rate analysis GUI (BERTool)
---------	--

Functions — Alphabetical List

algdeintrlv

Purpose Restore ordering of symbols using algebraically derived permutation table

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

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

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

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

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

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


See Also algintrlv, “Interleaving”

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.

algintrlv

Purpose

Reorder symbols using algebraically derived permutation table

Syntax

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

Description

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

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

Examples

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

- 1 Define `num` and the data to interleave.

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

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

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

```

3      1
4      1
5      1
9      1

```

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

```

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

      8
      7
      6
      2

```

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

```

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

```

Algorithm

- A Takeshita-Costello interleaver uses a length-`num` cycle vector whose n th element is $\text{mod}(k \cdot (n-1) \cdot n/2, \text{num})$ 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 $\text{mod}(A^K, \text{num}+1) - 1$.

See Also

algdeintrlv, “Interleaving”

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.

Purpose

Align two signals by delaying earliest signal

Syntax

```
[Xa Ya] = alignsignals(X,Y)
[Xa Ya] = alignsignals(X,Y,MAXLAG)
[Xa Ya] = alignsignals(X,Y,MAXLAG,'truncate')
[Xa Ya D] = alignsignals(...)
```

Description

`[Xa Ya] = alignsignals(X,Y)`, where X and Y are row or column vectors of length LX and LY , respectively, aligns the two vectors by estimating the delay D between the two. If Y is delayed with respect to X , D is positive, and X is delayed by D samples. If Y is advanced with respect to X , D is negative, and Y is delayed by $-D$ samples. The aligned signals Xa and Ya are returned. Delays in X and Y can be introduced by pre-pending zeros.

`[Xa Ya] = alignsignals(X,Y,MAXLAG)` uses $MAXLAG$ as the maximum window size used to find the estimated delay D between X and Y . $MAXLAG$ is an integer-valued scalar. By default, $MAXLAG$ is equal to $MAX(LX, LY)-1$. If $MAXLAG$ is input as `[]`, it is replaced by the default value. If $MAXLAG$ is negative, it is replaced by its absolute value. If $MAXLAG$ is not integer-valued, or is complex, `Inf`, or `NaN`, then `alignsignals` returns an error.

`[Xa Ya] = alignsignals(X,Y,MAXLAG,'truncate')` keeps the lengths of Xa and Ya the same as those of X and Y , respectively. If D is positive, D zeros are pre-pended to X , and the last D samples of X are truncated. If D is negative, $-D$ zeros are pre-pended to Y , and the last $-D$ samples of Y are truncated. Note: If $D \geq LX$, Xa will consist of LX zeros, and all samples of X are lost. Similarly, if $-D \geq LY$, Ya will consist of LY zeros, and all samples of Y are lost. To avoid assigning a specific value to $MAXLAG$ when using the 'truncate' option, set $MAXLAG$ to `[]`.

`[Xa Ya D] = alignsignals(...)` returns the estimated delay D .

Theory and Algorithm

The theory on delay estimation can be found in the specification of the `finddelay` function (see “Theory and Algorithm” on page 2-265).

The `alignsignals` function simply uses the estimated delay to delay the earliest signal such that the two signals have the same starting point.

As specified for the `finddelay` function, the pair of signals need not be exact delayed copies of each other. However, the signals can be successfully aligned only if there is sufficient correlation between them.

Examples

The following illustrates how X is aligned when Y is delayed with respect to X by two samples.

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

The resulting values are:

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

The following is a case where Y is advanced with respect to X by three samples.

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

The resulting values are:

```
Xa = [0 0 0 1 2 3 0 0]'  
Ya = [0 0 0 1 2 3 0]'
```

The following illustrates a signal Y that is aligned with respect to X but is noisy.

```
X = [0 0 1 2 3 0];  
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];  
[Xa Ya D] = alignsignals(X, Y)
```

The resulting values are:

```
Xa = [0 0 1 2 3 0]  
Ya = [0.02 0.12 1.08 2.21 2.95 -0.09];  
D = 0
```

The following shows that when Y is a periodic repetition of X, the smallest possible delay is returned.

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

The resulting values are:

```
Xa = [0 1 2 3];  
Ya = [0 1 2 3 0 0 0 0 1 2 3 0 0];  
D = -1
```

Here is an example of alignsignals using the 'truncate' option.

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

The resulting values are:

```
Xa = [0 0 1];  
Ya = [0 0 1 2 3];  
D = 2
```

In the case where using the 'truncate' option ends up truncating all the original data of X, a warning will be issued. The following example makes MATLAB issue such a warning.

alignsignals

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

See Also [finddelay](#)

Purpose Amplitude demodulation

Syntax

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

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

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

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

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

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

Examples The code below illustrates the use of a nondefault filter.

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

amdemod

```
[num,den] = butter(10,Fc*2/Fs); % Lowpass filter  
  
y1 = ammod(s,Fc,Fs); % Modulate.  
s1 = amdemod(y1,Fc,Fs,0,0,num,den); % Demodulate.
```

See Also

ammod, ssbdemod, fmdemod, pmdemod, "Modulation"

Purpose

Amplitude modulation

Syntax

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

Description

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

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

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

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

Examples

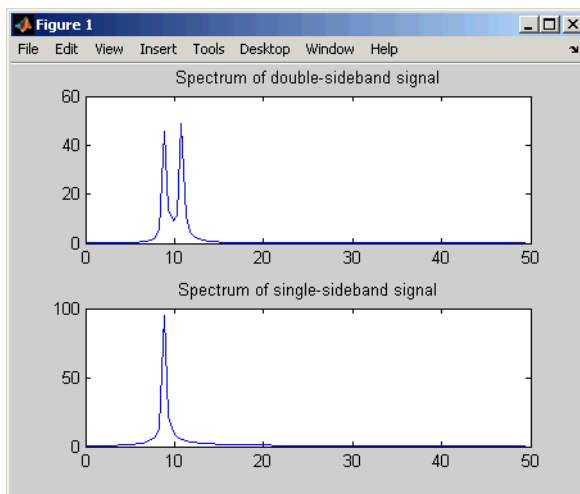
The example below compares double-sideband and single-sideband amplitude modulation.

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

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

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

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



See Also

amdemod, ssbmod, fmmmod, pmmod, “Modulation”

Purpose	Decode binary code using arithmetic decoding
Syntax	<code>dseq = arithdeco(code,counts,len)</code>
Description	<code>dseq = arithdeco(code,counts,len)</code> decodes the binary arithmetic code in the vector <code>code</code> to recover the corresponding sequence of <code>len</code> symbols. The vector <code>counts</code> 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>code</code> was produced by the <code>arithenco</code> function.
Examples	<p>This example is similar to the example on the <code>arithenco</code> reference page, except that it uses <code>arithdeco</code> to recover the original sequence.</p> <pre>counts = [99 1]; % A one occurs 99% of the time. len = 1000; seq = randsrc(1,len,[1 2; .99 .01]); % Random sequence code = arithenco(seq,counts); dseq = arithdeco(code,counts,length(seq)); % Decode. isequal(seq,dseq) % Check that dseq matches the original seq.</pre> <p>The output is</p> <pre>ans = 1</pre>
Algorithm	This function uses the algorithm described in [1].
See Also	<code>arithenco</code> , “Arithmetic Coding”
References	[1] Sayood, Khalid, <i>Introduction to Data Compression</i> , San Francisco, Morgan Kaufmann, 2000.

arithenco

Purpose Encode sequence of symbols using arithmetic coding

Syntax `code = arithenco(seq,counts)`

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

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

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

The output is

```
s =
    1    83
```

Algorithm This function uses the algorithm described in [1].

See Also `arithdeco`, “Arithmetic Coding”

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

Purpose

Add white Gaussian noise to signal

Syntax

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

Description

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

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

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

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

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

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

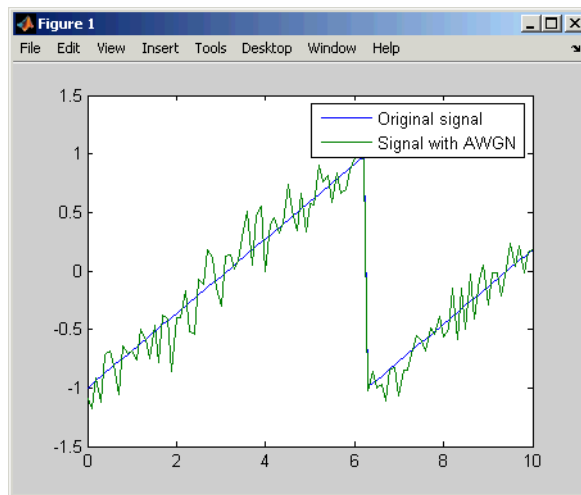
Relationship Among SNR, E_s/N_0 , and E_b/N_0

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

Examples

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

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



Several other examples that illustrate the use of `awgn` are in “Getting Started”. The following demos also use `awgn`: `basicsimdemo`, `vitsimdemo`, and `scattereydemo`.

See Also

`wgn`, `randn`, `bsc`, “AWGN Channel”

Purpose BCH decoder

Note bchdec will be removed in a future release. Use fec.bchdec instead.

Syntax

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

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

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

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

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

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

Results of Error Correction

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

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

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

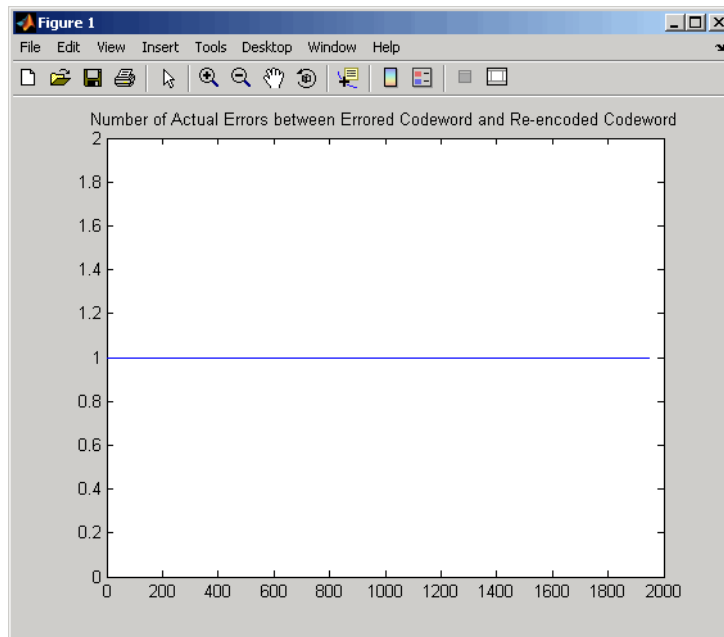
The following code illustrates this phenomenon for a single-error-correcting BCH decoder given input with two errors.

```
n = 63; k = 57;
msg = gf(randint(1, k, 2, 9973));
code = bchenc(msg, n, k);

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

```
% If bchdec thinks it corrected only one error,  
% then encode the decoded message. Check that  
% the re-encoded message differs from the errored  
% message in only one coordinate.  
if cnumerr2(cnumerrIdx) == 1  
    code2 = bchenc(decoded2, n, k);  
    nErrs(cnumerrIdx) = biterr(double(erroredCode.x),...  
        double(code2.x));  
end  
  
    cnumerrIdx = cnumerrIdx + 1;  
end  
end  
  
% Plot the computed number of errors, based on the difference  
% between the double-errored codeword and the codeword that was  
% re-encoded from the initial decoding.  
plot(nErrs)  
title('Number of Actual Errors between Errored Codeword and...  
    Re-encoded Codeword')
```

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



Examples

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

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

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

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

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

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

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

Algorithm

`bchdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” on page 2-23.

Limitations

The maximum allowable value of `n` is 65535.

See Also

`bchenc`, `bchgenpoly`, “Block Coding”

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.

bchenc

Purpose BCH encoder

Note `bchenc` will be removed in a future release. Use `fec.bchenc` instead.

Syntax

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

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

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

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

n	k	t
7	4	1

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

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

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

n	k	t
127	120	1
127	113	2
127	106	3
127	99	4
127	92	5
127	85	6
127	78	7
127	71	9
127	64	10
127	57	11
127	50	13
127	43	14
127	36	15
127	29	21
127	22	23
127	15	27
127	8	31

n	k	t
255	247	1
255	239	2
255	231	3
255	223	4

n	k	t
255	215	5
255	207	6
255	199	7
255	191	8
255	187	9
255	179	10
255	171	11
255	163	12
255	155	13
255	147	14
255	139	15
255	131	18
255	123	19
255	115	21
255	107	22
255	99	23
255	91	25
255	87	26
255	79	27
255	71	29
255	63	30
255	55	31
255	47	42
255	45	43
255	37	45

n	k	t
255	29	47
255	21	55
255	13	59
255	9	63

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

n	k	t
511	349	19
511	340	20
511	331	21
511	322	22
511	313	23
511	304	25
511	295	26
511	286	27
511	277	28
511	268	29
511	259	30
511	250	31
511	241	36
511	238	37
511	229	38
511	220	39
511	211	41
511	202	42
511	193	43
511	184	45
511	175	46
511	166	47
511	157	51
511	148	53
511	139	54

n	k	t
511	130	55
511	121	58
511	112	59
511	103	61
511	94	62
511	85	63
511	76	85
511	67	87
511	58	91
511	49	93
511	40	95
511	31	109
511	28	111
511	19	119
511	10	121

Examples

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

Limitations

The maximum allowable value of `n` is 65535.

See Also

`bchdec`, `bchgenpoly`, `bchnumerr`, “Block Coding”

Purpose Generator polynomial of BCH code

Syntax

```
genpoly = bchgenpoly(n,k)
genpoly = bchgenpoly(n,k,prim_poly)
[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 . The output `genpoly` is a Galois row vector in $GF(2)$ that represents the coefficients of the generator polynomial in order of descending powers. The narrow-sense generator polynomial is $LCM[m_1(x), m_2(x), \dots, m_{2t}(x)]$, where LCM is the least common multiple, $m_i(x)$ is the minimum polynomial corresponding to α^i , α is a root of the default primitive polynomial for the field $GF(n+1)$, and t is 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 α as a root. `prim_poly` is an integer whose binary representation indicates the coefficients of the primitive polynomial. To use the default primitive polynomial for $GF(n+1)$, set `prim_poly` to `[]`.

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

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

```
m = 4;
```

bchgenpoly

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

The output is

genpoly = GF(2) array.

Array elements =

1 0 0 1 1

t =

1

Limitations

The maximum allowable value of n is 511.

See Also

bchenc, bchdec, bchnumerr, “Block Coding”

References

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

Purpose Number of correctable errors for BCH code

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

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

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

See Also `bchenc`, `bchdec`, `bchgenpoly`

berawgn

Purpose Bit error rate (BER) for uncoded AWGN channels

Syntax

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

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

Description **For All Syntaxes**

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

Modulation Scheme	Second Input Argument
Phase shift keying (PSK)	'psk'
Offset quaternary phase shift keying (OQPSK)	'oqpsk'
Differential phase shift keying (DPSK)	'dpsk'

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 $J = 2^{\frac{k+1}{2}}$.

`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, 'qpsk', 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

An example using this function is in “Comparing Theoretical and Empirical Error Rates”.

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.

See Also

bercoding, berfading, bersync, “Theoretical Performance Results”, Analytical Expressions Used in berawgn, bercoding, berfading, and BERTool

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.

bercoding

Purpose

Bit error rate (BER) for coded AWGN channels

Syntax

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

Graphical Interface

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

Description

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

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

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

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

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

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

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

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

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

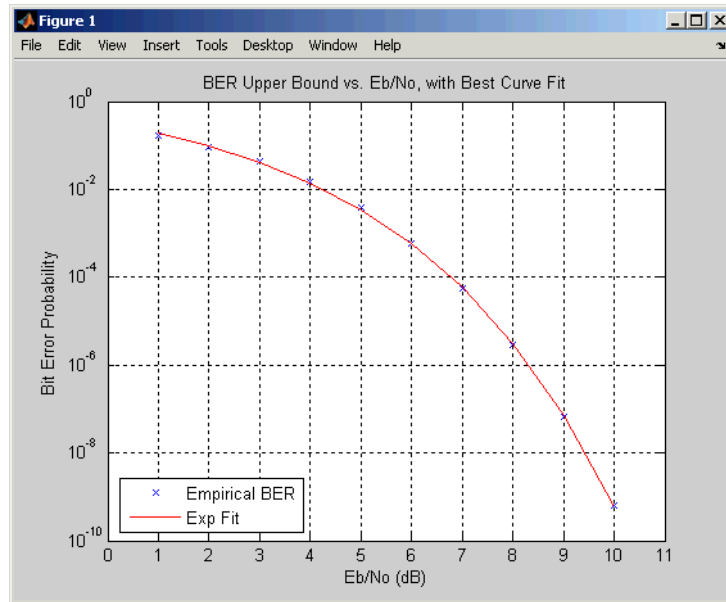
Examples

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

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

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

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



Limitations

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

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

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

See Also

berawgn, berfading, bersync, distspec, "Theoretical Performance Results" Analytical Expressions Used in berawgn, bercoding, berfading, and BERTool

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.

berconfint

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

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

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

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

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

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

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


```
ber =  
  
1.0000e-004  
  
interval =  
  
1.0e-003 *  
  
0.0814    0.1216
```

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

See Also

`binofit` (Statistics Toolbox), `mle` (Statistics Toolbox), “Performance Evaluation”

References

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

berfading

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

Syntax

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

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

Description **For All Syntaxes**

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

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

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

For cases where diversity is used, the SNR 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, 'dpsk', 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

berfading

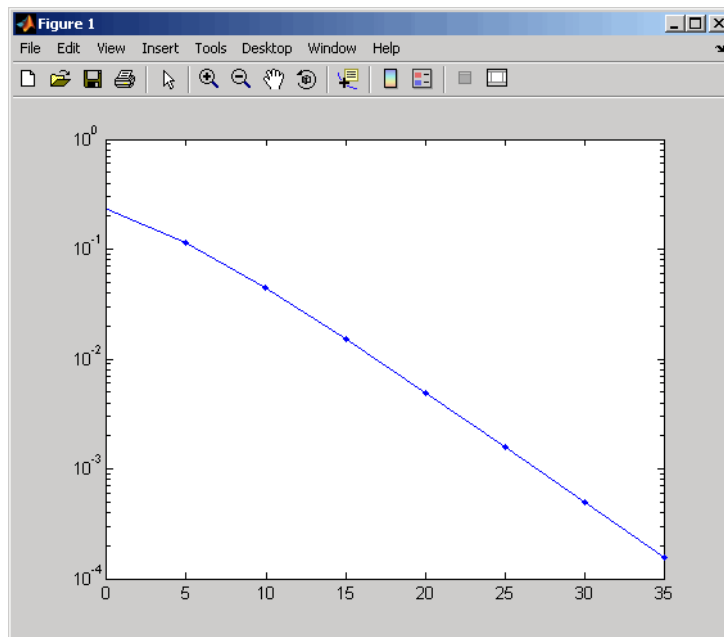
synchronization. `phaserr` is the standard deviation of the reference carrier phase error in radians.

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

Examples

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

```
EbNo = 0:5:35;  
M = 4; % Use DQPSK, so M = 4.  
divorder = 1;  
ber = berfading(EbNo,'dpsk',M,divorder);  
semilogy(EbNo,ber,'b.-');
```



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.

See Also

berawgn, bercoding, bersync, "Theoretical Performance Results" Analytical Expressions Used in berawgn, bercoding, berfading, and BERTool

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.

Purpose

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

Syntax

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

Description

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

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

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

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

berfit

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 “Algorithm” on page 2-52 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
<code>fitprops.fitType</code>	The closed-form function type used to generate the fit: 'exp', 'exp+const', 'polyRatio', or 'doubleExp+const'.
<code>fitprops.coeffs</code>	The coefficients used to generate the fit. If the function cannot find a valid fit, <code>fitprops.coeffs</code> is an empty vector.
<code>fitprops.sumSqErr</code>	The sum squared error between the log of the fitted BER points and the log of the empirical BER points.
<code>fitprops.exitState</code>	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'.
<code>fitprops.funcCount</code>	The number of function evaluations used in minimizing the sum squared error function.
<code>fitprops.iterations</code>	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 “Algorithm”

on page 2-52 below, that return a valid fit. To avoid overriding default options, use `options = []`.

Note A valid fit must be

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

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

Algorithm

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 <i>fittype</i>	Functional Expression
'exp'	$f(x) = \frac{a_1 \exp\{-(x - a_2)^{a_3}\}}{a_4}$
'exp+const'	$f(x) = \frac{a_1 \exp[-(x - a_2)^{a_3}]}{a_4} + a_5$

Value of <i>fittype</i>	Functional Expression
'polyRatio'	$f(x) = \frac{a_1 x^2 + a_2 x + a_3}{x^3 + a_4 x^2 + a_5 x + a_6}$
'doubleExp+const'	$\frac{a_1 \exp[-(x - a_2)^{a_3}]}{a_4} + \frac{a_5 \exp[-(x - a_6)^{a_7}]}{a_8} + a_9$

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

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

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

Examples

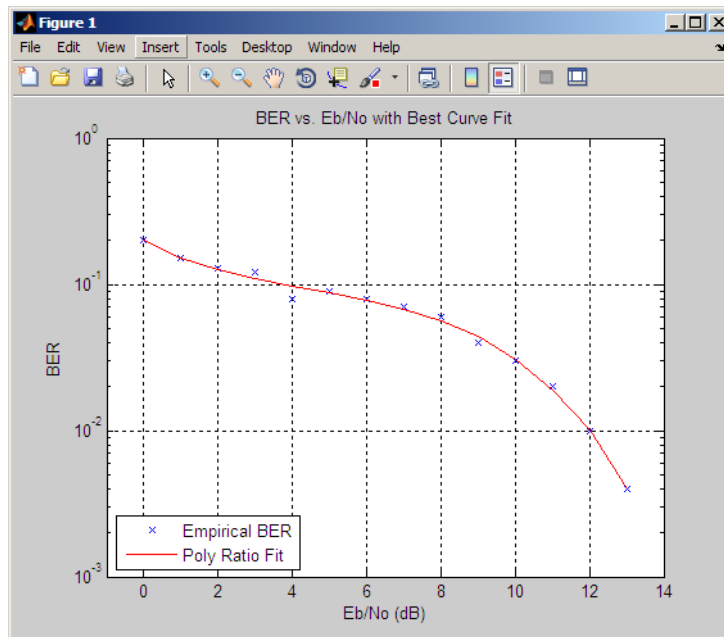
The examples below illustrate the syntax of the function, but they use hard-coded or theoretical BER data for simplicity. For an example that uses empirical BER data from a simulation, see “Example: Curve Fitting for an Error Rate Plot”.

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

```

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

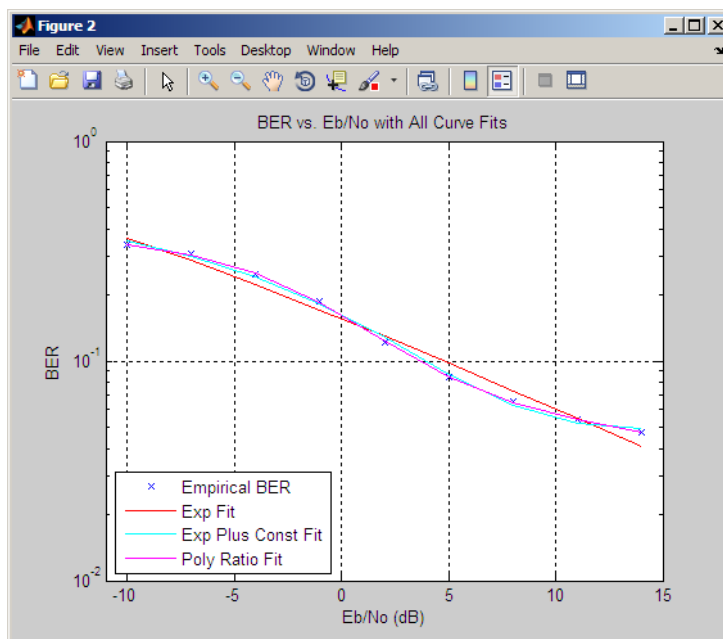
```



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

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

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



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

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

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

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

The output is as follows:

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

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

```
fitber1 =
```

```
    0.0766    0.0423    0.0205    0.0086    0.0030    0.0009    0.0002
```

```
fitprops1 =
```

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

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

```
fitber2 =
```

```
    0.0931    0.0476    0.0220    0.0090    0.0031    0.0008    0.0001
```

```
fitprops2 =
```

```
    fitType: 'polyRatio'
```

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

See Also

fminsearch, optimset, “Performance Evaluation”

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.

bersync

Purpose Bit error rate (BER) for imperfect synchronization

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

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

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

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

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

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



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

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

```
ber =
```

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

Limitations

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

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

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

Limitations Related to Extreme Values of Input Arguments

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

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

Algorithm

This function uses formulas from [3].

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

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

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

$$\frac{1}{\pi\sigma} \int_0^{\infty} \exp\left(-\frac{\phi^2}{2\sigma^2}\right) \int_{\sqrt{2R} \cos \phi}^{\infty} \exp\left(-\frac{y^2}{2}\right) dy d\phi$$

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

See Also

`berawgn`, `bercoding`, `berfading`, "Theoretical Performance Results"

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.

Purpose	Open bit error rate analysis GUI (BERTool)
Syntax	<code>bertool</code>
Description	<code>bertool</code> launches the Bit Error Rate Analysis Tool (BERTool). BERTool is a graphical user interface (GUI) that enables you to analyze communications links' BER performance via simulation-based, semianalytic, or theoretical approach. To learn about BERTool, see "BERTool: A Bit Error Rate Analysis GUI".

bi2de

Purpose Convert binary vectors to decimal numbers

Syntax

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

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

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

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

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

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

Examples

The code below generates a matrix that contains binary representations of five random numbers between 0 and 15. It then converts all five numbers to decimal integers.

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

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

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

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

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

The output is

```
d =
    551
```

See Also

de2bi

bin2gray

Purpose Convert positive integers into corresponding Gray-encoded integers

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

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

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

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

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

Example

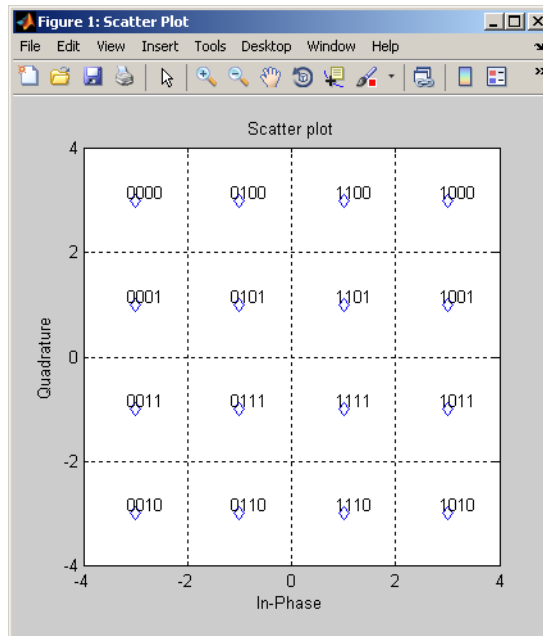
```
% To Gray encode a vector x with a 16-QAM Gray encoded
% constellation and return its map, use:
x=randint(1,100,16);
[y,map] = bin2gray(x,'qam',16);
% Obtain the symbols for 16-QAM
hMod = modem.qammod('M', 16);
symbols = hMod.Constellation;
% Plot the constellation
scatterplot(symbols);
set(get(gca,'Children'),'Marker','d','MarkerFaceColor',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

```

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

```

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



See Also

gray2bin

biterr

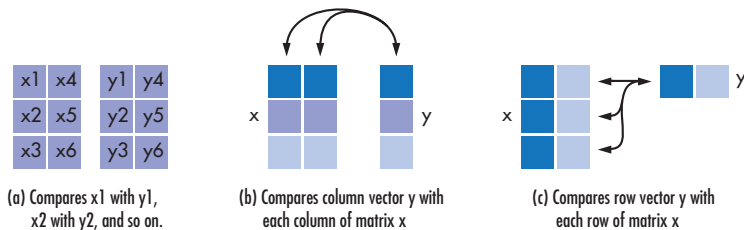
Purpose Compute number of bit errors and bit error rate (BER)

Syntax

```
[number,ratio] = biterr(x,y)
[number,ratio] = biterr(x,y,k)
[number,ratio] = biterr(x,y,k,flag)
[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 m th entry indicates the number of bits that differ when comparing the vector with the m th 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,flag)` is similar to the previous syntaxes, except that `flag` can override the defaults that govern which elements `biterr` compares and how `biterr` computes the outputs. The possible values of `flag` 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	flag	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'	m th row of x vs. m th row of y	Column vector whose entries count bit errors in each row	k times number of entries of y
	'column-wise'	m th column of x vs. m th column of y	Row vector whose entries count bit errors in each column	k times number of entries of y

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

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

`[number,ratio,individual] = biterr(...)` returns a matrix `individual` whose dimensions are those of the larger of x and y . Each

entry of `individual` corresponds to a comparison between a pair of elements of `x` and `y`, and specifies the number of bits by which the elements in the pair differ.

Examples

Example 1

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

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

The output is

number =

2	0	0	3	1
---	---	---	---	---

ratio =

2/3	0	0	1	1/3
-----	---	---	---	-----

individual =

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

Example 2

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

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

The output is

```
number =
```

```
5
```

```
ratio =
```

```
5/12
```

```
individual =
```

```
0      1
1      3
```

Example 3

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

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

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

```
num_bit =  
    1304  
  
ratio_bit =  
    0.0652  
  
num_sym =  
    981  
  
ratio_sym =  
    0.0981
```

See Also

`symerr`, “Performance Results via Simulation”

Purpose Model binary symmetric channel

Syntax

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

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

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

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

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

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

See `rand` for details on the generator algorithm.

Examples

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

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

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

```
numerrs =  
    1509
```

```
pcterrs =  
    0.1509
```

Another example using this function is in “Binary Symmetric Channel”.

See Also

rand, awgn, “Binary Symmetric Channel”

Purpose Construct constant modulus algorithm (CMA) object

Syntax

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

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

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 “Accessing Properties of an Adaptive Algorithm”.

Property	Description
<code>AlgType</code>	Fixed value, 'Constant Modulus'

Property	Description
StepSize	CMA step size parameter, a nonnegative real number
LeakageFactor	CMA leakage factor, a real number between 0 and 1

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

$$(\text{LeakageFactor}) w + (\text{StepSize}) u^*e$$

where the $*$ operator denotes the complex conjugate.

See Also

lms, signlms, normlms, varlms, rls, lineareq, dfe, equalize, “Equalizers”

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.

Purpose Package of communications scope classes

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

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

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

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

See Also `commscope.eyediagram`

commscope.eyediagram

Purpose Eye diagram analysis

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

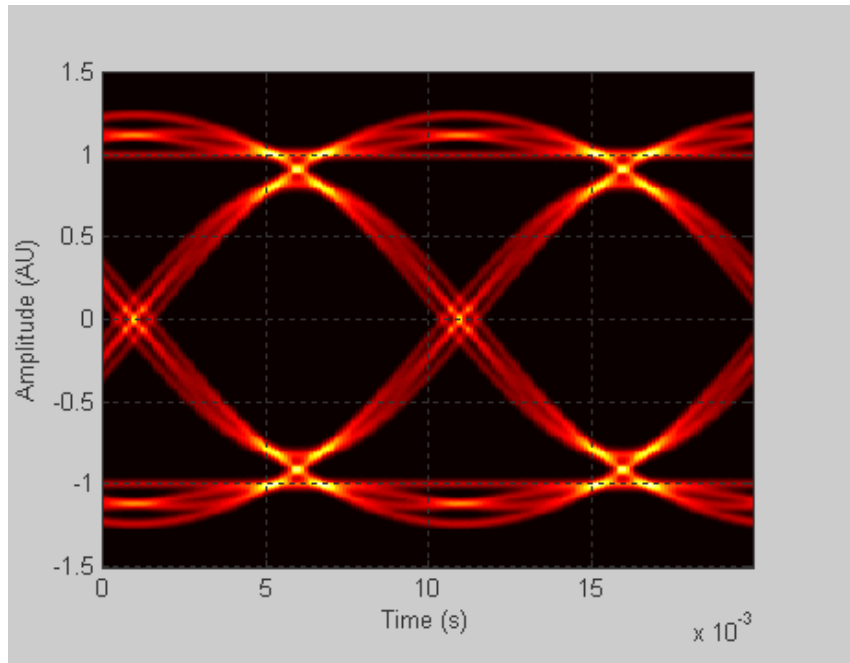
Description `h = commscope.eyediagram` constructs an eye diagram object, `h`, with default properties. This syntax is equivalent to:

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

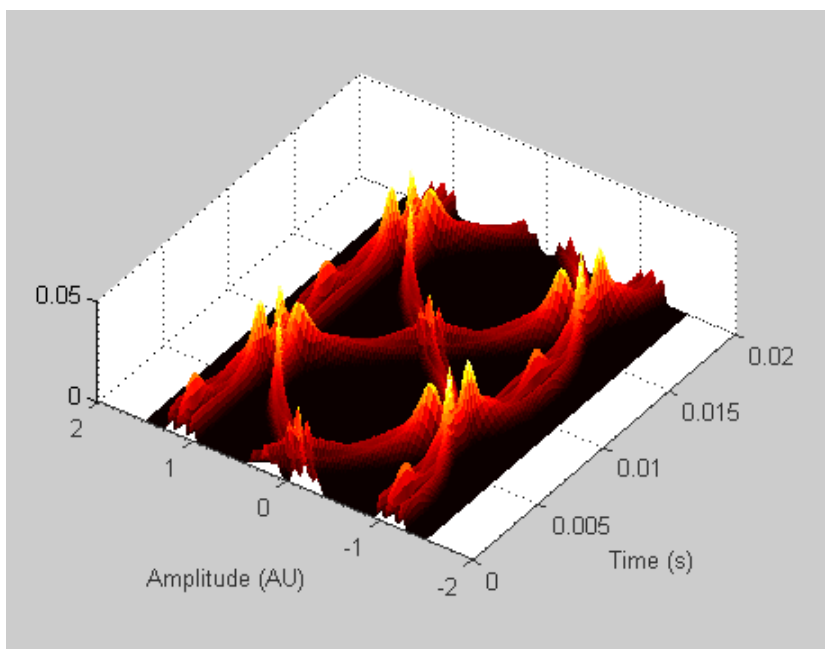
`h = commscope.eyediagram(property1,value1,...)` constructs an eye diagram object, `h`, with properties as specified by property/value pairs.

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

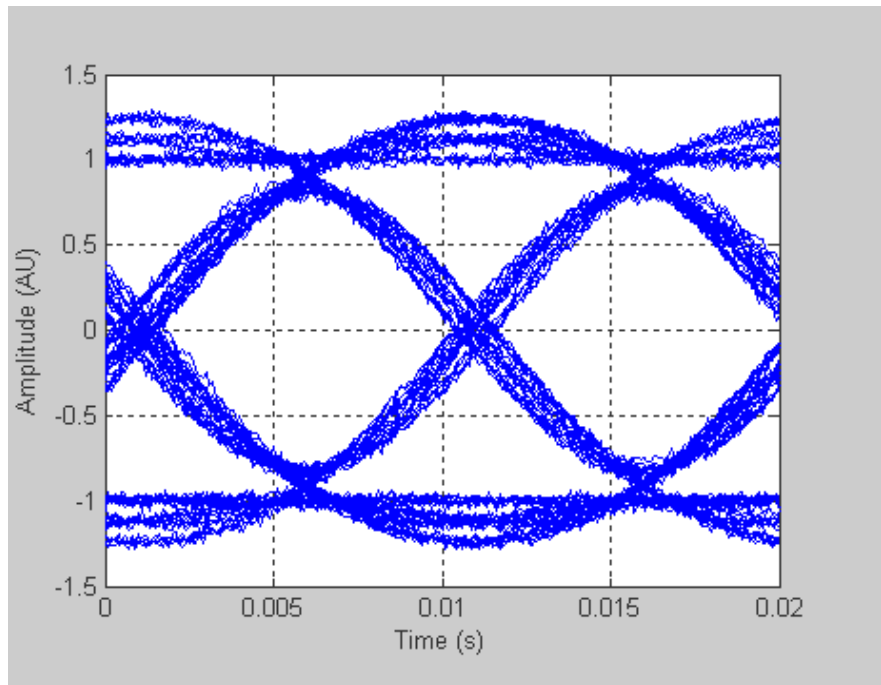
You can change the plot type by setting the PlotType property. The following plots are examples of each type.



2D-Color Eye Diagram



3D-Color Eye Diagram



2D-Line Eye Diagram

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

Properties

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

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

commscope.eyediagram

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

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

commscope.eyediagram

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

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

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

Methods

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

update

This method updates the eye diagram object data.

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

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

The following example shows this method's use:

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

% Prepare a noisy sinusoidal as input
x = awgn(0.5*sin(2*pi*(0:1/100:10))+j*cos(2*pi*(0:1/100:10)), 20);
% update the eyediagram
update(h, x);
% Check the number of processed 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
x = awgn(0.5*sin(2*pi*(0:1/100:10))+ ...
        j*0.5*cos(2*pi*(0:1/100:10)), 20);
```

```
% 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 EYEL 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
x = awgn(0.5*sin(2*pi*(0:1/100:10))+ ...
        j*0.5*cos(2*pi*(0:1/100:10)), 20);
```

```
% Update the eyediagram
update(h, x);
% Export the data
[eyec eyel 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
x = awgn(0.5*sin(2*pi*(0:1/100:10))+ ...
    j*0.5*cos(2*pi*(0:1/100:10)), 20);
update(h, x);           % update the eyediagram
h.SamplesProcessed     % Check the number of processed samples
reset(h);              % reset the object
h.SamplesProcessed     % Check the number of processed 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:

commscope.eyediagram

```
% 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 2-89 for more information.

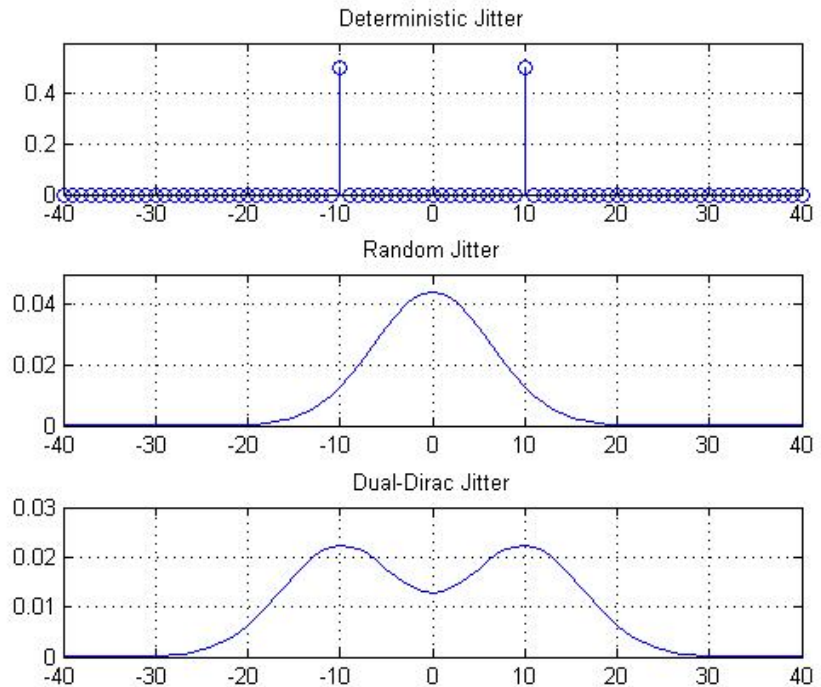
Measurements You can obtain the following measurements on an eye diagram:

- 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

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

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



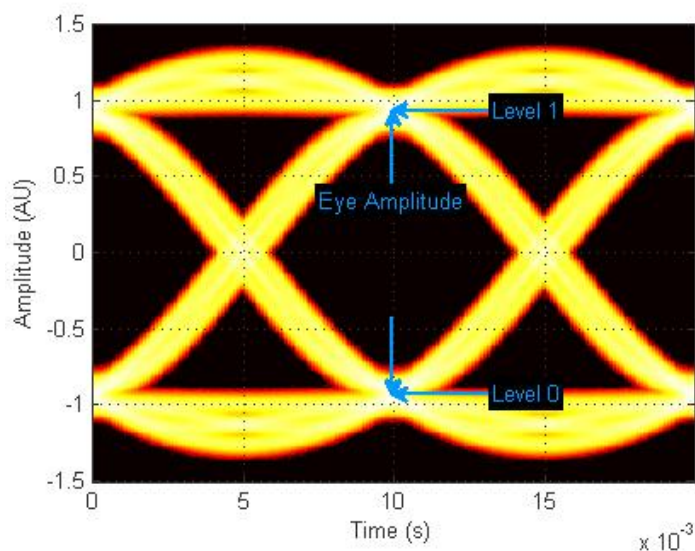
The dual-Dirac model is described in [5] in more detail. The amplitude of the two Dirac functions may not be the same. In such a case, the analyze method estimates these amplitudes, ρ_L and ρ_R .

Amplitude Measurements

You can use the vertical histogram to obtain a variety of amplitude measurements. For complex signals, measurements are done on both in-phase and the quadrature components, unless otherwise specified. For amplitude measurements, at least one bin per vertical histogram must reach 10 hits before the measurement is taken, ensuring higher accuracy.

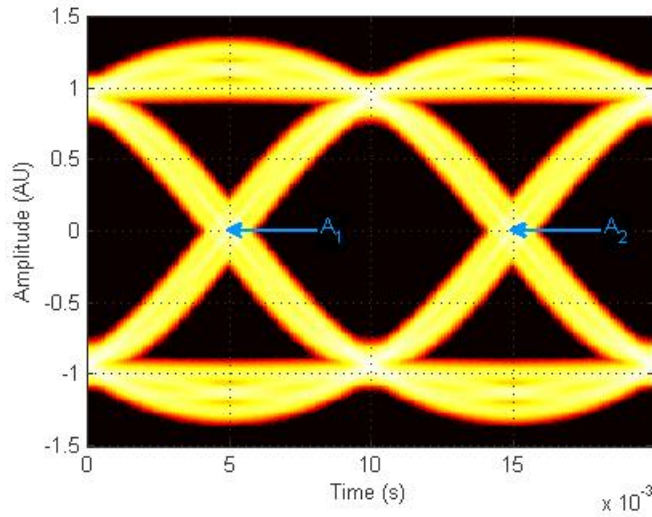
Eye Amplitude (EyeAmplitude)

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

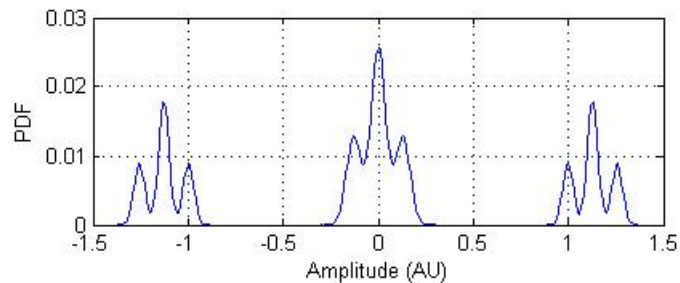


Eye Crossing Amplitude (EyeCrossingLevel)

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



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



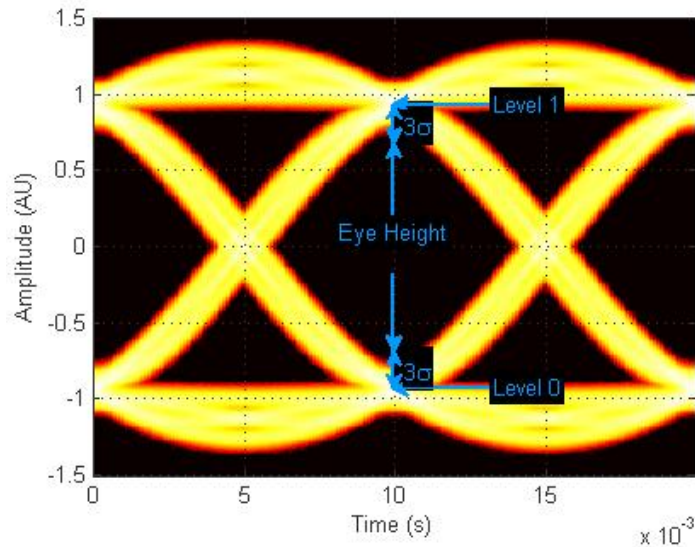
Eye Crossing Percentage (EyeOpeningVer)

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

Eye Height (EyeHeight)

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

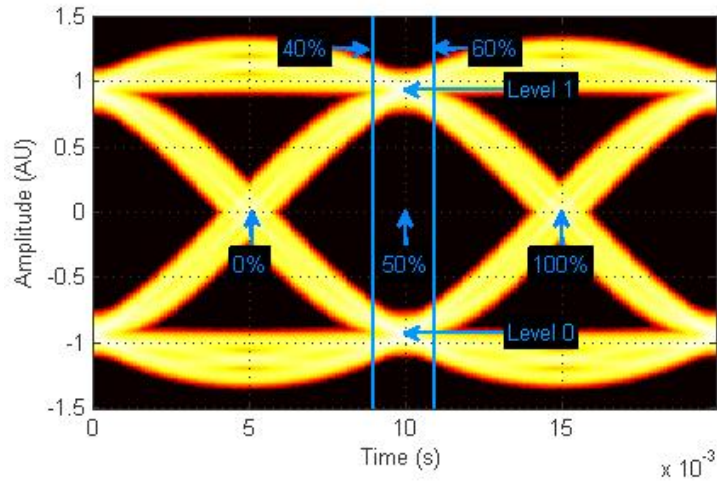
For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye height is the difference of the two 3σ points, as shown in the next figure. The 3σ point is defined as the point that is three standard deviations away from the mean value of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



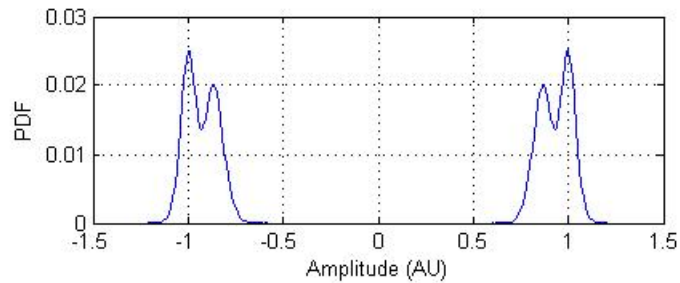
Eye Level (EyeLevel)

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

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



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



Eye SNR (EyeSNR)

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

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

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

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

Quality Factor (QualityFactor)

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

Vertical Eye Opening (EyeOpeningVer)

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

Deterministic Jitter (DeterministicJitter)

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

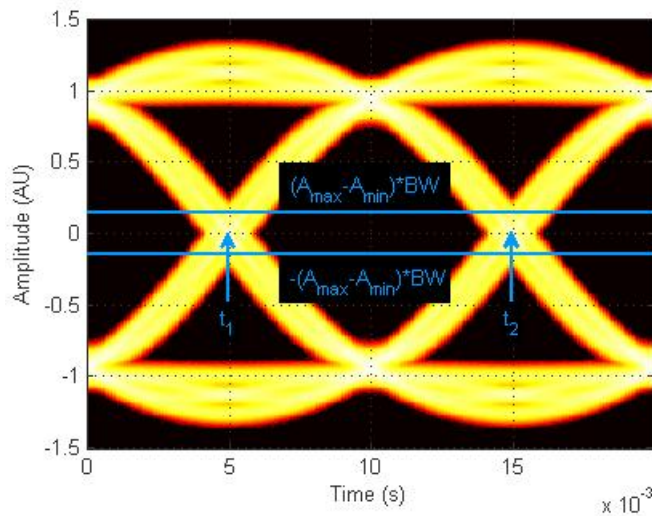
$$DJ = \mu_L - \mu_R$$

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

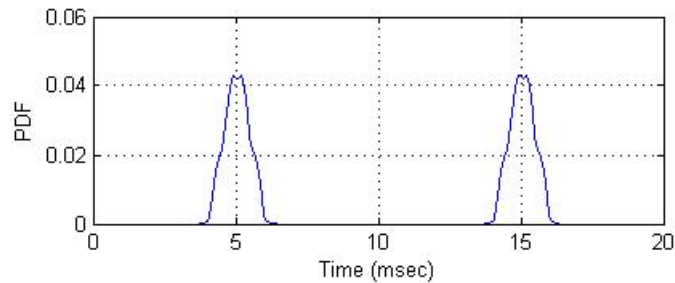
Eye Crossing Time (EyeCrossingTime)

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

The region is from $-A_{\text{total}} * BW$ to $+A_{\text{total}} * BW$, 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.

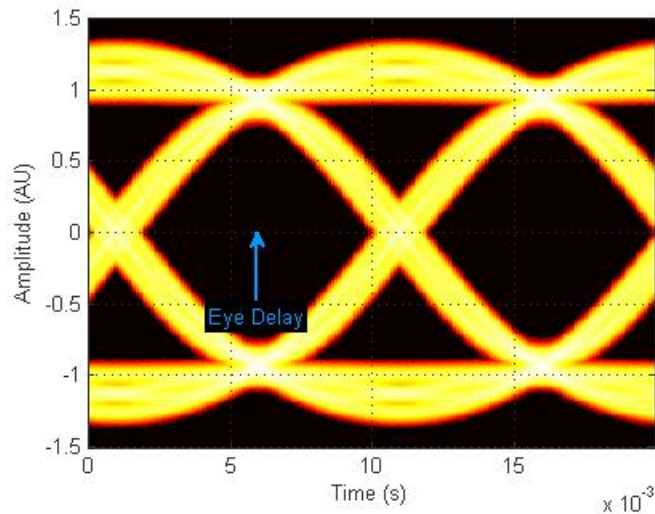


The following figure shows the average PDF in this region. Because this example assumes two symbols per trace, there are two crossing points.



Eye Delay (EyeDelay)

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



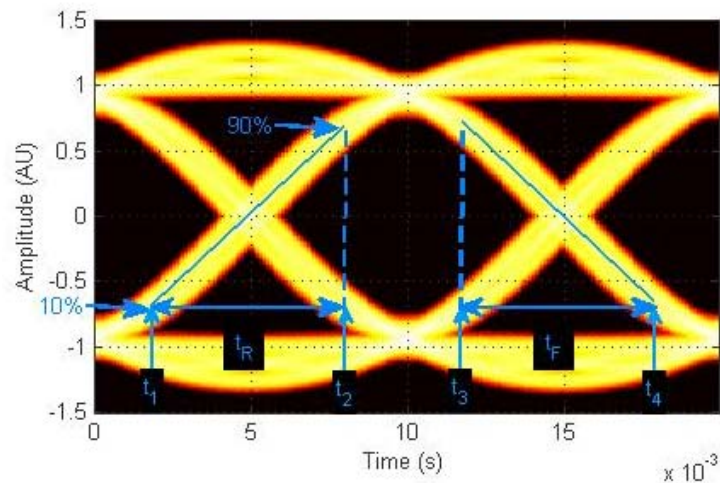
Eye Fall Time (EyeFallTime)

Eye Fall Time is the mean time between the high and low threshold values defined by the AmplitudeThreshold property of the

eyemeasurementsetup object. The previous figure shows the fall time calculated from 10% to 90% of the eye amplitude.

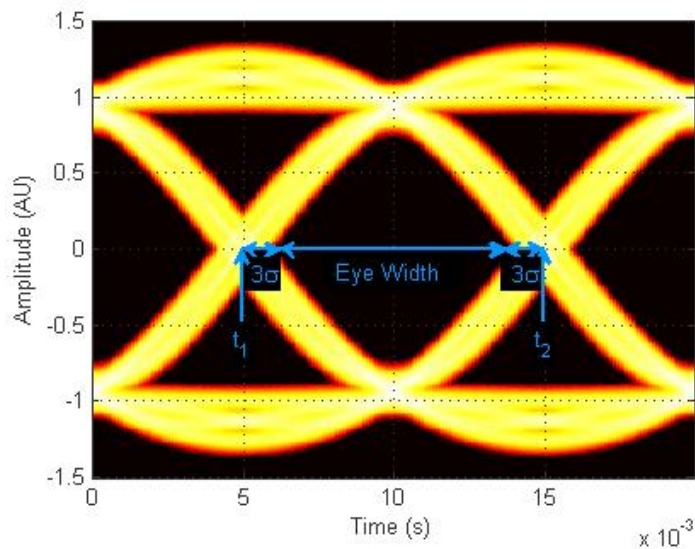
Eye Rise Time (EyeRiseTime)

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



Eye Width (EyeWidth)

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



Horizontal Eye Opening (EyeOpeningHor)

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

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

Peak-to-Peak Jitter (JitterP2P)

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

Random Jitter (RandomJitter)

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

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

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

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

RMS Jitter (JitterRMS)

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

Total Jitter (TotalJitter)

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. See *Eye Level* for more information. The default setting for *Eye level Boundaries*

is a 2x1 vector where the first element is the lower boundary and the second element is the upper boundary.

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 2x1 double vector where the first element is the lower boundary and the second element is the upper boundary.

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 $1.0000e^{-12}$.

Amplitude Threshold

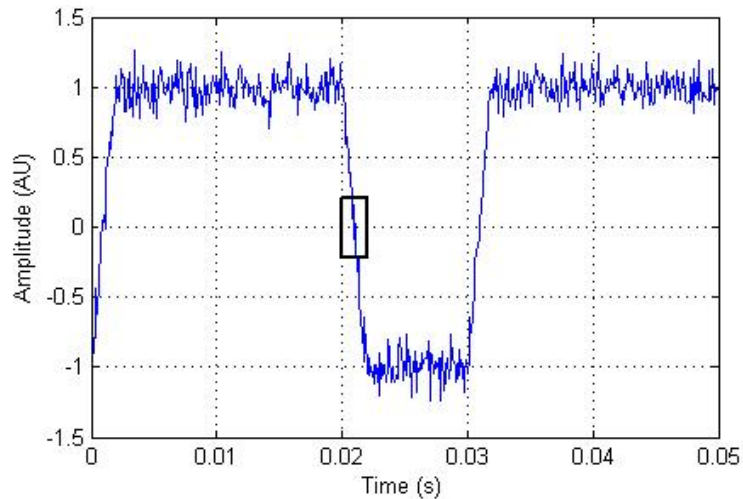
The rise time of the signal is defined as the time required for the signal to travel from lower amplitude threshold to the upper amplitude threshold. The fall time, measured from the upper amplitude threshold to the lower amplitude threshold, is defined as a percentage of the eye amplitude. The default setting is 10% for the lower threshold and 90%

for the upper threshold. See Eye Rise Time and Eye Fall Time for more information.

Jitter Hysteresis

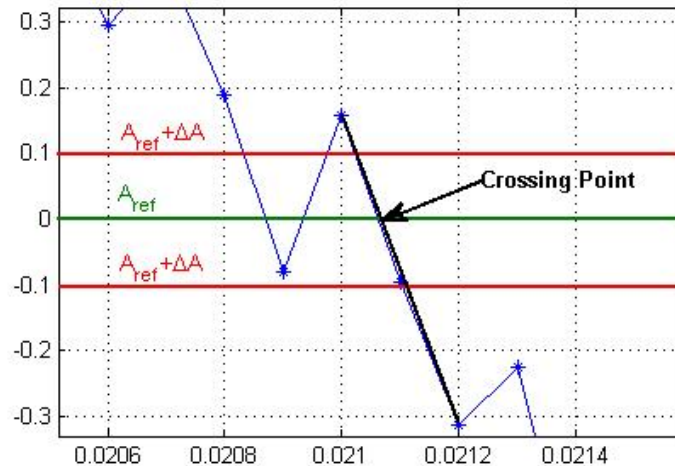
You can use the *JitterHysteresis* property of the *eyemeasurementsetup* object to remove the effect of noise from the horizontal histogram estimation. The default value for *Jitter Hysteresis* is zero.

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.

commscope.eyediagram



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)
```

See Also

commscope

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>

commscope.ScatterPlot

Purpose Create Scatter Plot scope

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

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

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

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

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

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

Property	Description
SamplingOffset	The number of samples skipped at each sampling point relative to the MeasurementDelay.
Constellation	Expected constellation of the input signal.
RefreshPlot	<p>The switch that controls the plot refresh style. The choices are:</p> <ul style="list-style-type: none"> • 'on' - The scatter plot refreshes every time the update method is called. • 'off' - The scatter plot does not refresh when the update method is called.
SamplesProcessed	The number of samples processed by the scope. This value does not include the discarded samples during the MeasurementDelay period. This property is read-only.
PlotSettings	<p>Plot settings control the scatter plot figure.</p> <ul style="list-style-type: none"> • 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

commscope.ScatterPlot

Property	Description
	<p>the constellation points. The choices are 'on' or 'off'.</p> <ul style="list-style-type: none">• 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).

commmeasure.EVM

Purpose Create EVM measurement object

Syntax
`h = commmeasure.EVM`
`h = commmeasure.EVM('PropertyName',PropertyValue,...)`

Description `h = commmeasure.EVM` constructs a default error vector magnitude (EVM) object, `h`.
`h = commmeasure.EVM('PropertyName',PropertyValue,...)` constructs an EVM object, `h`, with property values set to `PropertyValues`. See **Properties** for valid a list of valid `PropertyNames`.
The EVM object measures RMS EVM, maximum EVM, and percentile EVM. The unit for each measurement is a percentage.

Properties An EVM measurement object has the properties shown in the following table. You can write to all properties except for the ones explicitly noted otherwise.

Property	Description
Type	'EVM Measurements'. This property is read only.
RMSEVM	RMS EVM measurement result. This property is read only.
MaximumEVM	Maximum EVM measurement result. This property is read only.
Percentile	Percentile value to calculate PercentileEVM.
PercentileEVM	Percentile EVM measurement result. This property is read only.
NumberOfSymbols	Number of processed symbols.

Methods An EVM measurement object has the following three methods.

update

This method updates the EVM measurements with new data.

`update(h, rcv, xmt)` updates the EVM object `h` with new data. RCV represents the symbols under test, and XMT represents the ideal symbols.

reset

This method resets the EVM object.

`reset(h)` resets the EVM object `h`. This operation removes all the previously collected data from the object memory.

`reset(h, meas1, ...)` resets the `meas1` measurement of the EVM object `h`. `meas1` can be 'RMSEVM', 'MaximumEVM', or 'PercentileEVM'. The `NumberOfSymbols` property is not reset. You can use this method for implementing frame-based measurements.

copy

This method copies the EVM object.

`hcopy = copy(h)` copies the EVM object handle returns it in `hcopy`. `h` and `hcopy` are independent but identical objects; modifying the object does not affect `hcopy` object.

Algorithm for EVM Calculations

The block calculates EVM through I-Q value samples at the symbol time. If $x(t)$ is the ideal transmitter signal and $y(t)$ is the signal under test, then the error vector is defined as $e(n) = y(n) - x(n)$ where $n = nT$ and T is the symbol duration.

The RMS EVM for a frame is defined as

$$EVM = \sqrt{\frac{P_{error}}{P_{reference}}}$$

The RMS EVM for a symbol is defined as

$$EVM(n) = \sqrt{\frac{|e(n)|^2}{\frac{P_{reference}}{L}}}$$

where

$$P_{error} = \sum_{n=0}^{L-1} |e(n)|^2$$

and

$$P_{reference} = \sum_{n=0}^{L-1} |x(n)|^2 .$$

$$EVM(\%) = EVM * 100$$

See Also

commmeasure.MER

References

[1] 3GPP TS 45.005 V8.1.0 (2008-05): Radio Access Network; Radio transmission and reception

Purpose Create MER measurement object

Syntax
`h = commmeasure.MER`
`h = commmeasure.MER('PropertyName',PropertyValue,...)`

Description
`h = commmeasure.MER` returns a default modulation error ratio (MER) object, `h`.
`h = commmeasure.MER('PropertyName',PropertyValue,...)` returns an MER object `H`, with property values set to `PropertyValues`. See `Properties` for a list of valid `PropertyNames`.
 The MER object can be used to measure MER, minimum MER, and percentile MER, all in decibels.

Properties
 An MER measurement object has the properties shown in the following table. You can write to all properties except for the ones explicitly noted otherwise.

Property	Description
Type	'MER Measurements'. This property is read only.
MERdB	MER measurement result (in decibels). This property is read-only.
MinimumMER	Minimum MER measurement result (in decibels). This property is read only.
Percentile	Percentile value to calculate PercentileMER.
PercentileMER	Percentile MER measurement result (in decibels). This property is read only.
NumberOfSymbols	Number of processed symbols.

Methods

An MER measurement object has the following three methods.

update

This method updates the MER measurements with new data.

`update(H, RCV, XMT)` updates the MER object `H` with new data. `RCV` represents the symbols under test and `XMT` represents the ideal symbols.

reset

This method resets the MER object.

`reset(h)` resets the MER object `h`. This operation removes all the previously collected data from the object's memory.

`reset(h, meas1, ...)` resets the `meas1` measurement of the EVM object `h`. `meas1` can be 'MERdB', 'MinimumMER', or 'PercentileMER'. `NumberOfSymbols` property is not reset. You can use this format for implementing frame-based measurements.

copy

This method copies the MER object.

`hcopy = copy(h)` copies the MER object `h` and returns it `hcopy`. `h` and `hcopy` are independent but identical objects; modifying the object `h` does not affect the object `hcopy`.

Algorithm for MER Calculations

The block calculates the MER through I-Q value samples at the symbol time. If $x(t)$ is the ideal transmitter signal and $y(t)$ is the signal under test, then the error vector is defined as $e(n) = y(n) - x(n)$ where $n = nT$ and T is the symbol duration.

The MER for a frame, in dB, is defined as

$$MER = 10 \log_{10} \left(\frac{P_{reference}}{P_{error}} \right)$$

The MER for a symbol, in dB, is defined as

$$MER(n) = 10 \log_{10} \left(\frac{P_{reference}}{\frac{L}{|e(n)|^2}} \right)$$

where

$$P_{error} = \sum_{n=0}^{L-1} |e(n)|^2$$

and

$$P_{reference} = \sum_{n=0}^{L-1} |x(n)|^2.$$

See Also

commmeasure.EVM

References

[1] Grieve, David. "Measurement guidelines for Digital Video Broadcasting (DVB) systems." European Telecommunications Standards Institute (ETSI): *ETSI Technical Report*, ETR 290.

commsrc.pattern

Purpose Construct pattern generator object

Syntax `h = commsrc.pattern`

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

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

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

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

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

Property	Description
SamplesPerSymbol	The number of samples representing a symbol. <code>SamplesPerSymbol</code> must be an integer. This property affects <code>SymbolRate</code> .
PulseType	The type of pulse the object generates. Pulse types available: return-to-zero (RZ) and nonreturn-to-zero (NRZ).
OutputLevels	The amplitude levels corresponding to the logical low and high values of the pulse.
DutyCycle	The duty cycle of the pulse the object generates. Displays calculated duty cycle based on pulse parameters. This property is not writable.
RiseTime	Specifies 10% to 90% rise time of the pulse in seconds.
PulseDuration	Pulse duration in seconds defined by IEEE STD 181 standard. (See the Return-to-Zero (RZ) Signal Conversion: Ideal Pulse to STD-181 figure in the Methods section.) Setting <code>PulseType</code> to return-to-zero enables this property.
FallTime	Specifies 10% to 90% fall time of the pulse in seconds.
DataPattern	The bit sequence the object uses. The following patterns are available: PRBS5 to PRBS15, PRBS23, PRBS31, and User Defined.

Property	Description
UserDataPattern	User-defined bit pattern consisting of a vector of ones and zeroes. Setting data pattern to user defined enables this property.
Jitter	Specifies jitter characteristics. Use this property to configure Random, Periodic and Dual Dirac Jitter.

Methods

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

generate

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

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

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

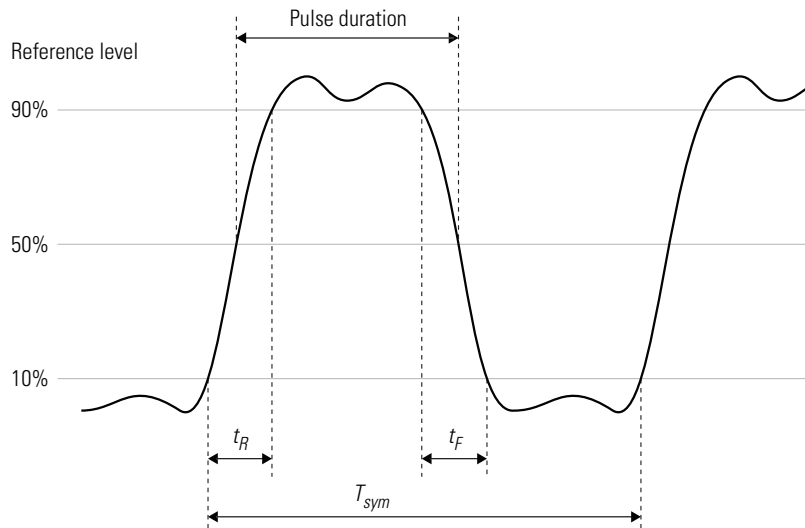
reset

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

idealtostd181

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

```
idealtostd181(tR, tF, PW)
```



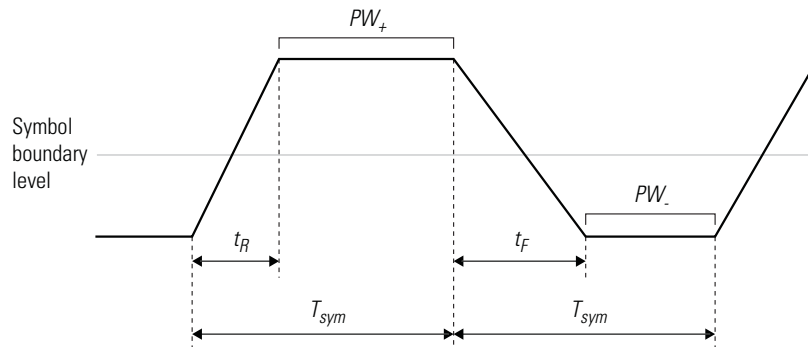
IEEE STD-181 Return-to-Zero (RZ) Signal Parameters

std181toideal

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

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

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



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

computedcd

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

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

$$\text{dcd} = \text{computedcd}(h)$$

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

$$T_h = (A_h - A_l) * \frac{t_R}{A_l} + (A_h - A_l) * \frac{t_F}{A_l} + PW_+$$

$$T_l = (A_h - A_l) * \frac{t_R}{A_l} + (A_h - A_l) * \frac{t_F}{A_l} + PW_-$$

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

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.

commsrc.pn

Purpose Create PN sequence generator package

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

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

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

or

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

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

Properties

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

Property	Description
GenPoly	Generator polynomial vector array of bits; must be descending order

Property	Description
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 <code>generate</code> 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.

`seqgen.pn` objects also have either a 'Mask' (vector of mask bits) or 'Shift' (scalar shift value) property.

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 "LFSR SSRG Details" on page 2-558.

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

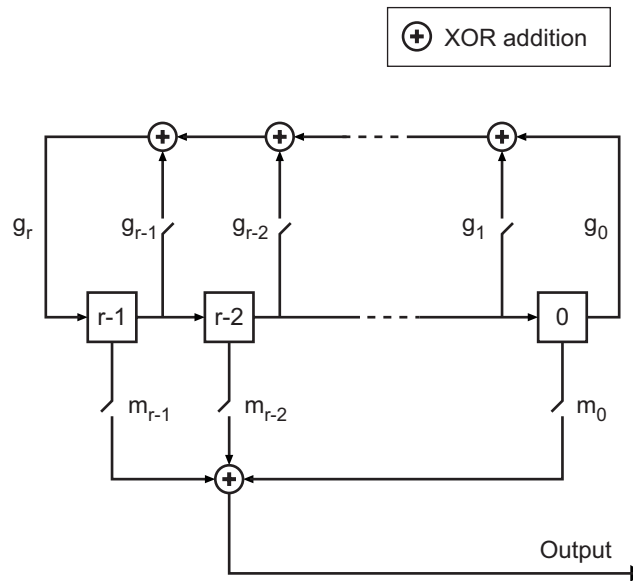
Every time this property is set, it will reset the entire object. In addition to changing the polynomial values, 'CurrentStates', 'InitialStates', and 'Mask' will be set to their default values ('NumBitsOut' will remain the same), and no warnings will be issued.

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 , $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \dots + g_0$. The coefficient g_k is 1 if there is a connection from the k th 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 $\text{length}(g1) - 1$	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 , $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + \dots + m_1z + m_0$, 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 Toolbox function `shift2mask`.

Sequences of Maximum Length

If you want to generate a sequence of the maximum possible length for a fixed degree, r , of the generator polynomial, you can set **Generator polynomial** to a value from the following table. See Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill,

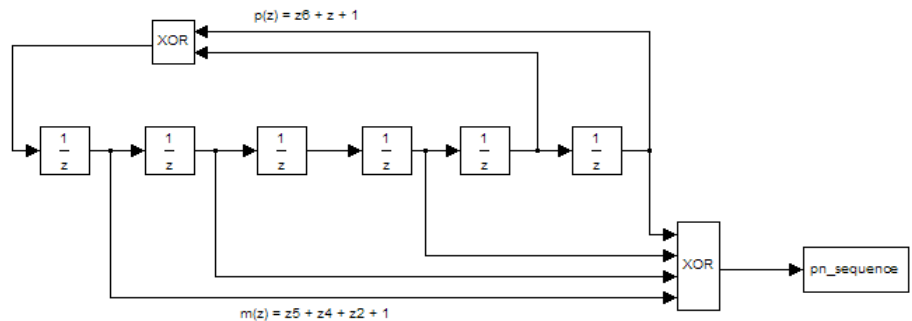
1995 for more information about the shift-register configurations that these polynomials represent.

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

r	Generator Polynomial	r	Generator Polynomial
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 the PN sequence generator



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

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

The output of the example is given below:

ans =

22

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

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

General Use of commsrc.pn

The following is an example of typical usage:

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

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

% Output 10 more PN bits
generate(h)

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

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

Behavior of a Copied commsrc.pn Object

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

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

h generates the sequence:

1


```
0
0
0
0
```

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

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

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

```
0
1
0
0
0
```

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

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

then it generates the same sequence that `h` did:

```
1
0
0
0
0
```

See Also

`mask2shift`, `shift2mask`

comband

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

Syntax

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

Description

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

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

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

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

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

The output is

```
compressed =
    3.5296    4.1629    4.5333    4.7961    5.0000

expanded =
    1.0000    2.0000    3.0000    4.0000    5.0000
```

Algorithm

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

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

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

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

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

See Also

`quantiz`, `dpcmenco`, `dpcmdeco`, “Companding a Signal”

References

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

Purpose

Restore ordering of symbols using shift registers

Syntax

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

Description

`deintrlved = convdeintrlv(data,nrows,slope)` restores the ordering of elements in `data` by using a set of `nrows` internal shift registers. The delay value of the k th shift register is $(nrows - k) * slope$, where $k = 1, 2, 3, \dots, nrows$. 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”.

convdeintrlv

Examples

The example in “Effect of Delays on Recovery of Convolutionally Interleaved Data” 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`, “Interleaving”

Purpose

Convolutionally encode binary data

Syntax

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

Description

`code = convenc(msg,trellis)` encodes the binary vector `msg` using the convolutional encoder whose MATLAB trellis structure is `trellis`. For details about MATLAB trellis structures, see “Trellis Description of a Convolutional Encoder”. Each symbol in `msg` consists of $\log_2(\text{trellis.numInputSymbols})$ bits. The vector `msg` contains one or more symbols. The output vector `code` contains one or more symbols, each of which consists of $\log_2(\text{trellis.numOutputSymbols})$ 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 1s and 0s, where the 0s indicate the punctured bits. `puncpat` must have a length of at least $\log_2(\text{trellis.numOutputSymbols})$ 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

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

```
code1 = convenc(randint(10,1,2,123),...
poly2trellis([5 4],[23 35 0; 0 5 13]));
```

The commands below define the encoder's trellis structure explicitly and then use `convenc` to encode 10 one-bit symbols. A schematic of this encoder is in “Trellis Description of a Convolutional Encoder”.

```
trell = 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(randint(10,1),trell);
```

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

```
trell = poly2trellis([5 4],[23 35 0; 0 5 13]);
msg = randint(10,1,2,123);
% Encode part of msg, recording final state for later use.
[code3,fstate] = convenc(msg(1:6),trell);
% Encode the rest of msg, using state as an input argument.
code4 = convenc(msg(7:10),trell,fstate);
```

Examples

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

See Also

`distspec`, `vitdec`, `poly2trellis`, `istrellis`, `vitsimdemo`,
“Convolutional Coding”

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.

convintrlv

Purpose Permute symbols using shift registers

Syntax

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

Description `intrlved = convintrlv(data,nrows,slope)` permutes the elements in `data` by using a set of `nrows` internal shift registers. The delay value of the k th shift register is $(k-1)*slope$, where $k = 1, 2, 3, \dots, nrows$. 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 = randint(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 y_1 , obtained using `muxintrlv`, are the same.

```
ans =
```

```
    1
```

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

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

References

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

See Also

`convdeintrlv`, `muxintrlv`, `helintrlv`, “Interleaving”

convmtx

Purpose Convolution matrix of Galois field vector

Syntax `A = convmtx(c,n)`

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

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

- $A \cdot x$, 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.
- $x \cdot A$, 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(randint(n,1,2^m),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, “Signal Processing Operations in Galois Fields”

cosets

Purpose Produce cyclotomic cosets for Galois field

Syntax `cst = cosets(m)`

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

A cyclotomic coset is a set of elements that share the same minimal polynomial. Together, the cyclotomic cosets mod $2^m - 1$ form a partition of the group of nonzero elements of $GF(2^m)$. For more details on cyclotomic cosets, see the works listed in “References” on page 2-145.

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
```

See Also

minpol

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.

crc.detector

Purpose Construct CRC detector object

Syntax

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

Description

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

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

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

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

```
h= crc.detector('Polynomial', '0x1021', 'InitialState',
'0xFFFF', 'ReflectInput', ...
false, 'ReflectRemainder', false, 'FinalXOR', '0x0000')
```

Properties

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

Property	Description
Type	Specifies the object as a 'CRC Detector'.

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 bitwise basis prior to entering the shift register.

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

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, refer to the Cyclic Redundancy Check Coding section of the Communications 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.

Usage 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])

% 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])

% 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')

% 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
```

crc.detector

```
% 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
```

See Also

`crc.generator`

Purpose Construct CRC generator object

Syntax

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

Description

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

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

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

`h = crc.generator` constructs a CRC generator object H with default properties. It constructs a CRC-CCITT generator, and is equivalent to:
`h = crc.generator('Polynomial', '0x1021', 'InitialState', '0xFFFF', ... 'ReflectInput', false, 'ReflectRemainder', false, 'FinalXOR', '0x0000')`.

Properties

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

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

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

CRC Generation Algorithm

For information pertaining to the CRC generation algorithm, refer to the Cyclic Redundancy Check Coding section of the Communications 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.

crc.generator

```
% 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  $x^3+x+1$ , 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])

% 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);
```

See Also

[crc.detector](#)

Purpose

Produce parity-check and generator matrices for cyclic code

Syntax

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

Description

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

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

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

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

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

Examples

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

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

The output is

```
parmat =  
  
    1    0    0    1    1    1    0  
    0    1    0    0    1    1    1  
    0    0    1    1    1    0    1
```

```
genmat =  
  
    1    0    1    1    0    0    0  
    1    1    1    0    1    0    0  
    1    1    0    0    0    1    0  
    0    1    1    0    0    0    1
```

```
k =
```

```
4
```

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

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

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

See Also

encode, decode, bchgenpoly, cyclpoly, “Block Coding”

Purpose Produce generator polynomials for cyclic code

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

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

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

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

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

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

Examples

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

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

The output is

```
c1 =
```

```
Columns 1 through 10
```

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

```
Columns 11 through 12
```

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

```
c2 =
```

```
Columns 1 through 10
```

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

```
Columns 11 through 12
```

```
0      1
```

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

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

```
No generator polynomial satisfies the given constraints.
```

```
c3 =
```

```
[]
```

Algorithm

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`, “Block Coding”

de2bi

Purpose Convert decimal numbers to binary vectors

Syntax

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

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

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

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

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

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

`b = de2bi(d,...,flag)` uses the string `flag` to determine whether the first column of `b` contains the lowest-order or highest-order

digits. Values for *flag* are 'right-msb' and 'left-msb'. The value 'right-msb' produces the default behavior.

Examples

The code below counts to 10 in decimal and binary.

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

The output is below.

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

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

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

bb =

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

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

de2bi

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

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

```
tleft = de2bi(12,[],3,'left-msb') % Significant digit on left
```

The output is

```
t =
```

```
    0    1    1
```

```
tleft =
```

```
    1    1    0
```

See Also

[bi2de](#)

Purpose Block decoder

Syntax

```

msg = decode(code,n,k,'hamming/fmt',prim_poly)
msg = decode(code,n,k,'linear/fmt',genmat,trt)
msg = decode(code,n,k,'cyclic/fmt',genpoly,trt)
msg = decode(code,n,k)
[msg,err] = decode(...)
[msg,err,ccode] = decode(...)
[msg,err,ccode,cerr] = decode(...)

```

Optional Inputs

Input	Default Value
<i>fmt</i>	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 2-211 section on the encode reference page explains the meanings of *n* and *k*, the possible values of *fmt*, and the possible formats for *code* and *msg*. You should be familiar with the conventions described there before reading the rest of this section. Using the decode function with an input argument *code* that was *not* created by the encode function might cause errors.

For Specific Syntaxes

`msg = decode(code,n,k,'hamming/fmt',prim_poly)` decodes `code` using the Hamming method. For this syntax, `n` must have the form 2^m-1 for some integer `m` greater than or equal to 3, and `k` must equal `n-m`. `prim_poly` is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive polynomial for $GF(2^m)$ that is used in the encoding process. The default value of `prim_poly` is `gfprimd(m)`. The decoding table that the function uses to correct a single error in each codeword is `syndtable(hammgen(m))`.

`msg = decode(code,n,k,'linear/fmt',genmat,trt)` decodes `code`, which is a linear block code determined by the `k`-by-`n` generator matrix `genmat`. `genmat` is required as input. `decode` tries to correct errors using the decoding table `trt`, where `trt` is a $2^{(n-k)}$ -by-`n` matrix.

`msg = decode(code,n,k,'cyclic/fmt',genpoly,trt)` decodes the cyclic code `code` and tries to correct errors using the decoding table `trt`, where `trt` is a $2^{(n-k)}$ -by-`n` matrix. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial of the code. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an $[n, k]$ cyclic code must have degree `n-k` and must divide x^n-1 .

`msg = decode(code,n,k)` is the same as
`msg = decode(code,n,k,'hamming/binary')`.

`[msg,err] = decode(...)` returns a column vector `err` that gives information about error correction. If the code is a convolutional code, `err` contains the metric calculations used in the decoding decision process. For other types of codes, a nonnegative integer in the `r`th row of `err` indicates the number of errors corrected in the `r`th *message* word; a negative integer indicates that there are more errors in the `r`th 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 `r`th row of `vec2matcerr` indicates the number of errors corrected in the `r`th *codeword*; a negative integer indicates that there are more errors in the `r`th codeword than can be corrected.
- If `code` is not a binary vector, `cerr = err`.

Examples

On the reference page for `encode`, some of the example code illustrates the use of the `decode` function.

The example below illustrates the use of `err` and `cerr` when the coding method is not convolutional code and the code is a binary vector. The script encodes two five-bit messages using a cyclic code. Each codeword has 15 bits. Errors are added to the first two bits of the first codeword and the first bit of the second codeword. Then `decode` is used to recover the original message. As a result, the errors are corrected. `err` reflects the fact that the first *message* was recovered after correcting two errors, while the second message was recovered after correcting one error. `cerr` reflects the fact that the first *codeword* was decoded after correcting two errors, while the second codeword was decoded after correcting one error.

```
m = 4; n = 2^m-1; % Codeword length is 15.
k = 5; % Message length
msg = ones(10,1); % Two messages, five bits each
code = encode(msg,n,k,'cyclic'); % Encode the message.
% Now place two errors in first word and one error
% in the second word. Create errors by reversing bits.
noisycode = code;
noisycode(1:2) = bitxor(noisycode(1:2),[1 1]');
noisycode(16) = bitxor(noisycode(16),1);
% Decode and try to correct the errors.
[newmsg,err,cerr] = decode(noisycode,n,k,'cyclic');
disp('Transpose of err is'); disp(err')
disp('Transpose of cerr is'); disp(cerr')
```

The output is below.

decode

Single-error patterns loaded in decoding table.

1008 rows remaining.

2-error patterns loaded. 918 rows remaining.

3-error patterns loaded. 648 rows remaining.

4-error patterns loaded. 243 rows remaining.

5-error patterns loaded. 0 rows remaining.

Transpose of err is

2 1

Transpose of cerr is

2 1

Algorithm

Depending on the decoding method, `decode` relies on such lower-level functions as `hammgen`, `syndtable`, and `cyclgen`.

See Also

`encode`, `cyclpoly`, `syndtable`, `gen2par`, “Block Coding”

Purpose Restore ordering of symbols

Syntax `deintrlvd = deintrlv(data,elements)`

Description `deintrlvd = deintrlv(data,elements)` restores the original ordering of the elements of `data` by acting as an inverse of `intrlv`. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. To use this function as an inverse of the `intrlv` function, use the same `elements` input in both functions. In that case, the two functions are inverses in the sense that applying `intrlv` followed by `deintrlv` leaves data unchanged.

Examples The code below illustrates the inverse relationship between `intrlv` and `deintrlv`.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p); % Rearrange [10 20 30 ... 100].
b = deintrlv(a,p) % Deinterleave a to restore ordering.
```

The output is

```
b =
    10    20    30    40    50    60    70    80    90   100
```

See Also `intrlv`, “Interleaving”

Purpose Construct decision-feedback equalizer object

Syntax

```
eqobj = dfe(nfwdweights,nfbkweights,alg)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst,nsamp)
```

Description The `dfe` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Using Adaptive Equalizer Functions and Objects”.

`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 [n fwdweights, n fbkweights]. 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 n fwdweights. Setting this to a value greater than 1 effectively delays the reference signal with respect to the equalizer's input signal.
SigConst	Signal constellation, a vector whose length is typically a power of 2.

Property	Description
Weights	Vector that concatenates the complex coefficients from the forward filter and the feedback filter. This is the set of w_i values in the schematic in “Decision-Feedback Equalizers”.
WeightInputs	Vector that concatenates the tap weight inputs for the forward filter and the feedback filter. This is the set of u_i values in the schematic in “Decision-Feedback Equalizers”.
ResetBeforeFiltering	If 1, each call to <code>equalize</code> resets the state of <code>eqobj</code> 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 <code>eqobj</code> , this property value is 0.
Properties specific to the adaptive algorithm represented by <code>alg</code>	See reference page for the adaptive algorithm function that created <code>alg</code> : <code>lms</code> , <code>signlms</code> , <code>normlms</code> , <code>varlms</code> , <code>rls</code> , or <code>cma</code> .

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	<code>zeros(1,sum(nWeights))</code>

Property	Adjusted Value
WeightInputs	<code>zeros(1,sum(nWeights))</code>
StepSize (Variable-step-size LMS equalizers)	<code>InitStep*ones(1,sum(nWeights))</code>
InvCorrMatrix (RLS equalizers)	<code>InvCorrInit*eye(sum(nWeights))</code>

An example illustrating relationships among properties is in “Linked Properties of an Equalizer Object”.

Examples

An example is in “Defining an Equalizer Object”.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `equalize`,
“Equalizers”

Purpose Discrete Fourier transform matrix in Galois field

Syntax `dm = dftmtx(alph)`

Description `dm = dftmtx(alph)` returns a Galois array that represents the discrete Fourier transform operation on a Galois vector, with respect to the Galois scalar `alph`. The element `alph` is a primitive n th 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 n th 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(randint(n,1,2^m),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 `n`th root of unity in the Galois field $GF(2^m)$, where `m` is an integer between 1 and 8.

Algorithm

The element $dm(a,b)$ equals $alph^{((a-1)*(b-1))}$.

See Also

`fft`, `ifft`, “Signal Processing Operations in Galois Fields”

distspec

Purpose Compute distance spectrum of convolutional code

Syntax
`spect = distspec(trellis,n)`
`spect = distspec(trellis)`

Description `spect = distspec(trellis,n)` computes the free distance and the first `n` components of the weight and distance spectra of a linear convolutional code. Because convolutional codes do not have block boundaries, the weight spectrum and distance spectrum are semi-infinite and are most often approximated by the first few components. The input `trellis` is a valid MATLAB trellis structure, as described in “Trellis Description of a Convolutional Encoder”. The output, `spect`, is a structure with these fields:

Field	Meaning
<code>spect.dfree</code>	Free distance of the code. This is the minimum number of errors in the encoded sequence required to create an error event.
<code>spect.weight</code>	A length- <code>n</code> vector that lists the total number of information bit errors in the error events enumerated in <code>spect.event</code> .
<code>spect.event</code>	A length- <code>n</code> vector that lists the number of error events for each distance between <code>spect.dfree</code> and <code>spect.dfree+n-1</code> . The vector represents the first <code>n</code> 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.
```

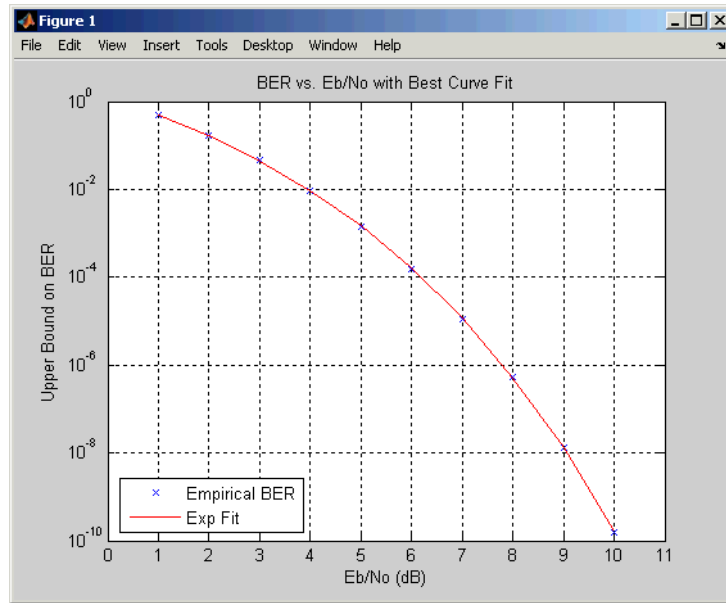
The output and plot are below.

```
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]
```



Algorithm

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, and poly2trellis

doppler

Purpose	Package of Doppler classes
Description	This package contains the classes that instantiate Doppler objects. These objects are used as values of the DopplerSpectrum property, which is common to both Rayleigh and Rician channel objects.
Properties and Methods	Every Doppler object has a read-only SpectrumType property. Other properties are specific to each Doppler class. Every Doppler object has a copy method, to duplicate itself, and a disp method, to display its properties.
See Also	doppler.ajakes, doppler.bell, doppler.bigaussian, doppler.flat, doppler.gaussian, doppler.jakes, doppler.rjakes, doppler.rounded, “Fading Channels”, rayleighchan, ricianchan, and stdchan

Purpose

Construct asymmetrical Doppler spectrum object

Syntax

```
dop = doppler.ajakes(freqminmaxajakes)
dop = doppler.ajakes
```

Description

The `doppler.ajakes` function creates an asymmetrical Jakes (AJakes) Doppler spectrum object. This object is to be used for the `DopplerSpectrum` property of a channel object created with the `rayleighchan` or the `ricianchan` functions.

`dop = doppler.ajakes(freqminmaxajakes)`, where `freqminmaxajakes` is a row vector of two finite real numbers between -1 and 1, creates a Jakes Doppler spectrum that is nonzero only for normalized (by the maximum Doppler shift f_d , in Hz) frequencies f_{norm} such that $-1 \leq f_{min,norm} \leq f_{norm} \leq f_{max,norm} \leq 1$, where $f_{min,norm}$ is given by `freqminmaxajakes(1)` and $f_{max,norm}$ is given by `freqminmaxajakes(2)`. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically: $f_{min,norm} = f_{min} / f_d$ and $f_{max,norm} = f_{max} / f_d$, where f_{min} is the minimum Doppler shift (in hertz) and f_{max} is the maximum Doppler shift (in hertz).

When `dop` is used as the `DopplerSpectrum` property of a channel object, space `freqminmaxajakes(1)` and `freqminmaxajakes(2)` by more than 1/50. Assigning a smaller spacing results in `freqminmaxajakes` 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 $-f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$. 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:

$$S(f) = \frac{A_a}{\pi f_d \sqrt{1 - (f / f_d)^2}}, \quad -f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$$

$$A_a = \frac{1}{\pi \left[\sin^{-1} \left(\frac{f_{\max}}{f_d} \right) - \sin^{-1} \left(\frac{f_{\min}}{f_d} \right) \right]}$$

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 $f_{\min, \text{norm}} = -0.2$ and maximum normalized Doppler shift $f_{\max, \text{norm}} = 0.05$. 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 $-f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$, where $f_{\min} = f_{\min, \text{norm}} \times f_d = -2$ Hz and $f_{\max} = f_{\max, \text{norm}} \times f_d = 0.5$ Hz.

```
chan = rayleighchan(1/1000, 10);
dop_ajakes = doppler.ajakes([-0.2 0.05]);
chan.DopplerSpectrum = dop_ajakes;
chan.DopplerSpectrum
```

This code returns:

```
SpectrumType: 'AJakes'
FreqMinMaxAJakes: [-0.2000 0.0500]
```

References

- [1] Jakes, W. C., Ed., *Microwave Mobile Communications*, Wiley, 1974.
- [2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.
- [3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

doppler, doppler.bell, doppler.biggaussian, doppler.flat, doppler.gaussian, doppler.jakes, doppler.rjakes,

doppler.rounded, rayleighchan, ricianchan, stdchan, and “Fading Channels”

Purpose Construct bell-shaped Doppler spectrum object

Syntax `doppler.bell`
`doppler.bell(coeffbell)`

Description `doppler.bell` creates a bell Doppler spectrum object. You can use this object with the `DopplerSpectrum` property of any channel object created with either the `rayleighchan` function, the `ricianchan` function, or the `mimochan` function.

`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 $f = 0$.

References

[1] IEEE P802.11 Wireless LANs, “TGn Channel Models”, IEEE 802.1103/940r4, 2004-05-10.

See Also

`doppler`, `doppler.ajakes`, `doppler.flat`, `doppler.gaussian`, `doppler.jakes`, `doppler.rjakes`, `doppler.rounded`, `rayleighchan`, `ricianchan`, `stdchan`, and “Fading Channels”

Purpose Construct bi-Gaussian Doppler spectrum object

Syntax
`dop = doppler.bigaussian(property1,value1,...)`
`dop = doppler.bigaussian`

Description The `doppler.bigaussian` function creates a bi-Gaussian Doppler spectrum object to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` function or the `ricianchan` function).

`dop = doppler.bigaussian(property1,value1,...)` creates a bi-Gaussian Doppler spectrum object with properties as specified by the property/value pairs. If you do not specify a value for a property, the property is assigned a default value.

`dop = doppler.bigaussian` creates a bi-Gaussian Doppler spectrum object with default properties. The constructed Doppler spectrum object is equivalent to a single Gaussian Doppler spectrum centered at zero frequency. The equivalent command with property/value pairs is:

```
dop = doppler.bigaussian('SigmaGaussian1', 1/sqrt(2),  
    'SigmaGaussian2', 1/sqrt(2),  
    'CenterFrequencyGaussian1', 0,  
    'CenterFrequencyGaussian2', 0,  
    'GainGaussian1', 0.5,  
    'GainGaussian2', 0.5)
```

Properties The bi-Gaussian Doppler spectrum object contains the following properties.

Property	Description
<code>SpectrumType</code>	Fixed value, 'BiGaussian'
<code>SigmaGaussian1</code>	Normalized standard deviation of first Gaussian function (real positive finite scalar value)

Property	Description
SigmaGaussian2	Normalized standard deviation of second Gaussian function (real positive finite scalar value)
CenterFreqGaussian1	Normalized center frequency of first Gaussian function (real scalar value between -1 and 1)
CenterFreqGaussian2	Normalized center frequency of second Gaussian function (real scalar value between -1 and 1)
GainGaussian1	Power gain of first Gaussian function (linear scale, real nonnegative finite scalar value)
GainGaussian2	Power gain of second Gaussian function (linear scale, real nonnegative finite scalar value)

All properties are writable except for the `SpectrumType` property.

The properties `SigmaGaussian1`, `SigmaGaussian2`, `GainGaussian1`, and `GainGaussian2` are normalized by the `MaxDopplerShift` property of the associated channel object.

Analytically, the normalized standard deviations of the first and

second Gaussian functions are determined as $\sigma_{G1, norm} = \sigma_{G1} / f_d$

and $\sigma_{G2, norm} = \sigma_{G2} / f_d$, respectively, where σ_{G1} and σ_{G2} are the standard deviations of the first and second Gaussian functions, and

f_d is the maximum Doppler shift, in hertz. Similarly, the normalized center frequencies of the first and second Gaussian functions are

determined as $f_{G1, norm} = f_{G1} / f_d$ and $f_{G2, norm} = f_{G2} / f_d$, respectively,

where f_{G1} and f_{G2} are the center frequencies of the first and second Gaussian functions. The properties `GainGaussian1` and `GainGaussian2`

Theory and Applications

correspond to the power gains C_{G1} and C_{G2} , respectively, of the two Gaussian functions.

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

doppler, doppler.ajakes, doppler.bell, doppler.flat, doppler.gaussian, doppler.jakes, doppler.rjakes, doppler.rounded, rayleighchan, ricianchan, stdchan, and “Fading Channels”

Purpose	Construct flat Doppler spectrum object
Syntax	<code>dop = doppler.flat</code>
Description	<code>dop = doppler.flat</code> creates a flat Doppler spectrum object that is to be used for the <code>DopplerSpectrum</code> property of a channel object (created with either the <code>rayleighchan</code> or the <code>ricianchan</code> function). The maximum Doppler shift of the flat Doppler spectrum object is specified by the <code>MaxDopplerShift</code> property of the channel object.
Properties	The flat Doppler spectrum object contains only one property, <code>SpectrumType</code> , which is read-only and has a fixed value of 'Flat'.
Theory and Applications	<p>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.</p> <p>The normalized flat Doppler power spectrum is given analytically by:</p> $S(f) = \frac{1}{2f_d}, f \leq f_d$ <p>where f_d is the maximum Doppler frequency.</p>
References	<p>[1] ANSI J-STD-008, <i>Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems</i>, March 1995.</p> <p>[2] Clarke, R. H., and Khoo, W. L., "3-D Mobile Radio Channel Statistics", <i>IEEE Trans. Veh. Technol.</i>, Vol. 46, No. 3, pp. 798–799, August 1997.</p>

doppler.flat

See Also

doppler, doppler.ajakes, doppler.bell, doppler.bigaussian,
doppler.gaussian, doppler.jakes, doppler.rjakes,
doppler.rounded, “Fading Channels”, rayleighchan, ricianchan,
and stdchan

Purpose Construct Gaussian Doppler spectrum object

Syntax
`dop = doppler.gaussian`
`dop = doppler.gaussian(sigmagaussian)`

Description The `doppler.gaussian` function creates a Gaussian Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.gaussian` creates a Gaussian Doppler spectrum object with a default standard deviation (normalized by the maximum Doppler

shift f_d , in Hz) $\sigma_{G,norm} = 1/\sqrt{2}$. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object.

Analytically, $\sigma_{G,norm} = \sigma_G / f_d = 1/\sqrt{2}$, where σ_G is the standard deviation of the Gaussian Doppler spectrum.

`dop = doppler.gaussian(sigmagaussian)` creates a Gaussian Doppler spectrum object with a normalized f_d (by the maximum Doppler shift f_d , in Hz) $\sigma_{G,norm}$ of value `sigmagaussian`.

Properties The Gaussian Doppler spectrum object contains the following properties.

Property	Description
<code>SpectrumType</code>	Fixed value, 'Gaussian'
<code>SigmaGaussian</code>	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:

$$\sigma_D = \sqrt{\frac{\int_{-\infty}^{\infty} f^2 S(f) df}{\int_{-\infty}^{\infty} S(f) df}}$$

Example

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 $\sigma_{G,\text{norm}} = 0.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = rayleighchan(1/1000,10);  
dop_gaussian = doppler.gaussian(0.5);  
chan.DopplerSpectrum = dop_gaussian;
```

References

- [1] ANSI J-STD-008, *Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems*, March 1995.
- [2] Bello, P. A., “Aeronautical channel characterizations,” *IEEE Trans. Commun.*, Vol. 21, pp. 548–563, May 1973.
- [3] Cox, D. C., “Delay Doppler characteristics of multipath propagation at 910 MHz in a suburban mobile radio environment,” *IEEE Transactions on Antennas and Propagation*, Vol. AP-20, No. 5, pp. 625–635, Sept. 1972.
- [4] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

doppler, doppler.ajakes, doppler.bell, doppler.bigaussian, doppler.flat, doppler.jakes, doppler.rjakes, doppler.rounded, “Fading Channels”, rayleighchan, ricianchan, and stdchan

Purpose Construct Jakes Doppler spectrum object

Syntax

Description `dop = doppler.jakes` creates a Jakes Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function). The maximum Doppler shift of the Jakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object. By default, channel objects are created with a Jakes Doppler spectrum.

Properties The Jakes Doppler spectrum object contains only one property, `SpectrumType`, which is read-only and has a fixed value of 'Jakes'.

Theory and Applications

The Jakes Doppler power spectrum model is actually due to Gans [2], who analyzed the Clarke-Gilbert model ([1], [3], and [5]). The Clarke-Gilbert model is also called the *classical model*.

The Jakes Doppler power spectrum applies to a mobile receiver. It derives from the following assumptions [6]:

- The radio waves propagate horizontally.
- At the mobile receiver, the angles of arrival of the radio waves are uniformly distributed over $[-\pi, \pi]$.
- At the mobile receiver, the antenna is omnidirectional (i.e., the antenna pattern is circular-symmetrical).

The normalized Jakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{1}{\pi f_d \sqrt{1 - (f/f_d)^2}}, \quad |f| \leq f_d$$

where f_d is the maximum Doppler frequency.

References

- [1] Clarke, R. H., “A Statistical Theory of Mobile-Radio Reception,” *Bell System Technical Journal*, Vol. 47, No. 6, pp. 957–1000, July-August 1968.
- [2] Gans, M. J., “A Power-Spectral Theory of Propagation in the Mobile-Radio Environment,” *IEEE Trans. Veh. Technol.*, Vol. VT-21, No. 1, pp. 27–38, Feb. 1972.
- [3] Gilbert, E. N., “Energy Reception for Mobile Radio,” *Bell System Technical Journal*, Vol. 44, No. 8, pp. 1779–1803, Oct. 1965.
- [4] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.
- [5] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.
- [6] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

doppler, doppler.ajakes, doppler.bell, doppler.bigaussian, doppler.flat, doppler.gaussian, doppler.rjakes, doppler.rounded, “Fading Channels”, rayleighchan, ricianchan, and stdchan

doppler.rjakes

Purpose Construct restricted Jakes Doppler spectrum object

Syntax
`dop = doppler.rjakes`
`dop = doppler.rjakes(freqminmaxrjakes)`

Description The `doppler.rjakes` function creates a restricted Jakes (RJakes) Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rjakes` creates a Doppler spectrum object equivalent to the Jakes Doppler spectrum. The maximum Doppler shift of the RJakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object.

`dop = doppler.rjakes(freqminmaxrjakes)`, where `freqminmaxrjakes` is a row vector of two finite real numbers between 0 and 1, creates a Jakes Doppler spectrum. This spectrum is nonzero only for normalized frequencies (by the maximum Doppler shift,

f_d , in Hertz), f_{norm} , such that $0 \leq f_{min,norm} \leq |f_{norm}| \leq f_{max,norm} \leq 1$,

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 $0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$.

The normalized RJakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{A_r}{\pi f_d \sqrt{1 - (f/f_d)^2}}, \quad 0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$$

where

$$A_r = \frac{1}{\pi \left[\sin^{-1} \left(\frac{f_{\max}}{f_d} \right) - \sin^{-1} \left(\frac{f_{\min}}{f_d} \right) \right]}$$

doppler.rjakes

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.

Example

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 $f_{\min, \text{norm}} = 0.14$ and maximum normalized Doppler shift $f_{\max, \text{norm}} = 0.9$.

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 $0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$, where $f_{\min} = f_{\min, \text{norm}} \times f_d = 1.4$ Hz and $f_{\max} = f_{\max, \text{norm}} \times f_d = 9$ Hz.

```
chan = rayleighchan(1/1000, 10);
dop_rjakes = doppler.rjakes([0.14 0.9]);
chan.DopplerSpectrum = dop_rjakes;
chan.DopplerSpectrum
```

The output is:

```
SpectrumType: 'RJakes'
FreqMinMaxRJakes: [0.1400 0.9000]
```

References

- [1] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.
- [2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.
- [3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler`, `doppler.ajakes`, `doppler.bell`, `doppler.bigaussian`, `doppler.flat`, `doppler.gaussian`, `doppler.jakes`, `doppler.rounded`, “Fading Channels”, `rayleighchan`, `ricianchan`, and `stdchan`

Purpose Construct rounded Doppler spectrum object

Syntax
`dop = doppler.rounded`
`dop = doppler.rounded(coeffrounded)`

Description The `doppler.rounded` function creates a rounded Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rounded` creates a rounded Doppler spectrum object with default polynomial coefficients $a_0 = 1$, $a_2 = -1.72$, $a_4 = 0.785$ (see “Theory and Applications” on page 2-199 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
<code>SpectrumType</code>	Fixed value, 'Rounded'
<code>CoeffRounded</code>	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:

$$S(f) = C_r \left[a_0 + a_2 \left(\frac{f}{f_d} \right)^2 + a_4 \left(\frac{f}{f_d} \right)^4 \right], |f| \leq f_d$$

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 $f = 0$.

Example

The following code creates a Rician channel object with a maximum Doppler shift of $f_d = 10$. It then creates a rounded Doppler spectrum object with polynomial coefficients $a_0 = 1.0$, $a_2 = -0.5$, $a_4 = 1.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = ricianchan(1/1000,10,1);
dop_rounded = doppler.rounded([1.0 -0.5 1.5]);
chan.DopplerSpectrum = dop_rounded;
```

References

[1] IEEE 802.16 Broadband Wireless Access Working Group, "Channel models for fixed wireless applications," *IEEE 802.16a-03/01*, 2003-06-27.

See Also

doppler, doppler.ajakes, doppler.bell, doppler.biggaussian, doppler.flat, doppler.gaussian, doppler.jakes, doppler.rjakes, “Fading Channels”, rayleighchan, ricianchan, and stdchan

Purpose Decode using differential pulse code modulation

Syntax
`sig = dpcmdeco(indx,codebook,predictor)`
`[sig,quanterror] = dpcmdeco(indx,codebook,predictor)`

Description `sig = dpcmdeco(indx,codebook,predictor)` implements differential pulse code demodulation to decode the vector `indx`. The vector `codebook` represents the predictive-error quantization codebook. The vector `predictor` specifies the predictive transfer function. If the transfer function has predictive order `M`, `predictor` has length `M+1` and an initial entry of 0. To decode correctly, use the same `codebook` and `predictor` in `dpcmenco` and `dpcmdeco`.

See “Representing Partitions”, “Representing 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`.

See Also `quantiz`, `dpcmopt`, `dpcmenco`, “Differential Pulse Code Modulation”

References [1] Kondo, A. M., *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.

Purpose	Encode using differential pulse code modulation
Syntax	<pre>indx = dpcmenco(sig,codebook,partition,predictor) [indx,quants] = dpcmenco(sig,codebook,partition,predictor)</pre>
Description	<p><code>indx = dpcmenco(sig,codebook,partition,predictor)</code> implements differential pulse code modulation to encode the vector <code>sig</code>. <code>partition</code> is a vector whose entries give the endpoints of the partition intervals. <code>codebook</code>, a vector whose length exceeds the length of <code>partition</code> by one, prescribes a value for each partition in the quantization. <code>predictor</code> specifies the predictive transfer function. If the transfer function has predictive order M, <code>predictor</code> has length $M+1$ and an initial entry of 0. The output vector <code>indx</code> is the quantization index.</p> <p>See “Differential Pulse Code Modulation” for more about the format of <code>predictor</code>. See “Representing Partitions”, “Representing Partitions”, or the reference page for <code>quantiz</code> in this chapter, for a description of the formats of <code>partition</code> and <code>codebook</code>.</p> <p><code>[indx,quants] = dpcmenco(sig,codebook,partition,predictor)</code> is the same as the syntax above, except that <code>quants</code> contains the quantization of <code>sig</code> based on the quantization parameters. <code>quants</code> is a vector of the same size as <code>sig</code>.</p>
	<hr/> <p>Note If <code>predictor</code> is an order-one transfer function, the modulation is called a <i>delta modulation</i>.</p> <hr/>
Examples	See “Example: DPCM Encoding and Decoding” and “Example: Comparing Optimized and Nonoptimized DPCM Parameters” for examples that use <code>dpcmenco</code> .
See Also	<code>quantiz</code> , <code>dpcmopt</code> , <code>dpcmdeco</code> , “Differential Pulse Code Modulation”
References	[1] Kondo, A. M., <i>Digital Speech</i> , Chichester, England, John Wiley & Sons, 1994.

dpcmopt

Purpose Optimize differential pulse code modulation parameters

Syntax

```
predictor = dpcmopt(training_set,ord)
[predictor,codebook,partition] = dpcmopt(training_set,ord,
    len)
[predictor,codebook,partition] = dpcmopt(training_set,ord,
    ini_cb)
```

Description `predictor = dpcmopt(training_set,ord)` returns a vector representing a predictive transfer function of order `ord` that is appropriate for the training data in the vector `training_set`. `predictor` is a row vector of length `ord+1`. See “Representing 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 “Representing Partitions”, “Representing Codebooks”, or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[predictor,codebook,partition] = dpcmopt(training_set,ord,ini_cb)` is the same as the first syntax, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `ini_cb`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `ini_cb`. The output `partition` is a vector whose length is one less than the length of `codebook`.

Examples

See “Example: Comparing Optimized and Nonoptimized DPCM Parameters” for an example that uses `dpcmopt`.

See Also

`dpcmenco`, `dpcmdeco`, `quantiz`, `lloyds`, “Differential Pulse Code Modulation”

dpskdemod

Purpose Differential phase shift keying demodulation

Syntax

```
z = dpskdemod(y,M)
z = dpskdemod(y,M,phaserot)
z = dpskdemod(y,M,phaserot,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.dpskdemod` instead.

`z = dpskdemod(y,M)` demodulates the complex envelope `y` of a DPSK modulated signal. `M` is the alphabet size and must be an integer. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

Note The first element of the output `z`, or the first row of `z`, if `z` is a matrix with multiple rows, represents an initial condition, because the differential algorithm compares two successive elements of the modulated signal.

`z = dpskdemod(y,M,phaserot)` specifies the phase rotation of the modulation in radians. In this case, the total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

`z = dpskdemod(y,M,phaserot,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples

The example below illustrates the fact that the first output symbol of a differential PSK demodulator is an initial condition rather than useful information.

```
M = 4; % Alphabet size
x = randint(1000,1,M); % Random message
y = dpskmod(x,M); % Modulate.
z = dpskdemod(y,M); % Demodulate.
% Check whether the demodulator recovered the message.
s1 = symerr(x,z) % Expect one symbol error, namely, the first symbol.
s2 = symerr(x(2:end),z(2:end)) % Ignoring 1st symbol, expect no errors.
```

The output is below.

```
s1 =
```

```
    1
```

```
s2 =
```

```
    0
```

For another example that uses this function, see “Example: Curve Fitting for an Error Rate Plot”.

See Also

dpskmod, pskdemod, pskmod, “Modulation”

dpskmod

Purpose Differential phase shift keying modulation

Syntax

```
y = dpskmod(x,M)
y = dpskmod(x,M,phaserot)
y = dpskmod(x,M,phaserot,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.dpskmod` instead.

`y = dpskmod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using differential phase shift keying modulation. `M` is the alphabet size and must be an integer. The message signal must consist of integers between 0 and `M-1`. If `x` is a matrix with multiple rows and columns, the function processes the columns independently.

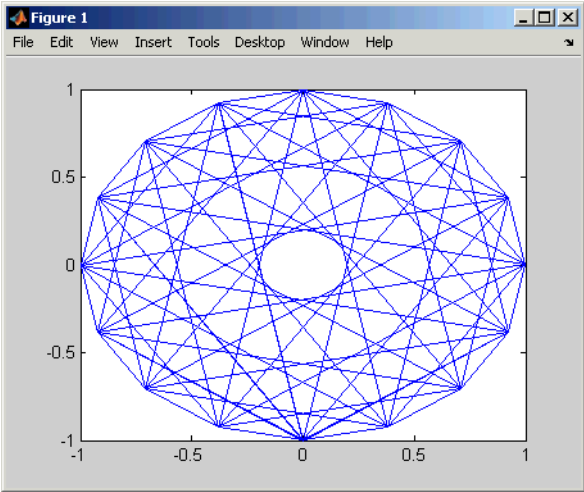
`y = dpskmod(x,M,phaserot)` specifies the phase rotation of the modulation in radians. In this case, the total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

`y = dpskmod(x,M,phaserot,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples

The example below plots the output of the `dpskmod` function. The image shows the possible transitions from each symbol in the DPSK signal constellation to the next symbol.

```
M = 4; % Use DQPSK in this example, so M is 4.
x = randint(500,1,M,13); % Random data
y = dpskmod(x,M,pi/8); % Modulate using a nonzero initial phase.
plot(y) % Plot all points, using lines to connect them.
```



For another example that uses this function, see “Example: Curve Fitting for an Error Rate Plot”.

See Also

dpskdemod, pskmod, pskdemod, “Modulation”

dvbs2ldpc

Purpose Low-density parity-check codes from DVB-S.2 standard

Syntax `H = dvbs2ldpc(r)`

Description `H = dvbs2ldpc(r)` returns the parity-check matrix of the LDPC code with code rate `r` from the DVB-S.2 standard. `H` is a sparse logical matrix. Possible values for `r` are 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, and 9/10. The block length of the code is 64800.

The default parity-check matrix (32400-by-64800) corresponds to an irregular LDPC code with the structure shown in the following table.

Row	Number of 1s Per Row
1	6
2 to 32400	7

Column	Number of 1s Per Column
1 to 12960	8
12961 to 32400	3

Columns 32401 to 64800 form a lower triangular matrix. Only the elements on its main diagonal and the subdiagonal immediately below are 1s. This LDPC code is used in conjunction with a BCH code in the Digital Video Broadcasting standard DVB-S.2 to achieve a packet error rate below 10^{-7} at about 0.7 dB to 1 dB from the Shannon limit.

Example

```
H = dvbs2ldpc(3/5);
spy(H); % Visualize the location of nonzero elements in H.
henc = fec.ldpcenc(H);
hdec = fec.ldpcdec(H);
```

See Also `fec.ldpcdec`, `fec.ldpcenc`, `spy`

Purpose Block encoder

Syntax

```
code = encode(msg,n,k,'linear/fmt',genmat)
code = encode(msg,n,k,'cyclic/fmt',genpoly)
code = encode(msg,n,k,'hamming/fmt',prim_poly)
code = encode(msg,n,k)
[code,added] = encode(...)
```

Optional Inputs

Input	Default Value
<i>fmt</i>	binary
genpoly	cyclpoly(n,k)
prim_poly	gfprindf(n-k)

Description For All Syntaxes

The encode function encodes messages using one of the following error-correction coding methods:

- Linear block
- Cyclic
- Hamming

For all of these methods, the codeword length is *n* and the message length is *k*.

msg, which represents the messages, can have one of several formats. The table below shows which formats are allowed for *msg*, how the argument *fmt* should reflect the format of *msg*, and how the format of the output *code* depends on these choices. The examples in the table are for *k* = 4. If *fmt* is not specified as input, its default value is *binary*.

encode

Note If 2^n or 2^k is large, use the default binary format instead of the decimal format. This is because the function uses a binary format internally, while the roundoff error associated with converting many bits to large decimal numbers and back might be substantial.

Information Formats

Dimension of msg	Value of <i>fmt</i> Argument	Dimension of code
Binary column or row vector	binary	Binary column or row vector
Example: msg = [0 1 1 0, 0 1 0 1, 1 0 0 1].'		
Binary matrix with k columns	binary	Binary matrix with n columns
Example: msg = [0 1 1 0; 0 1 0 1; 1 0 0 1]		
Column or row vector of integers in the range $[0, 2^k-1]$	decimal	Column or row vector of integers in the range $[0, 2^n-1]$
Example: msg = [6, 10, 9].'		

For Specific Syntaxes

`code = encode(msg,n,k,'linear/fmt',genmat)` encodes `msg` using `genmat` as the generator matrix for the linear block encoding method. `genmat`, a k -by- n matrix, is required as input.

`code = encode(msg,n,k,'cyclic/fmt',genpoly)` encodes `msg` and creates a systematic cyclic code. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an $[n,k]$ cyclic code must have degree $n-k$ and must divide x^n-1 .

`code = encode(msg,n,k,'hamming/fmt',prim_poly)` encodes `msg` using the Hamming encoding method. For this syntax, `n` must have the form 2^m-1 for some integer `m` greater than or equal to 3, and `k` must equal `n-m`. `prim_poly` is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive polynomial for $GF(2^m)$ that is used in the encoding process. The default value of `prim_poly` is the default primitive polynomial `gfprimdf(m)`.

`code = encode(msg,n,k)` is the same as `code = encode(msg,n,k,'hamming/binary')`.

`[code,added] = encode(...)` returns the additional variable `added`. `added` is the number of zeros that were placed at the end of the message matrix before encoding in order for the matrix to have the appropriate shape. “Appropriate” depends on `n`, `k`, the shape of `msg`, and the encoding method.

Examples

The example below illustrates the three different information formats (binary vector, binary matrix, and decimal vector) for Hamming code. The three messages have identical content in different formats; as a result, the three codes that `encode` creates have identical content in correspondingly different formats.

```
m = 4; n = 2^m-1; % Codeword length = 15
k = 11; % Message length

% Create 100 messages, k bits each.
msg1 = randint(100*k,1,[0,1]); % As a column vector
msg2 = vec2mat(msg1,k); % As a k-column matrix
msg3 = bi2de(msg2)'; % As a row vector of decimal integers

% Create 100 codewords, n bits each.
code1 = encode(msg1,n,k,'hamming/binary');
code2 = encode(msg2,n,k,'hamming/binary');
code3 = encode(msg3,n,k,'hamming/decimal');
if ( vec2mat(code1,n)==code2 & de2bi(code3',n)==code2 )
    disp('All three formats produced the same content.')
end
```

The output is

All three formats produced the same content.

The next example creates a cyclic code, adds noise, and then decodes the noisy code. It uses the `decode` function.

```
n = 3; k = 2; % A (3,2) cyclic code
msg = randint(100,k,[0,1]); % 100 messages, k bits each
code = encode(msg,n,k,'cyclic/binary');
% Add noise.
noisycode = rem(code + randerr(100,n,[0 1;.7 .3]), 2);
newmsg = decode(noisycode,n,k,'cyclic'); % Try to decode.
% Compute error rate for decoding the noisy code.
[number,ratio] = biterr(newmsg,msg);
disp(['The bit error rate is ',num2str(ratio)])
```

The output is below. Your error rate results might vary because the noise is random.

The bit error rate is 0.08

The next example encodes the same message using Hamming and cyclic methods. This example also creates Hamming code with the `'linear'` option of the `encode` command. It then decodes each code and recovers the original message.

```
n = 7; % Codeword length
k = 4; % Message length
m = log2(n+1); % Express n as 2^m-1.
msg = randint(100,1,[0,2^k-1]); % Column of decimal integers

% Create various codes.
codehamming = encode(msg,n,k,'hamming/decimal');
[parmat,genmat] = hamngen(m);
codehamming2 = encode(msg,n,k,'linear/decimal',genmat);
if codehamming==codehamming2
    disp('The ''linear'' method can create Hamming code.')
```

```

end
codecyclic = encode(msg,n,k,'cyclic/decimal');

% Decode to recover the original message.
decodedhamming = decode(codehamming,n,k,'hamming/decimal');
decodedcyclic = decode(codecyclic,n,k,'cyclic/decimal');
if (decodedhamming==msg & decodedcyclic==msg)
    disp('All decoding worked flawlessly in this noiseless world.')
end

```

The output is

```

The 'linear' method can create Hamming code.
All decoding worked flawlessly in this noiseless world.

```

Algorithm

Depending on the encoding method, `encode` relies on such lower-level functions as `hammgen` and `cyclgen`.

See Also

`decode`, `cyclpoly`, `cyclgen`, `hammgen`, “Block Coding”

equalize

Purpose Equalize signal using equalizer object

Syntax

```
y = equalize(eqobj,x)
y = equalize(eqobj,x,trainSIG)
[y,yd] = equalize(...)
[y,yd,e] = equalize(...)
```

Description `y = equalize(eqobj,x)` processes the baseband signal vector `x` with equalizer object `eqobj` and returns the equalized signal vector `y`. At the end of the process, `eqobj` contains updated state information such as equalizer weight values and input buffer values. To construct `eqobj`, use the `lineareq` or `dfe` function, as described in “Using Adaptive Equalizer Functions and Objects”. 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,trainseq)` 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 `trainseq` must be less than or equal to $\text{length}(x) - (\text{eqobj.RefTap} - 1) / \text{eqobj.nSampPerSym}$.

`[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 “Equalizing Using a Training Sequence”, “Example: Equalizing Multiple Times, Varying the Mode”, and “Example: Adaptive Equalization Within a Loop”.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `dfe`, “Equalizers”

eyediagram

Purpose Generate eye diagram

Syntax `eyediagram(x,n)`
 `eyediagram(x,n,period)`
 `eyediagram(x,n,period,offset)`
 `eyediagram(x,n,period,offset,plotstring)`
 `eyediagram(x,n,period,offset,plotstring,h)`
 `h = eyediagram(...)`

Description **Warning**

This is an obsolete function and may be removed in the future. Use the object `commscope.eyediagram` instead.

`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 $-\text{period}/2$ and $\text{period}/2$.

`eyediagram(x,n,period,offset)` is the same as the syntax above, except that the function assumes that the $(\text{offset}+1)$ st value of the signal, and every n th value thereafter, occur at times that are integer multiples of `period`. The variable `offset` must be a nonnegative integer between 0 and $n-1$.

`eyediagram(x,n,period,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function. The default string is 'b-', which produces a blue solid line.

`eyediagram(x,n,period,offset,plotstring,h)` is the same as the syntax above, except that the eye diagram is in the figure whose handle is `h`, rather than in a new figure. `h` must be a handle to a figure that `eyediagram` previously generated.

Note You cannot use `hold on` to plot multiple signals in the same figure.

`h = eyediagram(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the eye diagram.

Examples

For an online demonstration, type `showdemo scattereyedemo`.

See Also

`scatterplot`, `plot`, `scattereyedemo`, “Eye Diagrams”

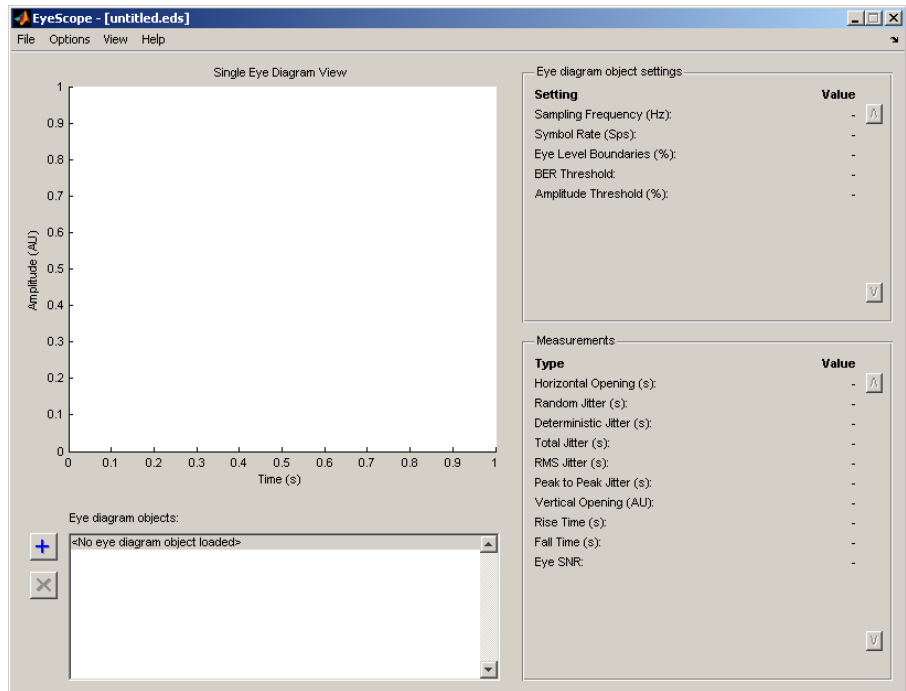
Purpose Launch eye diagram scope for eye diagram object H

Syntax `eyescope`

Description Use EyeScope to examine the data in an eye diagram object. EyeScope shows both the eye diagram plot and measurement results in a unified, graphical environment, providing a very efficient means for viewing eye diagram data. For more information, refer to the EyeScope chapter in the Communications Toolbox User's Guide.

Starting EyeScope To start EyeScope from the MATLAB® command line, type:
`eyescope`

The following figure shows an EyeScope that does not have an eye diagram object loaded in its memory.



Alternatively, you can start EyeScope so it displays an eye diagram object. To start EyeScope so it displays an eye diagram object, type the following at the MATLAB command line:

```
eyescope(h)
```

Note *h* is a handle to an eye diagram object in the workspace.

The EyeScope Environment

- “EyeScope Menu Bar” on page 2-222
- “Eye Diagram Object Plot and Plot Controls” on page 2-222
- “Eye Diagram Object Settings Panel” on page 2-224

- “Measurements” on page 2-225

EyeScope Menu Bar

EyeScope Menu Bar

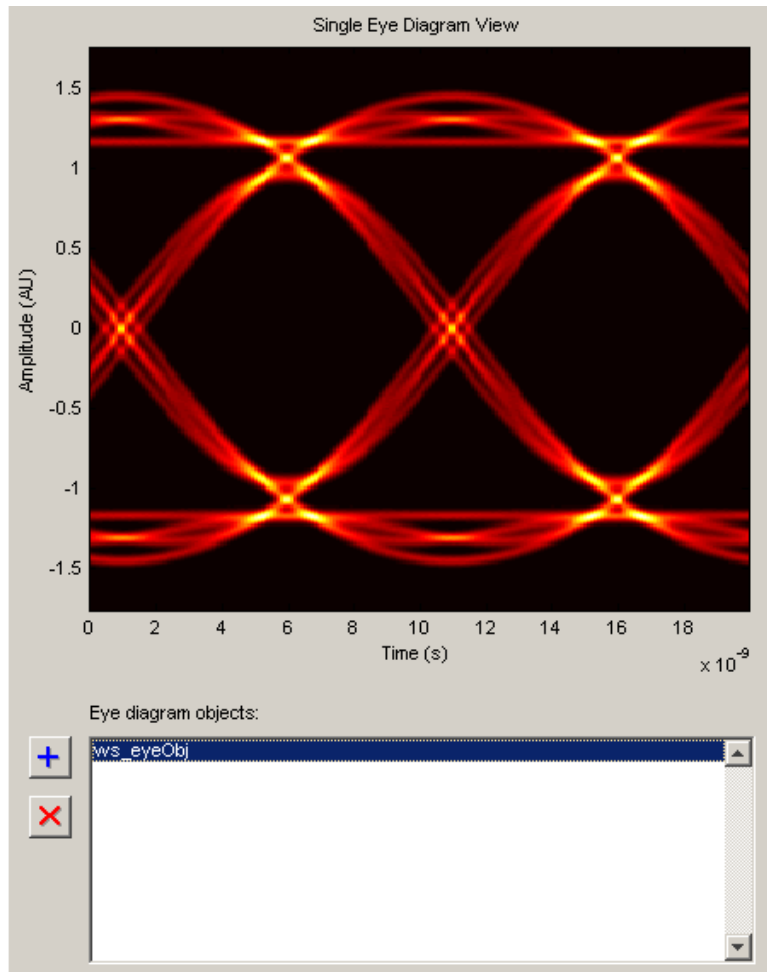
The EyeScope menu bar is comprised of four menus: File, Options, View, and Help.



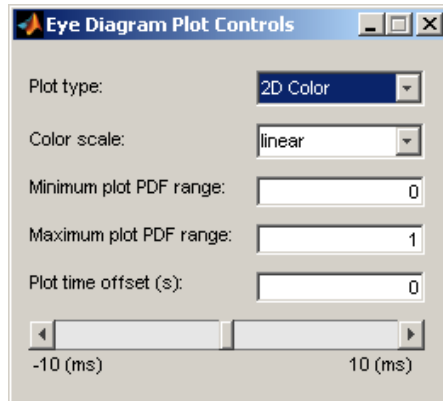
- Use the **File** menu to control the session management functions, import an eye diagram object into EyeScope, and export an eye diagram plot.
- Use the **Options** menu to setup the eye diagram scope by selecting which eye diagram settings and measurements EyeScope displays.
- Use the **View** menu to toggle between Single eye diagram view or Compare measurement results view, and to add or modify a legend for the eye diagram plot.
- The **Help** menu is used to access help pertaining to the eye diagram object and EyeScope.

Eye Diagram Object Plot and Plot Controls

The Eye diagram object plot is the region of the GUI where the eye diagram plot appears.



Eye diagram plot controls are user-configurable settings that specify plot type, color scale, minimum and maximum plot PDF range, and plot time offset for the eye diagram being analyzed. To access the EyeScope plot controls **Options > Eye Diagram Plot Controls**



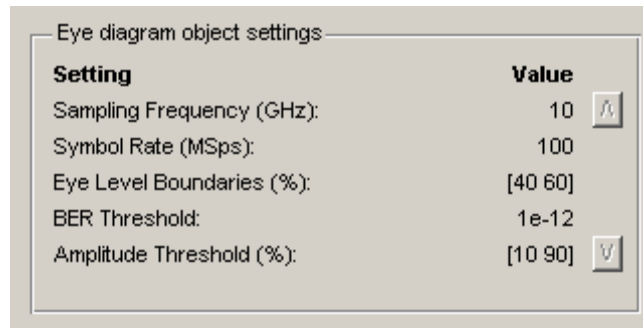
Note The value for the **Plot time offset** parameter can either be entered directly into the text box or set using the slide bar control.

For more information pertaining to the eye diagram properties, refer to the `commscope.eyediagram` reference page.

Eye Diagram Object Settings Panel

The eye diagram object settings panel displays the eye diagram object settings. The default EyeScope configuration displays the following eye diagram object settings:

- Sampling frequency
- Symbol rate
- Eye level boundaries
- BER threshold
- Amplitude threshold

The image shows a screenshot of the 'Eye diagram object settings' panel in EyeScope. It contains a table with two columns: 'Setting' and 'Value'. The settings listed are Sampling Frequency (GHz) at 10, Symbol Rate (MSps) at 100, Eye Level Boundaries (%) at [40 60], BER Threshold at 1e-12, and Amplitude Threshold (%) at [10 90]. There are scroll buttons next to the values for Sampling Frequency and Amplitude Threshold.

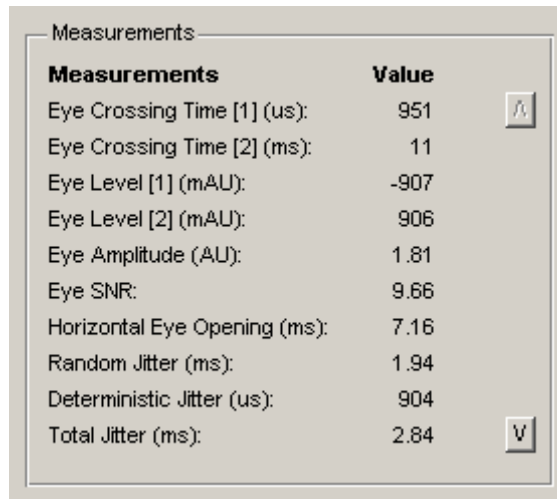
Setting	Value
Sampling Frequency (GHz):	10
Symbol Rate (MSps):	100
Eye Level Boundaries (%):	[40 60]
BER Threshold:	1e-12
Amplitude Threshold (%):	[10 90]

To specify which eye diagram object settings display in EyeScope, refer to “Selecting Which Eye Diagram Object Settings To Display” on page 2-232. 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



The screenshot shows a dialog box titled "Measurements" with a table of eye diagram measurements. The table has two columns: "Measurements" and "Value". There are also two scroll buttons, one at the top right and one at the bottom right.

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 2-233. 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 2-227
- “Starting a new Session” on page 2-227
- “Opening a Session” on page 2-227
- “Saving a Session” on page 2-228
- “Importing an Eye Diagram Object” on page 2-229
- “Printing to a Figure” on page 2-231
- “Selecting Which Eye Diagram Object Settings To Display” on page 2-232
- “Selecting Which Eye Diagram Measurements To Display” on page 2-233

Starting EyeScope with an Argument

You can start EyeScope so it is displaying an eye diagram object. To start EyeScope so it is displaying an eye diagram object, type the following at the MATLAB command line:

```
eyescope(h)
```

Note *h* is a handle to an eye diagram object presently in the workspace.

Starting a new Session

Starting a new session purges EyeScope memory, returning EyeScope to an empty plot display. If changes have been made to an open session and you start a new session, you will be prompted to save the open session.

Opening a Session

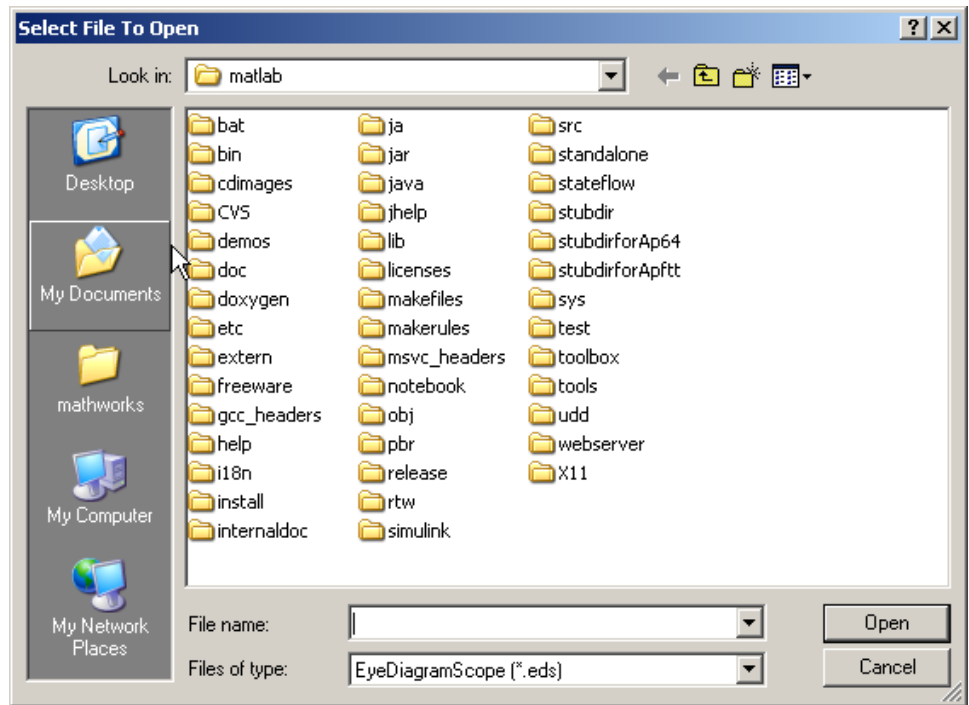
To open session, choose the file name and location of the session file. The file extensions for a session file is .eds, which stands for eye diagram scope. If changes have been made to a session that is presently open and you try to open up a new session, you will be prompted to save the session that is presently open before the new session can start.

To open a session:

1

Click **File > Open Session**.

The Select File To Open Window appears.



2

Navigate to the EyeScope session file you want, and click **Open**.

Saving a Session

The Save Session selection saves the current session, updating the session file. A session file includes the eye diagram object, eyescope options, and plot control selections.

If you attempt to save a session that you have not previously saved, EyeScope will prompt you for a file name and location. Otherwise, the session is saved to the previously selected file.

To save a session, follow these steps:

1

Click **File > Save Session**.

2

Navigate to the folder where you want to save the EyeScope session file and click **Save**.

Importing an Eye Diagram Object

The **Import** menu selection imports an eye diagram object from either the workspace or a MAT-file to EyeScope. The imported variable name will be reconstructed to reflect the origin of the eye diagram object, as follows:

- If an object is imported from the workspace, the variable name will be *ws_object name*, where *object name* is the name of the original variable.
- If the object is imported from a MATLAB file, then the file name (without the path) precedes the object name.

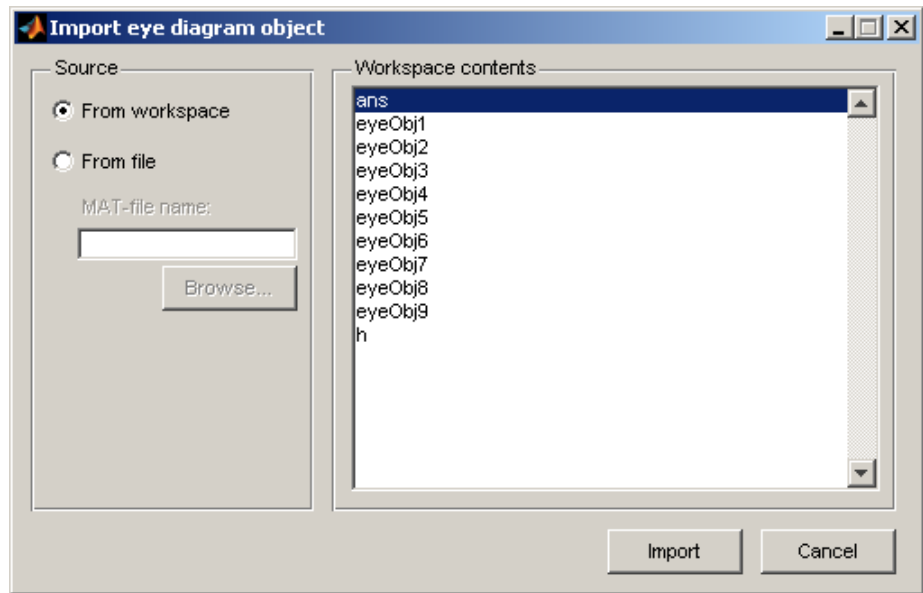
Importing an object creates a copy of the object, using the naming convention previously described. EyeScope displays the object's contents as configured when the object was imported. EyeScope does not track any object changes made in the workspace (or to the MATLAB file) from which the object was imported.

To import an eye diagram object:

1

Click **File > Import Eye Diagram Object**

The Import eye diagram object window appears.



The contents panel of the of the Import eye diagram object window displays all eye diagram objects available in the source location.

2

From the Import eye diagram object window, select the source for the object being imported.

- Select **From workspace** to import an eye diagram object directly from the workspace.
- Select **From File** to choose an eye diagram object file that was previously saved and click **Browse** to select the file to be loaded.

3

Click **Import**.

Printing to a Figure

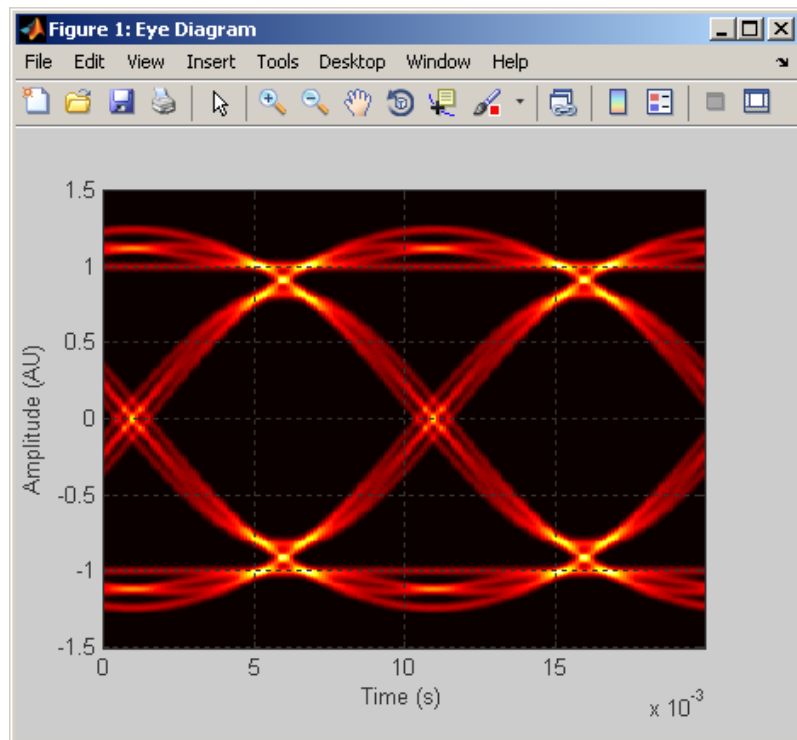
EyeScope allows you to print an eye diagram plot to a separate MATLAB figure window. From the MATLAB figure window, along with other tasks, you can print, zoom, or edit the plot.

To export an eye diagram figure:

1

Click **File > Print to Figure**

The MATLAB figure window, containing the exported image, appears.



Selecting Which Eye Diagram Object Settings To Display

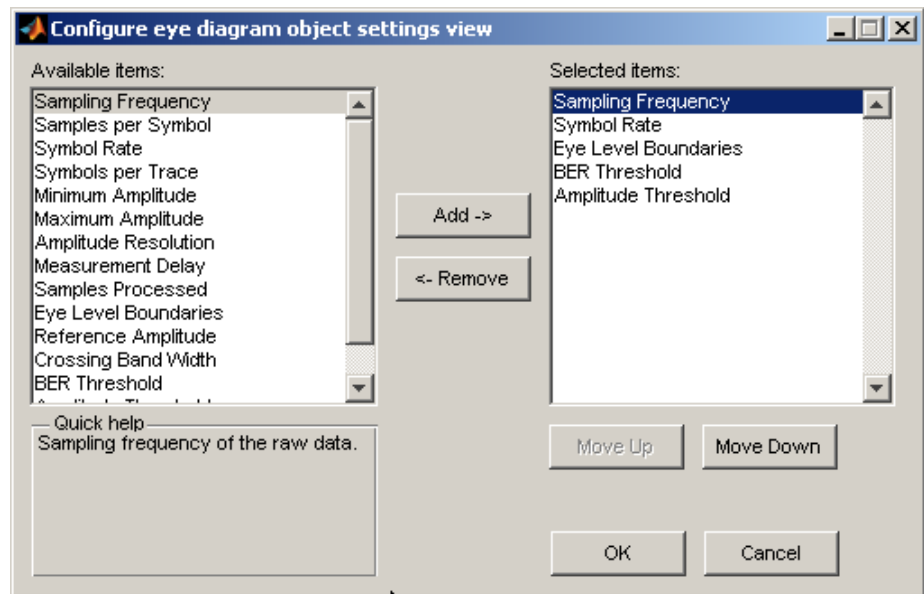
The **Eye Diagram Object Settings View** allows you to select which object settings display in the eye diagram object settings panel. You make your selections in the Configure eye diagram object settings view window, where a shuttle control allows you to add, remove, or reorder the settings you are displaying.

To add an eye diagram object setting:

1

Click **Options > Eye Diagram Object Settings View**

The Configure eye diagram object settings view window appears.



2

Locate any items to be added in the list of **Available items**, and left-click to select.

Note To select multiple items, you can either press and hold the <Shift> key and left-click or press and hold the <Ctrl> key and left-click.

When you select an item, the **Quick help** panel displays information about the item. If you select multiple items, **Quick help** displays information pertaining to the last item you select.

3

Click **Add**.

Note Using the **Move Up** or **Move Down** buttons, you can change the order in which the eye diagrams settings you select appear.

4

Click **OK** .

Selecting Which Eye Diagram Measurements To Display

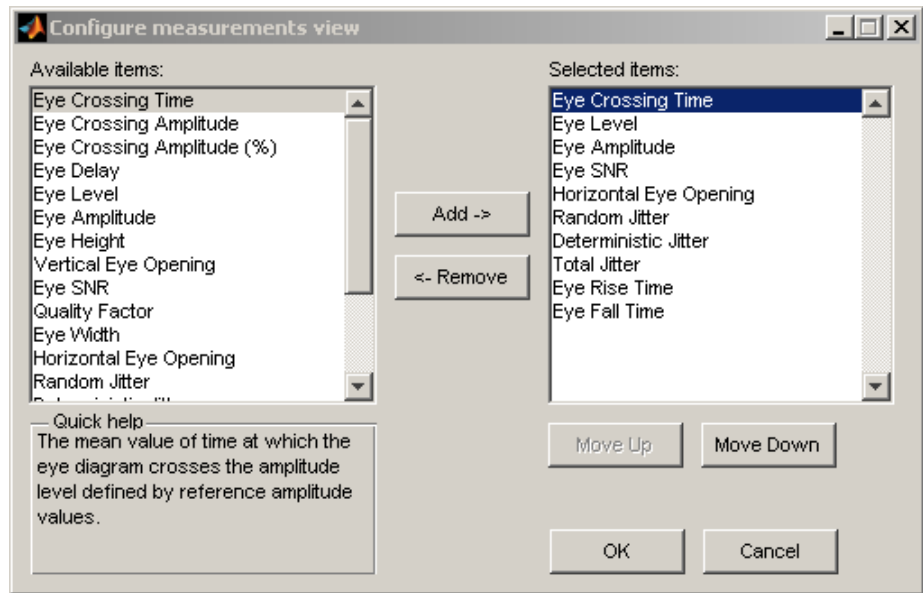
You can modify the contents of the measurement panel by selecting which eye diagram measurements display in the eye diagram object settings panel. You make your selections in the Configure measurements view window, where a shuttle control allows you to add, remove, or reorder the settings you are including.

Adding An Eye Diagram Measurement Setting

1

Click **Options > Measurements View**

The Configure measurements window appears.



2

Locate any items to be added in the list of **Available items**, and left-click to select.

Note To select multiple items, you can either press and hold the <Shift> key and left-click or press and hold the <Ctrl> key and left-click.

When you select an item, the **Quick help** panel displays information about the item. If you select multiple items, **Quick help** displays information pertaining to the last item you select.

3

Click **Add**.

Note Using the **Move Up** or **Move Down** buttons, you can change the order in which the eye diagrams settings you select appear.

4

Click **OK** .

fec.ldpcdec

Purpose Construct LDPC decoder object

Syntax
`l = fec.ldpcdec(H)`
`l = fec.ldpcdec`
`decoded = decode(l, llr)`

Description The `fec.ldpcdec` function creates a low-density parity-check (LDPC) decoder object that you can use with the `decode` method to decode output from a demodulator.

`l = fec.ldpcdec(H)` constructs an LDPC decoder object `l` for a binary systematic LDPC code with a parity-check matrix `H`.

`H` must be a sparse zero-one matrix. n and $n-k$ are the number of columns and the number of rows, respectively, in `H`.

`l = fec.ldpcdec` constructs an LDPC decoder object `l` with a default parity-check matrix (32400-by-64800). For more information, see `dvbs2ldpc`

Properties

The following table describes the properties of an LDPC decoder object.

`ParityCheckMatrix` specifies the LDPC code. `DecisionType`, `OutputFormat`, `DoParityChecks`, and `NumIterations` specify settings for the decoding operation. All other properties are read-only.

Property	Description
<code>ParityCheckMatrix</code>	Parity-check matrix of the LDPC code. Stored as a sparse logical 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 $2^{31}-1$
<code>BlockLength</code>	Total number of bits in a codeword, n .
<code>NumInfoBits</code>	Number of information bits in a codeword, k .

Property	Description
NumParityBits	Number of parity bits in a codeword, $n-k$.
DecisionType	Value can be 'Hard decision' (default) or 'Soft decision'.
OutputFormat	Value can be 'Information part' (default) or 'Whole codeword'.
DoParityChecks	Determines whether the parity checks should be verified after each iteration, and whether the decoder should stop iterating if all parity checks are satisfied. Value can be 'Yes' or 'No' (default).
NumIterations	Number of iterations to be performed for decoding one codeword. Default value is 50.
ActualNumIterations	Actual number of iterations executed for the last codeword. Initial value is [].
FinalParityChecks	$(n-k)$ -by-1 vector. 1s indicate the parity checks that are not satisfied when the decoder stops. Initial value is [].

When `ParityCheckMatrix` is changed, the properties `BlockLength`, `NumInfoBits`, and `NumParityBits` are updated.

Setting `DoParityChecks` to 'Yes' can speed up decoding in some situations by reducing the number of iterations executed.

Decoding Method

This object has a method `decode` that is used to decode signals.

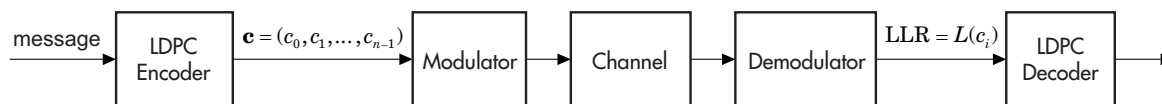
`decoded = decode(l, llr)` decodes an LDPC code using the message-passing algorithm, where `l` is an LDPC decoder object and `llr` is a 1-by-BlockLength vector.

The results returned in `decoded` depends on the parameters of the LDPC decoder object.

If the property...	is set to...	then decoded is...
DecisionType	'Hard decision'	The decoded bits. See "Decoding Algorithm" on page 2-238.
DecisionType	'Soft decision'	The log-likelihood ratios for the decoded bits.
OutputFormat	'Information part'	A 1-by-NumInfoBits vector.
OutputFormat	'Whole codeword'	A 1-by-BlockLength vector.

This method uses the properties `DecisionType`, `OutputFormat`, `NumIterations`, and `DoParityChecks`, and updates the values for `FinalParityChecks`, and `ActualNumIterations`.

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

$$L(r_{ji}) = 2 \operatorname{atanh} \left(\prod_{i' \in V_j \setminus i} \tanh \left(\frac{1}{2} L(q_{i'j}) \right) \right)$$

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})$$

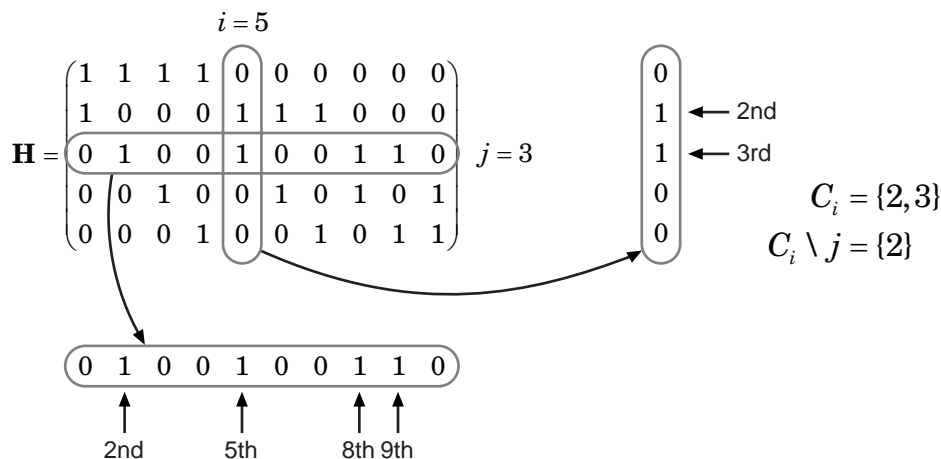
$$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i})$$

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 \mathbf{H} :

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

For $i = 5$ and $j = 3$, the index sets would be



$$V_j = \{2,5,8,9\}$$

$$V_j \setminus i = \{2,8,9\}$$

At the end of each iteration, $L(Q_i)$ provides an updated estimate of the *a posteriori* log-likelihood ratio for the transmitted bit c_i .

The soft-decision output for c_i is $L(Q_i)$. The hard-decision output for c_i is 1 if $L(Q_i) < 0$, and 0 otherwise.

If the property `DoParityCheck` is set to 'no', the algorithm iterates as many times as specified by `NumIterations`.

If the property `DoParityCheck` is set to 'yes', then at the end of each iteration the algorithm verifies the parity check equation ($\mathbf{Hc}^T = 0$) 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.

Usage Example

This example demonstrates the use of this object.

```
enc = fec.ldpcenc; % Construct a default LDPC encoder object

% Construct a companion LDPC decoder object
dec = fec.ldpcdec;
dec.DecisionType = 'Hard decision';
dec.OutputFormat = 'Information part';
dec.NumIterations = 50;
% Stop if all parity-checks are satisfied
dec.DoParityChecks = 'Yes';

% Generate and encode a random binary message
msg = randint(1,enc.NumInfoBits,2);
codeword = encode(enc,msg);

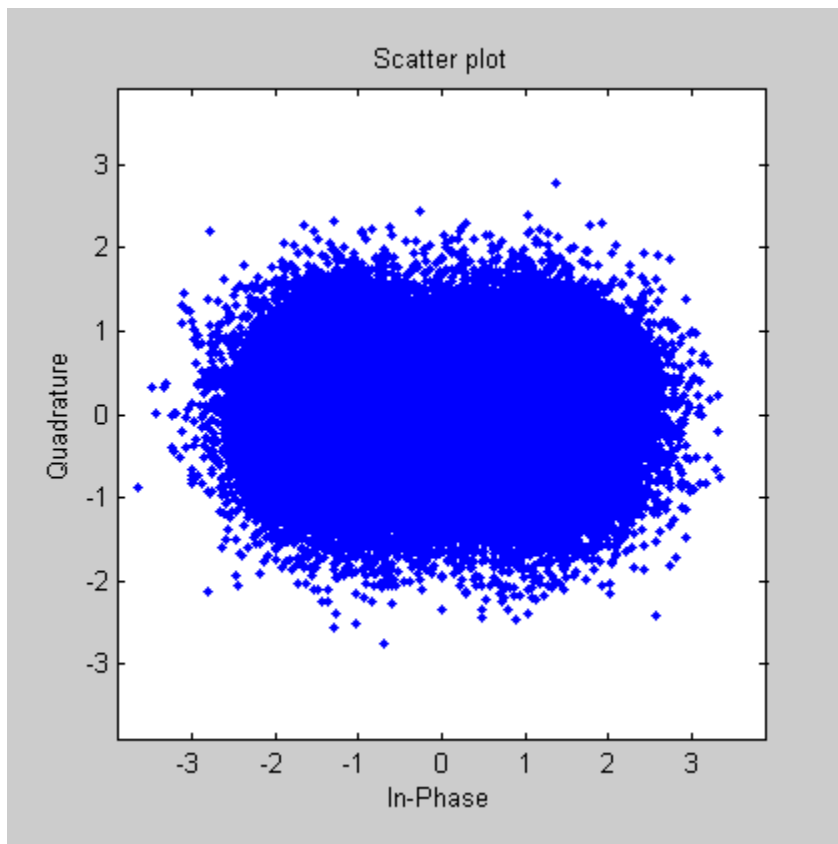
% Construct a BPSK modulator object
modObj = modem.pskmod('M',2,'InputType','Bit');

% Modulate the signal (map bit 0 to 1 + 0i, bit 1 to -1 + 0i)
modulatedsig = modulate(modObj, codeword);

% Noise parameters
SNRdB = 1;
sigma = sqrt(10^(-SNRdB/10));

% Transmit signal through AWGN channel
receivedsig = awgn(modulatedsig, SNRdB, 0); ...
    % Signal power = 0 dBW

% Visualize received signal
scatterplot(receivedsig)
```



```
% Construct a BPSK demodulator object to compute
% log-likelihood ratios
demodObj = modem.pskdemod(modObj,'DecisionType','LLR', ...
    'NoiseVariance',sigma^2);

% Compute log-likelihood ratios (AWGN channel)
llr = demodulate(demodObj, receivedsig);

% Decode received signal
decodedmsg = decode(dec, llr);
```



```
% Actual number of iterations executed
disp(['Number of iterations executed = ' ...
      num2str(dec.ActualNumIterations)]);
% Number of parity-checks violated
disp(['Number of parity-checks violated = ' ...
      num2str(sum(dec.FinalParityChecks)]);
% Compare with original message
disp(['Number of bits incorrectly decoded = ' ...
      num2str(nnz(decodedmsg-msg)]);
```

Example with a Parity-Check Matrix

This example demonstrates the construction of an LDPC decoder object with a parity-check matrix.

```
i = [1 3 2 4 1 2 3 3 4]; % row indices of 1s
j = [1 1 2 2 3 4 4 5 6]; % column indices of 1s
H = sparse(i,j,ones(length(i),1)); % parity-check matrix H
l = fec.ldpcdec(H);
```

References

[1] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

See Also

dvbs2ldpc, fec.ldpcenc, modem

fec.ldpcenc

Purpose Construct LDPC encoder object

Syntax

```
l = fec.ldpcenc(H)
l = fec.ldpcenc
codeword = encode(l, msg)
```

Description The `fec.ldpcenc` function creates a low-density parity-check (LDPC) encoder object that you can use with the `encode` method to encode a signal.

`l = fec.ldpcenc(H)` constructs an LDPC encoder object `l` for a binary systematic LDPC code with a parity-check matrix `H`.

`H` must be a sparse zero-one matrix. n and $n-k$ are the number of columns and the number of rows, respectively, in `H`. The last $n-k$ columns in `H` must be an invertible matrix in $GF(2)$.

`l = fec.ldpcenc` constructs an LDPC encoder object `l` with a default parity-check matrix (32400-by-64800). For more information, see `dvbs2ldpc`.

Properties

The following table describes the properties of an LDPC encoder object. Only `ParityCheckMatrix` is writable. All other properties are derived from it.

Property	Description
<code>ParityCheckMatrix</code>	Parity-check matrix of the LDPC code. Stored as a sparse logical matrix with
<code>BlockLength</code>	Total number of bits in a codeword, n .
<code>NumInfoBits</code>	Number of information bits in a codeword, k .

Property	Description
NumParityBits	Number of parity bits in a codeword, $n-k$.
EncodingAlgorithm	Method for solving the parity-check equation to compute the parity bits using the information bits. Set to 'Forward Substitution' if the last $n-k$ columns in H are a lower triangular matrix, 'Backward Substitution' if the last $n-k$ columns in H are an upper triangular matrix, and 'Matrix Inverse' in all other situations.

LDPC Encoding Method

This object has a method `encode` that is used to encode signals.

`codeword = encode(l, msg)` encodes `msg` using the LDPC code specified by the LDPC encoder object `l`. `msg` must be a binary 1-by-NumInfoBits vector.

`codeword` is a binary 1-by-BlockLength vector. The first NumInfoBits bits are the information bits (`msg`) and the last NumParityBits bits are the parity bits. The modulo-2 matrix product of `ParityCheckMatrix` and `codeword` is a zero vector.

$$\mathbf{H}\mathbf{c}^T = 0$$

Usage Example

This example demonstrates the use of this object.

```
% Construct a default LDPC encoder object
l = fec.ldpcenc;

% Generate a random binary message
msg = randint(1,l.NumInfoBits,2);

% Encode the message
```

```
codeword = encode(l, msg);

% Verify the parity checks (which should be a zero vector)
paritychecks = mod(l.ParityCheckMatrix * codeword', 2);
```

Example with a Parity-Check Matrix

This example demonstrates the construction of an LDPC encoder object with a parity-check matrix.

```
i = [1 3 2 4 1 2 3 3 4]; % row indices of 1s
j = [1 1 2 2 3 4 4 5 6]; % column indices of 1s
H = sparse(i,j,ones(length(i),1)); % parity-check matrix H
l = fec.ldpcenc(H);
```

References

[1] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

See Also

dvbs2ldpc, fec.ldpcdec, modem

Purpose Construct BCH decoder object

Syntax

```
h = fec.bchdec
h = fec.bchdec(N,K)
h = fec.bchdec(property1, value, ...)
h = fec.bchdec(bchenc_object)
```

Description

`h = fec.bchdec` constructs a BCH decoder with default properties. It is equivalent to: `dec = fec.bchdec(7,4)`

`h = fec.bchdec(N,K)` constructs an (N,K) BCH decoder object `dec`.

`h = fec.bchdec(property1, value1, ...)` constructs a BCH decoder object `dec` with properties as specified by PROPERTY/VALUE pairs.

`h = fec.bchdec(bchenc_object)` constructs a BCH decoder object `dec` by reading the property values from the BCH encoder object `bchenc_object`.

Properties A BCH decoder object has the following properties, which are all writable except for the ones explicitly noted otherwise.

Property	Description
Type	The type of decoder object. This property also displays the effective message length and codeword length, taking shortening and puncturing into consideration. This property is not writable.
N	The codeword length of the base code, not including shortening or puncturing.

Property	Description
K	The uncoded message length, not including shortening.
T	The number of errors the base code is capable of correcting. This property is not writable.
ShortenedLength	The number of bits by which the code has been shortened.
ParityPosition	Must be 'beginning' or 'end'. Specifies if parity bits should appear at the beginning or end of the codeword.
PuncturePattern	Indicates which parity bits in a codeword are punctured. This binary-valued vector is of length N-K. Values of "0" indicate bits that are punctured, and values of "1" indicate bits that are not.
GenPoly	The generator polynomial for the code. GenPoly must be a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial.

Methods

decoded = decode(dec, code)

Attempts to decode the received signal in CODE using the BCH decoder DEC. CODE must be a vector of binary elements, with an integer multiple of N-ShortenedLength-(Number of punctures) elements per column. There may be multiple codewords per channel, where each group of N-ShortenedLength-(Number of punctures) input elements represents one codeword to be decoded. Each column of CODE is

considered to be a separate channel, with the same BCH code applied to each channel.

decoded = decode(dec,code,erasures)

Attempts to decode the received signal with the additional erasure information provided by the ERASURES vector. The size of the ERASURES vector must be the same as the size of CODE, where a 0 marks no erasure, and a 1 marks an erased bit.

[decoded,cnumerr] = DECODE(...)

Returns an array CNUMERR with the same number of columns as CODE. Within each column of CNUMERR, each element is the number of corrected errors in the corresponding codeword of CODE. A value of -1 in CNUMERR indicates a decoding failure in that codeword in CODE.

[decoded,cnumerr,ccode] = decode(...)

Returns CCODE, the corrected version of CODE. The array CCODE is in the same format as CODE. If a decoding failure occurs in a certain codeword (i.e. full or partial column of CODE), then the corresponding full or partial column in CCODE contains that full or partial column unchanged.

Usage Examples

```
% Code parameters
n = 7; k =4;
% Construct encoder
coder = fec.bchenc(n,k);
% Message to encode
msg = [0 1 1 0]';
% Perform Coding
code = encode(coder,msg);
% Construct decoder from encoder
decoder = fec.bchdec(coder);
% Introduce 1 error in the codeword
code(end) = 0;
[decoded,cnumerr,ccode] = decode(decoder,code);

% Test for a encoding a punctured RS code
```

```
n = 7; k = 3;
msg = [1 1 1]';
puncVec = [0 1 1 1];

coderNonPunc = fec.rsenc(n,k);
code = encode(coderNonPunc,msg);

coderPunc = copy(coderNonPunc);
coderPunc.puncturepattern = puncVec;
codePunc = encode(coderPunc,msg);

expCode = code([1:k k+find(puncVec)]);
```


Purpose Construct BCH encoder object

Syntax

```
h = fec.bchenc
h = fec.bchenc(N,K)
h = fec.bchenc(property1, value1, ...)
h = fec.bchenc(bchdec_object)
```

Description

`enc = fec.bchenc` constructs a BCH encoder `enc` with default properties. It is equivalent to: `enc = fec.bchenc(7,4)`

`enc = fec.bchenc(N,K)` constructs an (N,K) BCH encoder object `enc`.

`enc = fec.bchenc(property1, valule1, ...)` constructs a BCH encoder object `enc` with properties as specified by PROPERTY/VALUE pairs.

`enc = fec.bchenc(bchdec_object)` constructs a BCH encoder object `enc` by reading the property values from the BCH decoder object `bchdec_object`

Properties A BCH encoder object has the following properties, which are all writable except for the ones explicitly noted otherwise.

Property	Description
Type	The type of encoder object. This property also displays the effective message length and codeword length, taking shortening and puncturing into consideration. This property is not writable.
N	The codeword length of the base code, not including shortening or puncturing.

Property	Description
K	The uncoded message length, not including shortening.
T	The number of errors the base code is capable of correcting. This property is not writable.
ShortenedLength	The number of bits by which the code has been shortened.
ParityPosition	Must be 'beginning' or 'end'. Specifies if parity bits should appear at the beginning or end of the codeword.
PuncturePattern	Indicates which parity bits in a codeword are punctured. This binary-valued vector is of length N-K. Values of "0" indicate bits that are punctured, and values of "1" indicate bits that are not.
GenPoly	The generator polynomial for the code. GenPoly must be a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial.

Methods

CODEWORD = ENCODE(ENC, MSG)

Encodes MSG using the BCH code specified by a BCH encoder object ENC. MSG must be an array of binary elements, with an integer multiple of K-ShortenedLength elements per column. There may be multiple codewords per channel, where each group of K-ShortenedLength input elements represents one message word to be encoded. Each column of MSG is considered to be a separate channel, with the same BCH code applied to each channel.

Usage Examples

```
%Create BCH encoder object.
enc = fec.bchenc(7,4);

% Create a message to be encoded.
msg = [0 1 1 0]';

% Encode msg with the ENCODE function.
code = encode(enc,msg);

% Create a shortened encoder
encShort = copy(enc);
encShort.ShortenedLength = 1;

% Create a shortened message
msgShort = [0 1 1]';

codeShort = encode(encShort,msgShort);

% Create a punctured encoder
encPunc = copy(enc);
encPunc.PuncturePattern = [1 0 1];

% Create a punctured message
codePunc = encode(encPunc,msg);
```

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.

fec.rsdec

Purpose Construct Reed-Solomon decoder object

Syntax

```
h = fec.rsdec
h = fec.rsdec(N,K)
h = fec.rsdec(property1, value1, ...)
h = fec.rsdec(rsenc_object)
```

Description

`dec = fec.rsdec` constructs a Reed-Solomon decoder with default properties. It is equivalent to `dec = fec.rsdec(7,3)`

`dec = fec.rsdec(N,K)` constructs an (N,K) Reed-Solomon decoder object `dec`.

`dec = fec.rsdec(property1, value1, ...)` constructs a Reed-Solomon decoder object `dec` with properties as specified by PROPERTY/VALUE pairs.

`dec = fec.rsdec(rsenc_object)` constructs a Reed-Solomon decoder object `dec` by reading the property values from the Reed-Solomon encoder object `rsenc_object`.

Properties A Reed-Solomon decoder object has the following properties, all of which are writable, except for the ones explicitly noted otherwise.

Property	Description
Type	The type of decoder object. This property also displays the effective message length
N	The codeword length of the base code, not including shortening or puncturing.
K	The uncoded message length, not including shortening.
T	The number of errors the base code is capable of correcting. This property is not writable.

Property	Description
ShortenedLength	The number of symbols by which the code has been shortened.
ParityPosition	Must be 'beginning' or 'end'. Specifies if parity bits should appear at the beginning or end of the codeword.
PuncturePattern	Indicates which parity symbols in a codeword are punctured. This binary-valued vector is of length N-K. Values of "0" indicate symbols that are punctured, and values of "1" indicate symbols that are not.
GenPoly	The generator polynomial for the code. GENPOLY must be a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial.

Methods

The fec.rsdec object has a method for encoding messages.

DECODED = DECODE(DEC, CODE)

Attempts to decode the received signal in CODE using the Reed-Solomon decoder DEC. CODE must be a vector of integer elements, with an integer multiple of N-ShortenedLength-(Number of punctures) elements per column. There may be multiple codewords per channel, where each group of N-ShortenedLength-(Number of punctures) input elements represents one codeword to be decoded. Each column of CODE is considered to be a separate channel, with the same Reed-Solomon code applied to each channel.

DECODED = DECODE(DEC, CODE, ERASURES)

Attempts to decode the received signal with the additional erasure information provided by the ERASURES vector. The size of the ERASURES vector must be the same as the size of CODE, where a 0 marks no erasure, and a 1 marks an erased symbol.

[DECODED,CNUMERR] = DECODE(...)

Returns an array CNUMERR with the same number of columns as CODE. Within each column of CNUMERR, each element is the number of corrected errors in the corresponding codeword of CODE. A value of -1 in CNUMERR indicates a decoding failure in that codeword in CODE.

[DECODED,CNUMERR,CCODE] = DECODE(...)

Returns CCODE, the corrected version of CODE. The array CCODE is in the same format as CODE. If a decoding failure occurs in a certain codeword (i.e. full or partial column of CODE), then the corresponding full or partial column in CCODE contains that full or partial column unchanged.

Usage Examples

```
% Code parameters
n = 7; k = 3;
% Construct encoder
coder = fec.rsenc(n,k);
% Message to encode
msg = [0 1 2]';
% Perform Coding
code = encode(coder,msg);
% Construct decoder from encoder
decoder = fec.rsdec(coder);
% Introduce 1 error in the codeword
code(end) = 0;
[decoded,cnumerr,ccode] = decode(decoder,code);
```

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.

Purpose Construct Reed-Solomon encoder object

Syntax

```
enc = fec.rsenc
enc = fec.rsenc(N,K)
enc = fec.rsenc(property1, ...)
enc = fec.rsenc(rsdec_object)
```

Description

`enc = fec.rsenc` constructs a Reed-Solomon encoder with default properties equivalent to: `enc = rsenc(7,3)`

`enc = fec.rsenc(N,K)` constructs an (N,K) Reed-Solomon encoder object `enc`.

`enc = fec.rsenc(property1, value1, ...)` constructs a Reed-Solomon encoder object `enc` with properties as specified by PROPERTY/VALUE pairs.

`enc = fec.rsenc(rsdec_object)` constructs a Reed-Solomon encoder object `enc` by reading the property values from the RS decoder object `rsdec_object`.

Properties A Reed-Solomon encoder object has the following properties, all of which are writable, except for the ones explicitly noted otherwise.

Property	Description
Type	The type of encoder object. This property also displays the effective message length
N	The codeword length of the base code, not including shortening or puncturing.
K	The uncoded message length, not including shortening.
T	The number of errors the base code is capable of correcting. This property is not writable.

Property	Description
ShortenedLength	The number of symbols by which the code has been shortened.
ParityPosition	Must be 'beginning' or 'end'. Specifies if parity symbols should appear at the beginning or end of the codeword.
GenPoly	The generator polynomial for the code. GenPoly must be a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial.

Methods

The fec.rsenc object has a method for encoding messages.

codeword =encode(enc, msg)

Encodes MSG using the Reed-Solomon code specified by a Reed-Solomon encoder object ENC. MSG must be an array of integer elements, with an integer multiple of K-ShortenedLength elements per column. There may be multiple codewords per channel, where each group of K-ShortenedLength input elements represents one message word to be encoded. Each column of MSG is considered to be a separate channel, with the same Reed-Solomon code applied to each channel.

Usage Examples

```
% Create Reed-Solomon encoder object.  
enc = fec.rsenc(7,3);  
  
% Create a message to be encoded.  
msg = [0 1 0]';  
  
% Encode msg with the ENCODE function.  
code = encode(enc,msg);  
  
% Create a shortened encoder  
encShort = copy(enc);
```



```
encShort.ShortenedLength = 1;  
  
% Create a shortened message  
msgShort = [0 1]';  
  
codeShort = encode(encShort,msgShort);
```

fft

Purpose Discrete Fourier transform

Syntax `fft(x)`

Description `fft(x)` is the discrete Fourier transform (DFT) of the Galois vector x . If x is in the Galois field $GF(2^m)$, the length of x must be 2^m-1 .

Examples

```
m = 4;
n = 2^m-1;
x = gf(randint(n,1,2^m),m); % Random vector
y = fft(x); % Transform of x
z = ifft(y); % Inverse transform of y
ck = isequal(z,x) % Check that ifft(fft(x)) recovers x.
```

The output is

```
ck =
```

```
1
```

Limitations The Galois field over which this function works must have 256 or fewer elements. In other words, x must be in the Galois field $GF(2^m)$, where m is an integer between 1 and 8.

Algorithm If x is a column vector, `fft` applies `dftmtx` to the primitive element of the Galois field and multiplies the resulting matrix by x .

See Also `ifft`, `dftmtx`, “Signal Processing Operations in Galois Fields”

Purpose Filter signal with channel object

Syntax `y = filter(chan,x)`

Description `y = filter(chan,x)` processes the baseband signal vector `x` with the channel object `chan`. The result is the signal vector `y`. The final state of the channel is stored in `chan`. You can construct `chan` using either `rayleighchan` or `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`.

Examples Examples using this function are in “Using Fading Channels”.

See Also `rayleighchan`, `ricianchan`, `reset`, “Fading Channels”

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

filter (gf)

Purpose 1-D digital filter over Galois field

Syntax
`y = filter(b,a,x)`
`[y,zf] = filter(b,a,x)`

Description `y = filter(b,a,x)` filters the data in the vector `x` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. The vectors `b`, `a`, and `x` must be Galois vectors in the same field. If `a(1)` is not equal to 1, `filter` normalizes the filter coefficients by `a(1)`. As a result, `a(1)` must be nonzero.

The filter is a “Direct Form II Transposed” implementation of the standard difference equation below.

$$\begin{aligned} a(1)*y(n) = & b(1)*x(n) + b(2)*x(n-1) + \dots + b(nb+1)*x(n-nb) \dots \\ & - a(2)*y(n-1) - \dots - a(na+1)*y(n-na) \end{aligned}$$

`[y,zf] = filter(b,a,x)` returns the final conditions of the filter delays in the Galois vector `zf`. The length of the vector `zf` is `max(size(a),size(b))-1`.

Examples An example is in “Huffman Coding”.

Purpose Estimate delay(s) between signals

Syntax
`D = finddelay(X,Y)`
`D = finddelay(...,MAXLAG)`

Description `D = finddelay(X,Y)`, where X and Y are row or column vectors, returns an estimate of the delay D between X and Y , where X serves as the reference vector. If Y is delayed with respect to X , then D is positive. If Y is advanced with respect to X , then D is negative. Delays in X and Y can be introduced by pre-pending zeros.

X and Y need not be exact delayed copies of each other, as `finddelay(X,Y)` returns an estimate of the delay via cross-correlation. However this estimated delay has a useful meaning only if there is sufficient correlation between delayed versions of X and Y . Also, if several delays are possible, as in the case of periodic signals, the delay with the smallest absolute value is returned. In the case that both a positive and a negative delay with the same absolute value are possible, the positive delay is returned.

`D = finddelay(X,Y)`, where X is a matrix of size MX -by- NX ($MX > 1$ and $NX > 1$) and Y is a matrix of size MY -by- NY ($MY > 1$ and $NY > 1$), returns a row vector D of estimated delays between each column of X and the corresponding column of Y . With this usage the number of columns of X must be equal to the number of columns of Y (i.e., $NX = NY$).

`D = finddelay(...,MAXLAG)`, uses `MAXLAG` as the maximum correlation window size used to find the estimated delay(s) between X and Y . The usage of `MAXLAG` is detailed in the table below.

By default, `MAXLAG` is equal to $\text{MAX}(LX, LY) - 1$ for two vector inputs (where LX and LY are the lengths of X and Y , respectively), $\text{MAX}(MX, MY) - 1$ for two matrix inputs, and $\text{MAX}(LX, MY) - 1$ or $\text{MAX}(MX, LY) - 1$ for one vector input and one matrix input. If `MAXLAG` is input as `[]`, it is replaced by the default value. If any element of `MAXLAG` is negative, it is replaced by its absolute value. If any element of `MAXLAG` is not integer-valued, or is complex, `Inf`, or `NaN`, then `finddelay` returns an error.

The calculation of the vector of estimated delays, D , depends on X , Y , and $MAXLAG$ as shown in the following table.

MAXLAG	X	Y	D is calculated by...
Integer-valued scalar	Row or column vector or matrix	Row or column vector or matrix	Cross-correlating the columns of X and Y over a range of lags $-MAXLAG:MAXLAG$.
Integer-valued row or column vector	Row or column vector of length $LX \geq 1$	Matrix of size MY -by- NY ($MY > 1, NY > 1$)	Cross-correlating X and column j of Y over a range of lags $-MAXLAG(j):MAXLAG(j)$, for $j=1:NY$.
Integer-valued row or column vector	Matrix of size MX -by- NX ($MX > 1, NX > 1$)	Row or column vector of length $LY \geq 1$	Cross-correlating column j of X and Y over a range of lags $-MAXLAG(j):MAXLAG(j)$, for $j=1:NX$.
Integer-valued row or column vector	Matrix of size MX -by- NX ($MX > 1, NX > 1$)	Matrix of size MY -by- NY ($MY > 1, NY = NX > 1$)	Cross-correlating column j of X and column j of Y over a range of lags $-MAXLAG(j):MAXLAG(j)$, for $j=1:NY$.

Treating X as Multiple Channels

If you wish to treat a row vector X of length LX as comprising one sample from LX different channels, you need to append one or more rows of zeros to X so that it appears as a matrix. Then each column of X will be considered a channel.

For example, $X = [1 \ 1 \ 1 \ 1]$ is considered a single channel comprising four samples. To treat it as four different channels, each channel comprising one sample, define a new matrix Xm :

$$Xm = \begin{bmatrix} 1 & 1 & 1 & 1; \\ 0 & 0 & 0 & 0 \end{bmatrix};$$

Each column of Xm corresponds to a single channel, each one containing the samples 1 and 0.

Theory and Algorithm

The `finddelay` function uses the `xcorr` function of Signal Processing Toolbox to determine the cross-correlation between each pair of signals at all possible lags specified by the user. The normalized cross-correlation between each pair of signals is then calculated. The estimated delay is given by the negative of the lag for which the normalized cross-correlation has the largest absolute value.

If more than one lag leads to the largest absolute value of the cross-correlation, such as in the case of periodic signals, the delay is chosen as the negative of the smallest (in absolute value) of such lags.

Pairs of signals need not be exact delayed copies of each other. However, the estimated delay has a useful meaning only if there is sufficient correlation between at least one pair of the delayed signals.

Examples

X and Y Are Vectors, and MAXLAG Is Not Specified

The following shows Y being delayed with respect to X by two samples.

```
X = [1 2 3];
Y = [0 0 1 2 3];
D = finddelay(X,Y)
```

The result is $D = 2$.

Here is a case of Y advanced with respect to X by three samples.

```
X = [0 0 0 1 2 3 0 0]';
Y = [1 2 3 0]';
D = finddelay(X,Y)
```

The result is $D = -3$.

The following illustrates a case where Y is aligned with X but is noisy.

```
X = [0 0 1 2 3 0];
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];
D = finddelay(X,Y)
```

The result is $D = 0$.

If Y is a periodic version of X, the smallest possible delay is returned.

```
X = [0 1 2 3];  
Y = [1 2 3 0 0 0 0 1 2 3 0 0];  
D = finddelay(X,Y)
```

The result is D = -1.

X is a Vector, Y a Matrix, and MAXLAG Is a Scalar

MAXLAG is specified as a scalar (same maximum window sizes).

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

The result is D = [0 -1 1 1].

X and Y Are Matrices, and MAXLAG Is Not Specified

```
X = [0 1 0 0;  
      1 2 0 0;  
      2 0 1 0;  
      1 0 2 1;  
      0 0 0 2];  
Y = [0 0 1 0;  
      1 1 2 0;  
      2 2 0 1;  
      1 0 0 2;  
      0 0 0 0];  
D = finddelay(X,Y)
```

The result is D = [0 -1 -2 -1].

X and Y Are Matrices, and MAXLAG Is Specified

```
X = [0 1 0 0;  
     1 2 0 0;  
     2 0 1 0;  
     1 0 2 1;  
     0 0 0 2];  
Y = [0 0 1 0;  
     1 1 2 0;  
     2 2 0 1;  
     1 0 0 2;  
     0 0 0 0];  
MAXLAG = [10 10 20 20];  
D = finddelay(X,Y,MAXLAG)
```

The result is $D = [0 \ 1 \ -2 \ -1]$.

See Also

`alignsignals`, `xcorr`

fmdemod

Purpose Frequency demodulation

Syntax
`z = fmdemod(y,Fc,Fs,freqdev)`
`z = fmdemod(y,Fc,Fs,freqdev,ini_phase)`

Description `z = fmdemod(y,Fc,Fs,freqdev)` demodulates the modulating signal `z` from the carrier signal using frequency demodulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least $2 \cdot Fc$. The `freqdev` argument is the frequency deviation (Hz) of the modulated signal `y`.

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

Examples An example using `fmdemod` is on the reference page for `fmod`.

See Also `fmod`, `pmod`, `pmdemod`, “Modulation”

Purpose

Frequency modulation

Syntax

```
y = fmmod(x,Fc,Fs,freqdev)
y = fmmod(x,Fc,Fs,freqdev,ini_phase)
```

Description

`y = fmmod(x,Fc,Fs,freqdev)` modulates the message signal `x` using frequency modulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least $2 \cdot Fc$. The `freqdev` argument is the frequency deviation constant (Hz) of the modulated signal.

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

Examples

The code below modulates a multichannel signal using `fmmod` and demodulates it using `fmdemod`.

```
Fs = 8000; % Sampling rate of signal
Fc = 3000; % Carrier frequency
t = [0:Fs]'/Fs; % Sampling times
s1 = sin(2*pi*300*t)+2*sin(2*pi*600*t); % Channel 1
s2 = sin(2*pi*150*t)+2*sin(2*pi*900*t); % Channel 2
x = [s1,s2]; % Two-channel signal
dev = 50; % Frequency deviation in modulated signal
y = fmmod(x,Fc,Fs,dev); % Modulate both channels.
z = fmdemod(y,Fc,Fs,dev); % Demodulate both channels.
```

See Also`fmdemod`, `ammod`, `pmmod`, “Modulation”

fskdemod

Purpose Frequency shift keying demodulation

Syntax

```
z = fskdemod(y,M,freq_sep,nsamp)
z = fskdemod(y,M,freq_sep,nsamp,Fs)
z = fskdemod(y,M,freq_sep,nsamp,Fs,symbol_order)
```

Description `z = fskdemod(y,M,freq_sep,nsamp)` noncoherently demodulates the complex envelope `y` of a signal using the frequency shift key method. `M` is the alphabet size and must be an integer power of 2. `freq_sep` is the frequency separation between successive frequencies in Hz. `nsamp` is the required number of samples per symbol and must be a positive integer greater than 1. The sampling frequency is 1 Hz. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

`z = fskdemod(y,M,freq_sep,nsamp,Fs)` specifies the sampling frequency in Hz.

`z = fskdemod(y,M,freq_sep,nsamp,Fs,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples The example below illustrates FSK modulation and demodulation over an AWGN channel.

```
M = 2; k = log2(M);
EbNo = 5;
Fs = 16; nsamp = 17; freqsep = 8;
msg = randint(5000,1,M); % Random signal
txsig = fskmod(msg,M,freqsep,nsamp,Fs); % Modulate.
msg_rx = awgn(txsig,EbNo+10*log10(k)-10*log10(nsamp),...
    'measured',[],'dB'); % AWGN channel
msg_rrx = fskdemod(msg_rx,M,freqsep,nsamp,Fs); % Demodulate
[num,BER] = biterr(msg,msg_rrx) % Bit error rate
BER_theory = berawgn(EbNo,'fsk',M,'noncoherent') % Theoretical BER
```

The output is shown below. Your BER value might vary because the example uses random numbers.

BER =

0.1086

BER_theory =

0.1029

See Also

fskmod, pskmod, pskdemod, “Modulation”

fskmod

Purpose Frequency shift keying modulation

Syntax

```
y = fskmod(x,M,freq_sep,nsamp)
y = fskmod(x,M,freq_sep,nsamp,Fs)
y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)
y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)
```

Description `y = fskmod(x,M,freq_sep,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using frequency shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. `freq_sep` is the desired separation between successive frequencies in Hz. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer greater than 1. The sampling rate of `y` is 1 Hz. By the Nyquist sampling theorem, `freq_sep` and `M` must satisfy $(M-1)*freq_sep \leq 1$. 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 $(M-1)*freq_sep \leq Fs$.

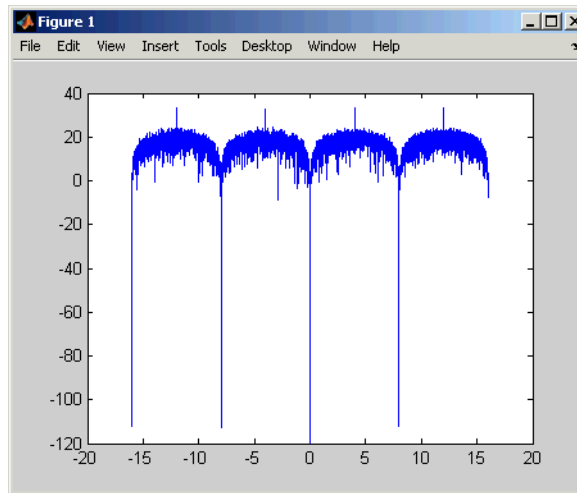
`y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)` specifies the phase continuity. Set `phase_cont` to 'cont' to force phase continuity across symbol boundaries in `y`, or 'discont' to avoid forcing phase continuity. The default is 'cont'.

`y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples The example below illustrates the syntax of `fskmod` using a random signal.

```
M = 4; freqsep = 8; nsamp = 8; Fs = 32;
```

```
x = randint(1000,1,M); % Random signal
y = fskmod(x,M,freqsep,nsamp,Fs); % Modulate.
ly = length(y);
% Create an FFT plot.
freq = [-Fs/2 : Fs/ly : Fs/2 - Fs/ly];
Syy = 10*log10(fftshift(abs(fft(y))));
plot(freq,Syy)
```

**See Also**

fskdemod, pskmod, pskdemod, “Modulation”

gen2par

Purpose Convert between parity-check and generator matrices

Syntax
`parmat = gen2par(genmat)`
`genmat = gen2par(parmat)`

Description `parmat = gen2par(genmat)` converts the standard-form binary generator matrix `genmat` into the corresponding parity-check matrix `parmat`.

`genmat = gen2par(parmat)` converts the standard-form binary parity-check matrix `parmat` into the corresponding generator matrix `genmat`.

The standard forms of the generator and parity-check matrices for an $[n,k]$ binary linear block code are shown in the table below

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k \ P]$ or $[P \ I_k]$	k-by-n
Parity-check	$[-P' \ I_{n-k}]$ or $[I_{n-k} \ -P']$	(n-k)-by-n

where I_k is the identity matrix of size k and the ' symbol indicates matrix transpose. Two standard forms are listed for each type, because different authors use different conventions. For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, $-1 = 1$ in the binary field.

Examples The commands below convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

```
parmat = hammgen(3)
genmat = gen2par(parmat)
parmat2 = gen2par(genmat) % Ans should be the same as parmat above
```

The output is

parmat =

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

genmat =

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

parmat2 =

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

See Also

cyclgen, hammgen, “Block Coding”

genqamdemod

Purpose General quadrature amplitude demodulation

Syntax `z = genqamdemod(y, const)`

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.genqamdemod` instead.

`z = genqamdemod(y, const)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. The complex vector `const` specifies the signal mapping. If `y` is a matrix with multiple rows, the function processes the columns independently.

Examples The reference page for `genqammod` has an example that uses `genqamdemod`.

See Also `genqammod`, `qammod`, `qamdemod`, `pammod`, `pamdemod`, “Modulation”

Purpose General quadrature amplitude modulation

Syntax `y = genqammod(x,const)`

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.genqammod` instead.

`y = genqammod(x,const)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. The message signal must consist of integers between 0 and `length(const) - 1`. The complex vector `const` specifies the signal mapping. If `x` is a matrix with multiple rows, the function processes the columns independently.

Examples

The code below plots a signal constellation that has a hexagonal structure. It also uses `genqammod` and `genqamdemod` to modulate and demodulate a message `[3 8 5 10 7]` using this constellation.

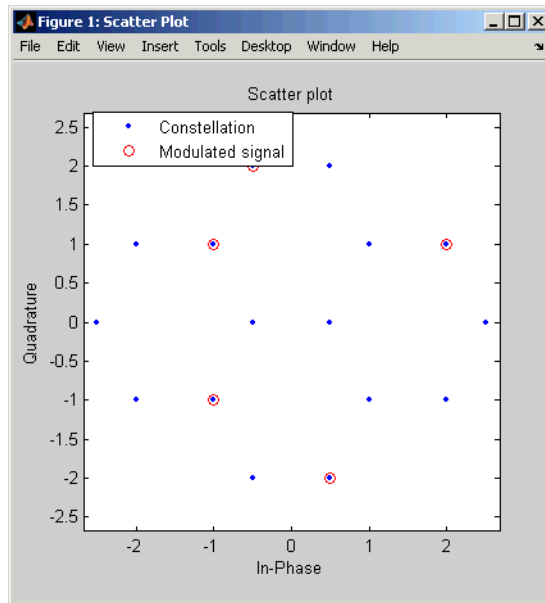
```
% Describe hexagonal constellation.
inphase = [1/2 1 1 1/2 1/2 2 2 5/2];
quadr = [0 1 -1 2 -2 1 -1 0];
inphase = [inphase;-inphase]; inphase = inphase(:);
quadr = [quadr;quadr]; quadr = quadr(:);
const = inphase + j*quadr;

% Plot constellation.
h = scatterplot(const);

% Modulate message using this constellation.
x = [3 8 5 10 7]; % Message signal
y = genqammod(x,const);
z = genqamdemod(y,const); % Demodulate.

% Plot modulated signal in same figure.
hold on; scatterplot(y,1,0,'ro',h);
```

```
legend('Constellation','Modulated signal','Location','NorthWest'); % Include legend.  
hold off;
```



Another example using this function is the Gray-coded constellation example in “Examples of Signal Constellation Plots”.

See Also

genqamdemod, qammod, qamdemod, pammod, pamdemod, “Modulation”

Purpose Create Galois field array

Syntax

```
x_gf = gf(x,m)
x_gf = gf(x,m,prim_poly)
x_gf = gf(x)
```

Description `x_gf = gf(x,m)` creates a Galois field array from the matrix `x`. The Galois field has 2^m elements, where `m` is an integer between 1 and 16. The elements of `x` must be integers between 0 and 2^m-1 . The output `x_gf` is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate `x_gf` using operators or functions such as `+` or `det`, MATLAB works within the Galois field you have specified.

Note To learn how to manipulate `x_gf` using familiar MATLAB operators and functions, see “Galois Field Computations”. To learn how the integers in `x` represent elements of $GF(2^m)$, see “How Integers Correspond to Galois Field Elements”.

`x_gf = gf(x,m,prim_poly)` is the same as the previous syntax, except it uses the primitive polynomial `prim_poly` to define the field. `prim_poly` is the integer representation of a primitive polynomial. For example, the number 41 represents the polynomial D^5+D^2+1 because the binary form of 37 is 1 0 0 1 0 1. For more information about the primitive polynomial, see “Specifying the Primitive Polynomial”.

`x_gf = gf(x)` creates a $GF(2)$ array from the matrix `x`. Each element of `x` must be 0 or 1.

Default Primitive Polynomials

The table below lists the primitive polynomial that `gf` uses by default for each Galois field $GF(2^m)$. To use a different primitive polynomial, specify `prim_poly` as an input argument when you invoke `gf`.

m	Default Primitive Polynomial	Integer Representation
1	$D + 1$	3
2	$D^2 + D + 1$	7
3	$D^3 + D + 1$	11
4	$D^4 + D + 1$	19
5	$D^5 + D^2 + 1$	37
6	$D^6 + D + 1$	67
7	$D^7 + D^3 + 1$	137
8	$D^8 + D^4 + D^3 + D^2 + 1$	285
9	$D^9 + D^4 + 1$	529
10	$D^{10} + D^3 + 1$	1033
11	$D^{11} + D^2 + 1$	2053
12	$D^{12} + D^6 + D^4 + D + 1$	4179
13	$D^{13} + D^4 + D^3 + D + 1$	8219
14	$D^{14} + D^{10} + D^6 + D + 1$	17475
15	$D^{15} + D + 1$	32771
16	$D^{16} + D^{12} + D^3 + D + 1$	69643

Examples

For examples that use gf, see

- “Example: Creating Galois Field Variables”
- “Example: Representing a Primitive Element”

-
- Other sample code within “Galois Field Computations”
 - The Galois field demonstration: type `showdemo gfdemo`.

See Also

`gftable`, list of functions and operators for Galois field computations,
`gfdemo`, “Galois Field Computations”

gfadd

Purpose Add polynomials over Galois field

Syntax

```
c = gfadd(a,b)
c = gfadd(a,b,p)
c = gfadd(a,b,p,len)
c = gfadd(a,b,field)
```

Description

Note This function performs computations in $GF(p^m)$ where p is prime. To work in $GF(2^m)$, apply the $+$ operator to Galois arrays of equal size. For details, see “Example: Addition and Subtraction”.

`c = gfadd(a,b)` adds two $GF(2)$ polynomials, a and b . If a and b are vectors of the same orientation but different lengths, then the shorter vector is zero-padded. If a and b are matrices they must be of the same size.

`c = gfadd(a,b,p)` adds two $GF(p)$ polynomials, where p is a prime number. a , b , and c are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and $p-1$. If a and b are matrices of the same size, the function treats each row independently.

`c = gfadd(a,b,p,len)` adds row vectors a and b as in the previous syntax, except that it returns a row vector of length `len`. The output c is a truncated or extended representation of the sum. If the row vector corresponding to the sum has fewer than `len` entries (including zeros), extra zeros are added at the end; if it has more than `len` entries, entries from the end are removed.

`c = gfadd(a,b,field)` adds two $GF(p^m)$ elements, where m is a positive integer. a and b are the exponential format of the two elements, relative to some primitive element of $GF(p^m)$. `field` is the matrix listing all elements of $GF(p^m)$, arranged relative to the same primitive element. c is the exponential format of the sum, relative to the same primitive element. See “Representing Elements of Galois Fields” for an

explanation of these formats. If a and b are matrices of the same size, the function treats each element independently.

Examples

In the code below, `sum5` is the sum of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over $\text{GF}(5)$, and `linpart` is the degree-one part of `sum5`.

```
sum5 = gfadd([2 3 1],[4 2 3],5)
linpart = gfadd([2 3 1],[4 2 3],5,2)
```

The output is

```
sum5 =
      1      0      4

linpart =
      1      0
```

The code below shows that $A^2 + A^4 = A^1$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for $\text{GF}(9)$.

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
g = gfadd(2,4,field)
```

The output is

```
g =
      1
```

Other examples are in “Arithmetic in Galois Fields”.

See Also

`gfsub`, `gfconv`, `gfmul`, `gfdeconv`, `gfdiv`, `gftuple`, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Multiply polynomials over Galois field

Syntax

```
c = gfconv(a,b)
c = gfconv(a,b,p)
c = gfconv(a,b,field)
```

Description

Note This function performs computations in $\text{GF}(p^m)$, where p is prime. To work in $\text{GF}(2^m)$, 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 $\text{GF}(2)$ polynomials, a and b . The polynomial degree of the resulting $\text{GF}(2)$ polynomial c equals the degree of a plus the degree of b .

`c = gfconv(a,b,p)` multiplies two $\text{GF}(p)$ 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 $\text{GF}(p^m)$ 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 $\text{GF}(p^m)$. `field` is the matrix listing all elements of $\text{GF}(p^m)$, arranged relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats.

Examples

The command below shows that

$$(1 + x + x^4)(x + x^2) = x + 2x^2 + x^3 + x^5 + x^6$$

over GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)
```

The output is

```
gfc =
      0      1      2      1      0      1      1
```

The code below illustrates the identity

$$(x^r + x^s)^p = x^{rp} + x^{sp}$$

for the case in which $p = 7$, $r = 5$, and $s = 3$. (The identity holds when p is any prime number, and r and s are positive integers.)

```
p = 7; r = 5; s = 3;
a = gfrepconv([r s]); % x^r + x^s

% Compute a^p over GF(p).
c = 1;
for ii = 1:p
    c = gfconv(c,a,p);
end;

% Check whether c = x^(rp) + x^(sp).
powers = [];
for ii = 1:length(c)
    if c(ii)~=0
        powers = [powers, ii];
    end;
end;
if (powers==[r*p+1 s*p+1] | powers==[s*p+1 r*p+1])
```

gfconv

```
        disp('The identity is proved for this case of r, s, and p.')
    end
```

See Also

gfdeconv, gfadd, gfsub, gfmul, gftuple, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Produce cyclotomic cosets for Galois field

Syntax

```
c = gfcosets(m)
c = gfcosets(m,p)
```

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, use the `cosets` function.

`c = gfcosets(m)` produces cyclotomic cosets mod($2^m - 1$). Each row of the output GFCS contains one cyclotomic coset.

`c = gfcosets(m,p)` produces the cyclotomic cosets for $GF(p^m)$, where m is a positive integer and p is a prime number.

The output matrix `c` is structured so that each row represents one coset. The row represents the coset by giving the exponential format of the elements of the coset, relative to the default primitive polynomial for the field. For a description of exponential formats, see “Representing Elements of Galois Fields”.

The first column contains the coset leaders. Because the lengths of cosets might vary, entries of NaN are used to fill the extra spaces when necessary to make `c` rectangular.

A cyclotomic coset is a set of elements that all satisfy the same minimal polynomial. For more details on cyclotomic cosets, see the works listed in “References” on page 2-288.

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 =
     2     0     1
     2     0     1
```

See Also

`gfminpol`, `gfprimdf`, `gfroots`, “Galois Fields of Odd Characteristic” on page 1-13

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.

Purpose Divide polynomials over Galois field

Syntax

```
[quot,remd] = gfdeconv(b,a)
[quot,remd] = gfdeconv(b,a,p)
[quot,remd] = gfdeconv(b,a,field)
```

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, use the `deconv` function with Galois arrays. For details, see “Multiplication and Division of Polynomials”.

The `gfdeconv` function divides polynomials over a Galois field. (To divide elements of a Galois field, use `gfdiv` instead.) Algebraically, dividing polynomials over a Galois field is equivalent to deconvolving vectors containing the polynomials’ coefficients, where the deconvolution operation uses arithmetic over the same Galois field.

`[quot,remd] = gfdeconv(b,a)` computes the quotient `quot` and remainder `remd` of the division of `b` by `a` in $GF(2)$.

`[quot,remd] = gfdeconv(b,a,p)` divides the polynomial `b` by the polynomial `a` over $GF(p)$ and returns the quotient in `quot` and the remainder in `remd`. `p` is a prime number. `b`, `a`, `quot`, and `remd` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p-1`.

`[quot,remd] = gfdeconv(b,a,field)` divides the polynomial `b` by the polynomial `a` over $GF(p^m)$ and returns the quotient in `quot` and the remainder in `remd`. Here `p` is a prime number and `m` is a positive integer. `b`, `a`, `quot`, and `remd` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of $GF(p^m)$. `field` is the matrix listing all elements of $GF(p^m)$, arranged relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats.

Examples

The code below shows that

$$(x + x^3 + x^4) \div (1 + x) = 1 + x^3 \text{ Remainder } 2$$

in GF(3). It also checks the results of the division.

```
p = 3;
b = [0 1 0 1 1]; a = [1 1];
[quot, remd] = gfdeconv(b,a,p)
% Check the result.
bnew = gfadd(gfconv(quot,a,p),remd,p);
if isequal(bnew,b)
    disp('Correct.')
end;
```

The output is below.

```
quot =
     1     0     0     1

remd =
     2

Correct.
```

Working over GF(3), the code below outputs those polynomials of the form $x^k - 1$ ($k = 2, 3, 4, \dots, 8$) that $1 + x^2$ divides evenly.

```
p = 3; m = 2;
a = [1 0 1]; % 1+x^2
for ii = 2:p^m-1
    b = gfrepconv(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 “Algorithm” on page 2-304 on the `gfprimck` reference page, along with the irreducibility of $1 + x^2$ over $\text{GF}(3)$, this output indicates that $1 + x^2$ is not primitive for $\text{GF}(9)$.

Algorithm

The algorithm of `gfdeconv` is similar to that of the MATLAB function `deconv`.

See Also

`gfconv`, `gfadd`, `gfsub`, `gfdiv`, `gftuple`, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Divide elements of Galois field

Syntax

```
quot = gfdiv(b,a)
quot = gfdiv(b,a,p)
quot = gfdiv(b,a,field)
```

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, apply the `./` operator to Galois arrays. For details, see “Example: Division”.

The `gfdiv` function divides elements of a Galois field. (To divide polynomials over a Galois field, use `gfdeconv` instead.)

`quot = gfdiv(b,a)` divides `b` by `a` in $GF(2)$ element-by-element. `a` and `b` are scalars, vectors or matrices of the same size. Each entry in `a` and `b` represents an element of $GF(2)$. The entries of `a` and `b` are either 0 or 1.

`quot = gfdiv(b,a,p)` divides `b` by `a` in $GF(p)$ and returns the quotient. `p` is a prime number. If `a` and `b` are matrices of the same size, the function treats each element independently. All entries of `b`, `a`, and `quot` are between 0 and `p-1`.

`quot = gfdiv(b,a,field)` divides `b` by `a` in $GF(p^m)$ and returns the quotient. `p` is a prime number and `m` is a positive integer. If `a` and `b` are matrices of the same size, then the function treats each element independently. All entries of `b`, `a`, and `quot` are the exponential formats of elements of $GF(p^m)$ relative to some primitive element of $GF(p^m)$. `field` is the matrix listing all elements of $GF(p^m)$, arranged relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats.

In all cases, an attempt to divide by the zero element of the field results in a “quotient” of NaN.

Examples

The code below displays lists of multiplicative inverses in $GF(5)$ and $GF(25)$. It uses column vectors as inputs to `gfdiv`.

```
% Find inverses of nonzero elements of GF(5).
p = 5;
b = ones(p-1,1);
a = [1:p-1]';
quot1 = gfddiv(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 = gfddiv(b,a,field);
disp('Inverses in GF(25), expressed in EXPONENTIAL FORMAT with')
disp('respect to a root of the default primitive polynomial:')
disp('element  inverse')
disp([a, quot2])
```

See Also

gfmul, gfdeconv, gfconv, gftuple, “Galois Fields of Odd Characteristic”
on page 1-13

gffilter

Purpose Filter data using polynomials over prime Galois field

Syntax
`y = gffilter(b,a,x)`
`y = gffilter(b,a,x,p)`

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, use the `filter` function with Galois arrays. For details, see “Filtering”.

`y = gffilter(b,a,x)` filters the data in vector x with the filter described by vectors b and a . The vectors b , a and x must be in $GF(2)$, that is, be binary and y is also in $GF(2)$.

`y = gffilter(b,a,x,p)` filters the data x using the filter described by vectors a and b . y is the filtered data in $GF(p)$. p is a prime number, and all entries of a and b are between 0 and $p-1$.

By definition of the filter, y solves the difference equation

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + b(3)x(n-2) + \dots + b(B+1)x(n-B) \\ - a(2)y(n-1) - a(3)y(n-2) - \dots - a(A+1)y(n-A)$$

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 .

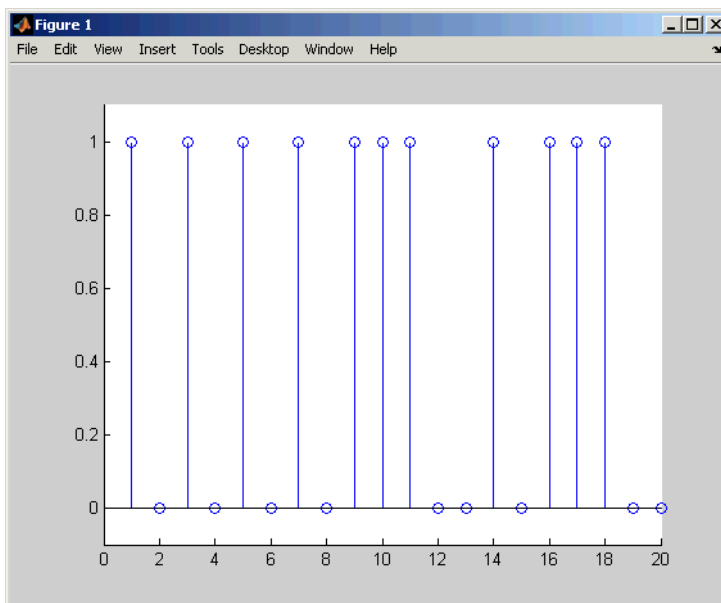
The vector a represents the degree- n_a polynomial

$$a(1) + a(2)x + a(3)x^2 + \dots + a(A+1)x^A$$

Examples

The impulse response of a particular filter is given in the code and diagram below.

```
b = [1 0 0 1 0 1 0 1];  
a = [1 0 1 1];  
y = gffilter(b,a,[1,zeros(1,19)]);  
stem(y);  
axis([0 20 -.1 1.1])
```

**See Also**

gfconv, gfadd, filter, “Galois Fields of Odd Characteristic” on page 1-13

gflneq

Purpose Find particular solution of $Ax = b$ over prime Galois field

Syntax

```
x = gflneq(A,b)
x = gflneq(A,b,p)
[x,vld] = gflneq(...)
```

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 = gflneq(A,b)` outputs a particular solution of the linear equation $Ax = b$ 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 = gflneq(A,b,p)` returns a particular solution of the linear equation $Ax = b$ 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] = gflneq(...)` 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] = gflneq(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] = gflinq(zeros(3,3),[2;0;0],3)
```

The output is below.

```
This linear equation has no solution.
```

```
x2 =
```

```
[]
```

```
vld2 =
```

```
0
```

Algorithm

`gflinq` uses Gaussian elimination.

See Also

`gfadd`, `gfdiv`, `gfroots`, `gfrank`, `gfconv`, `conv`, “Galois Fields of Odd Characteristic” on page 1-13

gfminpol

Purpose Find minimal polynomial of Galois field element

Syntax

```
pol = gfminpol(k,m)
pol = gfminpol(k,m,p)
pol = gfminpol(k,prim_poly,p)
```

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, use the `minpol` function with Galois arrays. For details, see “Minimal Polynomials”.

`pol = gfminpol(k,m)` produces a minimal polynomial for each entry in k . k must be either a scalar or a column vector. Each entry in k represents an element of $GF(2^m)$ in exponential format. That is, k represents α^k , where α 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 r th row of `pol` gives the coefficients of the minimal polynomial of $A^{k(r)}$ in order of ascending powers.

`pol = gfminpol(k,prim_poly,p)` is the same as the first syntax listed, except that A is a root of the primitive polynomial for $GF(p^m)$ specified by `prim_poly`. `prim_poly` is a row vector that gives the coefficients of the degree- m primitive polynomial in order of ascending powers.

Examples

The syntax `gfminpol(k,m,p)` is used in the sample code in “Characterization of Polynomials”.

See Also

`gfprimdf`, `gfcosets`, `gfroots`, “Galois Fields of Odd Characteristic” on page 1-13

gfmul

Purpose Multiply elements of Galois field

Syntax
`c = gfmul(a,b,p)`
`c = gfmul(a,b,field)`

Description

Note This function performs computations in $GF(p^m)$ where p is prime. To work in $GF(2^m)$, apply the `.*` operator to Galois arrays. For details, see “Example: Multiplication”.

The `gfmul` function multiplies elements of a Galois field. (To multiply polynomials over a Galois field, use `gfconv` instead.)

`c = gfmul(a,b,p)` multiplies a and b in $GF(p)$. Each entry of a and b is between 0 and $p-1$. p is a prime number. If a and b are matrices of the same size, the function treats each element independently.

`c = gfmul(a,b,field)` multiplies a and b in $GF(p^m)$, where p is a prime number and m is a positive integer. a and b represent elements of $GF(p^m)$ in exponential format relative to some primitive element of $GF(p^m)$. `field` is the matrix listing all elements of $GF(p^m)$, arranged relative to the same primitive element. c is the exponential format of the product, relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats. If a and b are matrices of the same size, the function treats each element independently.

Examples

“Arithmetic in Galois Fields” contains examples. Also, the code below shows that

$$A^2 \cdot A^4 = A^6$$

where A is a root of the primitive polynomial $2 + 2x + x^2$ for $GF(9)$.

```
p = 3; m = 2;  
prim_poly = [2 2 1];  
field = gftuple([-1:p^m-2]',prim_poly,p);
```

```
a = gfmul(2,4,field)
```

The output is

```
a =
```

```
6
```

See Also

`gfddiv`, `gfdeconv`, `gfadd`, `gfsub`, `gftuple`, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Polynomial in traditional format

Syntax `gfpretty(a)` `gfpretty(a,st)` `gfpretty(a,st,n)`

Description `gfpretty(a)` displays a polynomial in a traditional format, using X as the variable and the entries of the row vector `a` as the coefficients in order of ascending powers. The polynomial is displayed in order of ascending powers. Terms having a zero coefficient are not displayed.

`gfpretty(a,st)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of X .

`gfpretty(a,st,n)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of X , and each line of the display has width `n` instead of the default value of 79.

Note For all syntaxes: If you do not use a fixed-width font, the spacing in the display might not look correct.

Examples

The code below displays statements about the elements of $GF(81)$.

```
p = 3; m = 4;
ii = randint(1,1,[1,p^m-2]); % Random exponent for prim element
primpolys = gfprimfd(m,'all',p);
[rows, cols] = size(primpolys);
jj = randint(1,1,[1,rows]); % 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

$$x^4 + 2x^3 + 2$$

then the element

$$A^{22}$$

can also be expressed as

$$A^2 + A^3 + 2$$

See Also

gftuple, gfprindf, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Check whether polynomial over Galois field is primitive

Syntax
`ck = gfprimck(a)`
`ck = gfprimck(a,p)`

Description

Note This function performs computations in $GF(p^m)$, where p is prime. If you are working in $GF(2^m)$, use the `isprimitive` function. For details, see “Finding Primitive Polynomials”.

`ck = gfprimck(a)` checks whether the degree- m $GF(2)$ polynomial a is a primitive polynomial for $GF(2^m)$, where $m = \text{length}(a) - 1$. The output `ck` is as follows:

- -1 if a is not an irreducible polynomial
- 0 if a is irreducible but not a primitive polynomial for $GF(p^m)$
- 1 if a is a primitive polynomial for $GF(p^m)$

`ck = gfprimck(a,p)` checks whether the degree- m $GF(P)$ polynomial a is a primitive polynomial for $GF(p^m)$. p is a prime number.

This function considers the zero polynomial to be “not irreducible” and considers all polynomials of degree zero or one to be primitive.

Examples “Characterization of Polynomials” contains examples.

Algorithm 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 $p^m - 1$.

See Also `gfprimfd`, `gfprimdf`, `gftuple`, `gfminpol`, `gfadd`, “Galois Fields of Odd Characteristic” on page 1-13

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.

gfprimdf

Purpose Provide default primitive polynomials for Galois field

Syntax
`pol = gfprimdf(m)`
`pol = gfprimdf(m,p)`

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To work in $GF(2^m)$, use the `primpoly` function. For details, see “Finding Primitive Polynomials”.

`pol = gfprimdf(m)` outputs the default primitive polynomial `pol` in $GF(2^m)$.

`pol = gfprimdf(m,p)` returns the row vector that gives the coefficients, in order of ascending powers, of the default primitive polynomial for $GF(p^m)$. m is a positive integer and p is a prime number.

Examples

The command below shows that $2 + x + x^2$ is the default primitive polynomial for $GF(5^2)$.

```
pol = gfprimdf(2,5)
pol =
```

```
      2      1      1
```

The code below displays the default primitive polynomial for each of the fields $GF(3^m)$, where m ranges between 3 and 5.

```
for m = 3:5
    gfpretty(gfprimdf(m,3))
end
```

The output is below.

```

                                     3
1 + 2 X + X
```


$$2 + X + X^4$$

$$1 + 2X + X^5$$

See Also

gfprimck, gfprimfd, gftuple, gfminpol, “Galois Fields of Odd Characteristic” on page 1-13

gfprimfd

Purpose Find primitive polynomials for Galois field

Syntax `pol = gfprimfd(m,opt,p)`

Description

Note This function performs computations in $\text{GF}(p^m)$, where p is prime. To work in $\text{GF}(2^m)$, use the `primpoly` function. For details, see “Finding Primitive Polynomials”.

- If $m = 1$, $\text{pol} = [1 \ 1]$.
- 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 $\text{GF}(p^m)$, where p is a prime number and m is a positive integer. If $m = 1$, $\text{pol} = [1 \ 1]$. 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 $\text{GF}(p^m)$ having the smallest possible number of nonzero terms	The row vector representing the polynomial
'max'	One primitive polynomial for $\text{GF}(p^m)$ 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 <i>opt</i> 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
```

gfprimfd

No primitive polynomial satisfies the given constraints.

fourterms =

[]

Algorithm

gfprimfd tests for primitivity using gfprimck. If *opt* is 'min', 'max', or omitted, polynomials are constructed by converting decimal integers to base *p*. Based on the decimal ordering, gfprimfd returns the first polynomial it finds that satisfies the appropriate conditions.

See Also

gfprimck, gfprimdf, gftuple, gfminpol, “Galois Fields of Odd Characteristic” on page 1-13

Purpose Compute rank of matrix over Galois field

Syntax `rk = gfrank(A,p)`

Description

Note This function performs computations in $GF(p^m)$ where p is prime. If you are working in $GF(2^m)$, use the rank function with Galois arrays. For details, see “Computing Ranks”.

`rk = gfrank(A,p)` calculates the rank of the matrix A in $GF(p)$, where p is a prime number.

Algorithm

`gfrank` uses an algorithm similar to Gaussian elimination.

Examples

In the code below, `gfrank` says that the matrix A has less than full rank. This conclusion makes sense because the determinant of A is zero mod p .

```
A = [1 0 1;
     2 1 0;
     0 1 1];
p = 3;
det_a = det(A); % Ordinary determinant of A
detmodp = rem(det(A),p); % Determinant mod p
rankp = gfrank(A,p);
disp(['Determinant = ',num2str(det_a)])
disp(['Determinant mod p is ',num2str(detmodp)])
disp(['Rank over GF(p) is ',num2str(rankp)])
```

The output is below.

```
Determinant = 3
Determinant mod p is 0
Rank over GF(p) is 2
```

Purpose Convert one binary polynomial representation to another

Syntax `polystandard = gfrepconv(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

$$A_0 + A_1x + A_2x^2 + \dots + A_{(m-1)}x^{m-1}$$

Each entry A_k is either one or zero.

2 `[A_0 A_1 A_2 ... A_(m-1)]` represents the polynomial

$$x^{A_0} + x^{A_1} + x^{A_2} + \dots + x^{A_{(m-1)}}$$

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 = gfrepconv(poly2)` converts from the second format to the first, for polynomials of degree *at least* 2. `poly2` and `polystandard` are row vectors. The entries of `poly2` are distinct integers, and at least one entry must exceed 1. Each entry of `polystandard` is either 0 or 1.

Note If `poly2` is a *binary* row vector, `gfrepconv` assumes that it is already in Format **1** above and returns it unaltered.

Examples

The command below converts the representation format of the polynomial $1 + x^2 + x^5$.

```
polystandard = gfrepconv([0 2 5])
```

polystandard =

1 0 1 0 0 1

See Also

gfpretty, “Galois Fields of Odd Characteristic” on page 1-13

gfroots

Purpose Find roots of polynomial over prime Galois field

Syntax

```
rt = gfroots(f,m,p)
rt = gfroots(f,prim_poly,p)
[rt,rt_tuple] = gfroots(...)
[rt,rt_tuple,field] = gfroots(...)
```

Description

Note This function performs computations in $\text{GF}(p^m)$, where p is prime. To work in $\text{GF}(2^m)$, use the `roots` function with Galois arrays. For details, see “Roots of Polynomials”.

For all syntaxes, `f` is a row vector that gives the coefficients, in order of ascending powers, of a degree- d polynomial.

Note `gfroots` lists each root exactly once, ignoring multiplicities of roots.

`rt = gfroots(f,m,p)` finds roots in $\text{GF}(p^m)$ 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 $\text{GF}(p^m)$.

`rt = gfroots(f,prim_poly,p)` finds roots in $\text{GF}(p^m)$ 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 $\text{GF}(p^m)$ that `prim_poly` represents.

`[rt,rt_tuple] = gfroots(...)` returns an additional matrix `rt_tuple`, whose k th 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] = groots(...)` 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 `groots` uses, see “Representing Elements of Galois Fields”.

Examples

“Roots of Polynomials” contains a description and example of the use of `groots`.

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] = groots(f,prim_poly,p);
% Display roots as polynomials in alpha.
for ii = 1:length(rt_tuple)
    gfppretty(rt_tuple(ii,:), 'alpha')
end
```

See Also

`gfprimdf`, “Galois Fields of Odd Characteristic” on page 1-13

gfsb

Purpose Subtract polynomials over Galois field

Syntax

```
c = gfsb(a,b,p)
c = gfsb(a,b,p,len)
c = gfsb(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 = gfsb(a,b,p)` calculates a minus b , where a and b represent polynomials over $GF(p)$ and p is a prime number. a , b , and c are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and $p-1$. If a and b are matrices of the same size, the function treats each row independently.

`c = gfsb(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 = gfsb(a,b,field)` calculates a minus b , where a and b are the exponential format of two elements of $GF(p^m)$, relative to some primitive element of $GF(p^m)$. p is a prime number and m is a positive integer. `field` is the matrix listing all elements of $GF(p^m)$, arranged relative to the same primitive element. c is the exponential format of the answer, relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats. If a and b are matrices of the same size, the function treats each element independently.

Examples

In the code below, `differ` is the difference of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over $\text{GF}(5)$, and `linpart` is the degree-one part of `differ`.

```
differ = gfsb([2 3 1],[4 2 3],5)
linpart = gfsb([2 3 1],[4 2 3],5,2)
```

The output is

```
differ =
      3      1      3

linpart =
      3      1
```

The code below shows that $A^2 - A^4 = A^7$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for $\text{GF}(9)$.

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
d = gfsb(2,4,field)
```

The output is

```
d =
      7
```

See Also

`gfadd`, `gfconv`, `gfmul`, `gfdeconv`, `gfdiv`, `gftuple`, “Galois Fields of Odd Characteristic” on page 1-13

gftable

Purpose Generate file to accelerate Galois field computations

Syntax `gftable(m,prim_poly);`

Description `gftable(m,prim_poly)` generates a file that can help accelerate computations in the field $GF(2^m)$ as described by the *nondefault* primitive polynomial `prim_poly`. The integer `m` is between 1 and 16. The integer `prim_poly` represents a primitive polynomial for $GF(2^m)$ using the format described in “Specifying the Primitive Polynomial”. The function places the file, called `userGftable.mat`, in your current working directory. 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)`.

```
% 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`, “Speed and Nondefault Primitive Polynomials”

Purpose Minimize length of polynomial representation

Syntax `c = gftrunc(a)`

Description `c = gftrunc(a)` truncates a row vector, `a`, that gives the coefficients of a GF(`p`) polynomial in order of ascending powers. If `a(k) = 0` whenever `k > d + 1`, the polynomial has degree `d`. The row vector `c` omits these high-order zeros and thus has length `d + 1`.

Examples In the code below, zeros are removed from the end, but *not* from the beginning or middle, of the row-vector representation of $x^2 + 2x^3 + 3x^4 + 4x^7 + 5x^8$.

```
c = gftrunc([0 0 1 2 3 0 0 4 5 0 0])
```

```
c =
```

```
0 0 1 2 3 0 0 4 5
```

See Also `gfadd`, `gfsub`, `gfconv`, `gfdeconv`, `gftuple`, “Galois Fields of Odd Characteristic” on page 1-13

gftuple

Purpose Simplify or convert Galois field element formatting

Syntax

```
tp = gftuple(a,m)
tp = gftuple(a,prim_poly)
tp = gftuple(a,m,p)
tp = gftuple(a,prim_poly,p)
tp = gftuple(a,prim_poly,p,prim_ck)
[tp,expform] = gftuple(...)
```

Description

Note This function performs computations in $GF(p^m)$, where p is prime. To perform equivalent computations in $GF(2^m)$, apply the `.` operator and the `log` function to Galois arrays. For more information, see “Example: Exponentiation” and “Example: Elementwise Logarithm”.

For All Syntaxes

`gftuple` serves to simplify the polynomial or exponential format of Galois field elements, or to convert from one format to another. For an explanation of the formats that `gftuple` uses, see “Representing Elements of Galois Fields”.

In this discussion, the format of an element of $GF(p^m)$ is called “simplest” if all exponents of the primitive element are

- Between 0 and $m-1$ for the polynomial format
- Either `-Inf`, or between 0 and p^{m-2} , for the exponential format

For all syntaxes, `a` is a matrix, each row of which represents an element of a Galois field. The format of `a` determines how MATLAB interprets it:

- If `a` is a column of integers, MATLAB interprets each row as an *exponential* format of an element. Negative integers are equivalent to `-Inf` in that they all represent the zero element of the field.

- If a has more than one column, MATLAB interprets each row as a *polynomial* format of an element. (Each entry of a must be an integer between 0 and $p-1$.)

The exponential or polynomial formats mentioned above are all relative to a primitive element specified by the *second* input argument. The second argument is described below.

For Specific Syntaxes

`tp = gftuple(a,m)` returns the simplest polynomial format of the elements that a represents, where the k th row of tp corresponds to the k th row of a . The formats are relative to a root of the default primitive polynomial for $GF(2^m)$, where m is a positive integer.

`tp = gftuple(a,prim_poly)` is the same as the syntax above, except that `prim_poly` is a row vector that lists the coefficients of a degree m primitive polynomial for $GF(2^m)$ in order of ascending exponents.

`tp = gftuple(a,m,p)` is the same as `tp = gftuple(a,m)` except that 2 is replaced by a prime number p .

`tp = gftuple(a,prim_poly,p)` is the same as `tp = gftuple(a,prim_poly)` except that 2 is replaced by a prime number p .

`tp = gftuple(a,prim_poly,p,prim_ck)` is the same as `tp = gftuple(a,prim_poly,p)` except that `gftuple` checks whether `prim_poly` represents a polynomial that is indeed primitive. If not, then `gftuple` generates an error and tp is not returned. The input argument `prim_ck` can be any number or string; only its existence matters.

`[tp,expform] = gftuple(...)` returns the additional matrix `expform`. The k th row of `expform` is the simplest exponential format of the element that the k th row of a represents. All other features are as described in earlier parts of this “Description” section, depending on the input arguments.

Examples

Some examples are in these subsections of “Galois Fields of Odd Characteristic” on page 1-13

- “List of All Elements of a Galois Field” (end of section)
- “Converting to Simplest Polynomial Format”
- “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.

$$\alpha^0 = 1 + 0\alpha + 0\alpha^2$$

$$\alpha^1 = 0 + 1\alpha + 0\alpha^2$$

$$0 + 0\alpha + 0\alpha^2 + \alpha^3 = 2 + \alpha + 0\alpha^2$$

Algorithm

gftuple uses recursive callbacks to determine the exponential format.

See Also

gfadd, gfmul, gfconv, gfddiv, gfdeconv, gfprimdf, “Galois Fields of Odd Characteristic” on page 1-13

gfweight

Purpose Calculate minimum distance of linear block code

Syntax

```
wt = gfweight(genmat)
wt = gfweight(genmat,'gen')
wt = gfweight(parmat,'par')
wt = gfweight(genpoly,n)
```

Description The minimum distance, or minimum weight, of a linear block code is defined as the smallest positive number of nonzero entries in any n-tuple that is a codeword.

`wt = gfweight(genmat)` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(genmat,'gen')` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(parmat,'par')` returns the minimum distance of the linear block code whose parity-check matrix is `parmat`.

`wt = gfweight(genpoly,n)` returns the minimum distance of the *cyclic* code whose codeword length is `n` and whose generator polynomial is represented by `genpoly`. `genpoly` is a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples The commands below illustrate three different ways to compute the minimum distance of a (7,4) cyclic code.

```
n = 7;
% Generator polynomial of (7,4) cyclic code
genpoly = cyclpoly(n,4);
[parmat, genmat] = cyclgen(n,genpoly);
wts = [gfweight(genmat,'gen'),gfweight(parmat,'par'),...
       gfweight(genpoly,n)]
```

The output is

wts =

3 3 3

See Also

hammgen, cyclpoly, bchgenpoly, "Block Coding"

gray2bin

Purpose Convert Gray-encoded positive integers to corresponding Gray-decoded integers

Syntax `y = gray2bin(x,modulation,M)`
`[y,map] = gray2bin(x,modulation,M)`

Description `y = gray2bin(x,modulation,M)` generates a Gray-decoded output vector or matrix `y` with the same dimensions as its input parameter `x`. `x` can be a scalar, vector, or matrix. `modulation` is the modulation type and must be a string equal to 'qam', 'pam', 'fsk', 'dpsk', or 'psk'. `M` is the modulation order that can be an integer power of 2.

`[y,map] = gray2bin(x,modulation,M)` generates a Gray-decoded output `y` with its respective Gray-encoded constellation map, `map`.

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

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

Examples

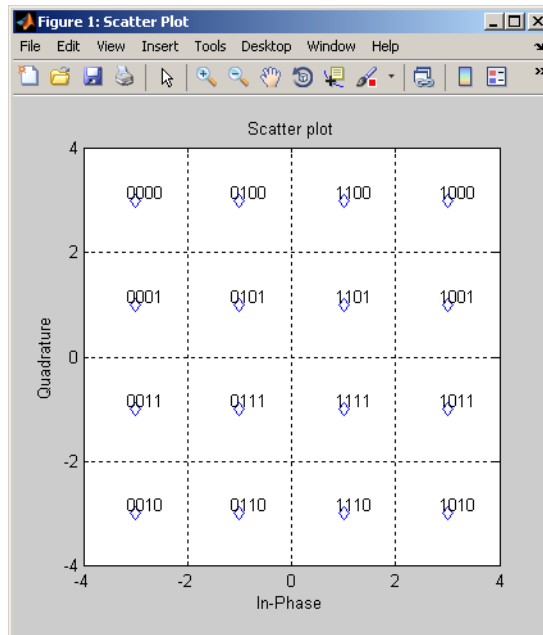
```
% To Gray decode a vector x with a 16-QAM Gray encoded
% constellation and return its map, use:
x=randint(1,100,16);
[y,map] = gray2bin(x,'qam',16);
% Obtain the symbols for 16-QAM
hMod = modem.qammod('M', 16);
symbols = hMod.Constellation;
% Plot the constellation
scatterplot(symbols);
set(get(gca,'Children'),'Marker','d','MarkerFaceColor','auto');
hold on;
```

```

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

```

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



See Also

bin2gray

hammgen

Purpose Produce parity-check and generator matrices for Hamming code

Syntax

```
h = hammgen(m)
h = hammgen(m,pol)
[h,g] = hammgen(...)
[h,g,n,k] = hammgen(...)
```

Description For all syntaxes, the codeword length is n . n has the form 2^m-1 for some positive integer m greater than or equal to 3. The message length, k , has the form $n-m$.

`h = hammgen(m)` produces an m -by- n parity-check matrix for a Hamming code having codeword length $n = 2^m-1$. The input m is a positive integer greater than or equal to 3. The message length of the code is $n-m$. The binary primitive polynomial used to produce the Hamming code is the default primitive polynomial for $GF(2^m)$, represented by `gfprimdf(m)`.

`h = hammgen(m,pol)` produces an m -by- n parity-check matrix for a Hamming code having codeword length $n = 2^m-1$. The input m is a positive integer greater than or equal to 3. The message length of the code is $n-m$. `pol` is a row vector that gives the coefficients, in order of ascending powers, of the binary primitive polynomial for $GF(2^m)$ that is used to produce the Hamming code. `hammgen` produces an error if `pol` represents a polynomial that is not, in fact, primitive.

`[h,g] = hammgen(...)` is the same as `h = hammgen(...)` except that it also produces the k -by- n generator matrix `g` that corresponds to the parity-check matrix `h`. k , the message length, equals $n-m$, or 2^m-1-m .

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

Note If your value of m is less than 25 and if your primitive polynomial is the default primitive polynomial for $GF(2^m)$, the syntax `hammgen(m)` is likely to be faster than the syntax `hammgen(m,pol)`.

Examples

The command below exhibits the parity-check and generator matrices for a Hamming code with codeword length $7 = 2^3 - 1$ and message length $4 = 7 - 3$.

```
[h,g,n,k] = hammgen(3)
```

h =

```

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

g =

```

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

n =

7

k =

4

The command below, which uses $1 + x^2 + x^3$ as the primitive polynomial for $GF(2^3)$, shows that the parity-check matrix depends on the choice of primitive polynomial. Notice that h1 below is different from h in the example above.

```
h1 = hammgen(3,[1 0 1 1])
```

hammgen

h1 =

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

Algorithm

Unlike `gftuple`, which processes one m -tuple at a time, `hammgen` generates the entire sequence from 0 to $2^m - 1$. The computation algorithm uses all previously computed values to produce the computation result.

See Also

`encode`, `decode`, `gen2par`, “Block Coding”

Purpose

Convert Hankel matrix to linear system model

Syntax

```
[num,den] = hank2sys(h,ini,tol)
[num,den,sv] = hank2sys(h,ini,tol)
[a,b,c,d] = hank2sys(h,ini,tol)
[a,b,c,d,sv] = hank2sys(h,ini,tol)
```

Description

`[num,den] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a linear system transfer function with numerator `num` and denominator `den`. The vectors `num` and `den` list the coefficients of their respective polynomials in ascending order of powers of z^{-1} . The argument `ini` is the system impulse at time zero. If `tol > 1`, `tol` is the order of the conversion. If `tol < 1`, `tol` is the tolerance in selecting the conversion order based on the singular values. If you omit `tol`, its default value is 0.01. This conversion uses the singular value decomposition method.

`[num,den,sv] = hank2sys(h,ini,tol)` returns a vector `sv` that lists the singular values of `h`.

`[a,b,c,d] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a corresponding linear system state-space model. `a`, `b`, `c`, and `d` are matrices. The input parameters are the same as in the first syntax above.

`[a,b,c,d,sv] = hank2sys(h,ini,tol)` is the same as the syntax above, except that `sv` is a vector that lists the singular values of `h`.

Examples

```
h = hankel([1 0 1]);
[num,den,sv] = hank2sys(h,0,.01)
```

The output is

```
num =
      0      1.0000      0.0000      1.0000

den =
```

hank2sys

1.0000 0.0000 0.0000 0.0000

sv =

1.6180

1.0000

0.6180

See Also

rcosflt, hankel

Purpose

Restore ordering of symbols permuted using helintrlv

Syntax

```
[deintrlved,state] = heldeintrlv(data,col,ngroup,step)
[deintrlved,state] = heldeintrlv(data,col,ngroup,step,
    init_state)
deintrlved = heldeintrlv(data,col,ngroup,step,init_state)
```

Description

[deintrlved,state] = heldeintrlv(data,col,ngroup,step) 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*ngroup elements. If data is a matrix with multiple rows and columns, it must have col*ngroup 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*ngroup symbols from the input into the next ngroup rows of the array. The function places symbols from the array in the output, intrlved, placing ngroup symbols at a time; the kth group of ngroup symbols comes from the kth column of the array, starting from row 1+(k-1)*step. 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,ngroup,step,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,ngroup,step,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 $\text{col} * \text{ngrp} * \text{ceil}(\text{stp} * (\text{col} - 1) / \text{ngrp})$ 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

The example below illustrates how to 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 = randint(len,1,10); % 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,ngroup,step);
[d2,dstate] = heldeintrlv(i2,col,ngroup,step,dstate);
d3 = heldeintrlv(i3,col,ngroup,step,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, “Interleaving”

helintrlv

Purpose Permute symbols using helical array

Syntax

```
intrlv = helintrlv(data,col,ngroup,step)
[intrlv,state] = helintrlv(data,col,ngroup,step)
[intrlv,state] = helintrlv(data,col,ngroup,step,init_state)
```

Description `intrlv = helintrlv(data,col,ngroup,step)` 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, `intrlv`. `data` must have `col*ngroup` elements. If `data` is a matrix with multiple rows and columns, it must have `col*ngroup` 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*ngroup` symbols from the input into consecutive groups of `ngroup` symbols. The function places the `k`th group in the array along column `k`, starting from row `1+(k-1)*step`. Positions in the array that do not contain input symbols have default values of 0. The function places `col*ngroup` symbols from the array in the output, `intrlv`, by reading the first `ngroup` 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.

`[intrlv,state] = helintrlv(data,col,ngroup,step)` 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.

`[intrlv,state] = helintrlv(data,col,ngroup,step,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.

helintrlv

The output from this example is below. (The actual interleaved data is a tall matrix, but it has been transposed into a wide matrix for display purposes.) The interleaved data comes from the top four rows of the three-column arrays shown above. Notice that some of the symbols in the first half of the interleaved data are default values of 0, some of the symbols in the second half of the interleaved data were left in the array from the first call to `helintrlv`, and some of the input symbols (20, 23, and 24) do not appear in the interleaved data at all.

Interleaved data:

Columns 1 through 10

1	0	0	2	5	0	3	6	9	4
13	8	11	14	17	12	15	18	21	16

Columns 11 through 12

7	10
19	22

Values left in array after first interleaving operation:

ans =

[]

ans =

8

ans =

11 12

The example on the reference page for `heldeintrlv` also uses this function.

See Also

`heldeintrlv`, “Interleaving”

helscandeintrlv

Purpose Restore ordering of symbols in helical pattern

Syntax `deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)`

Description `deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements in a helical fashion and then sending the matrix contents to the output row by row. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function places input elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `helscanintrlv` function, use the same `Nrows`, `Ncols`, and `hstep` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helscanintrlv` followed by `helscandeintrlv` leaves `data` unchanged.

Examples

The command below rearranges a vector using a 3-by-4 temporary matrix and diagonals of slope 1.

```
d = helscandeintrlv(1:12,3,4,1)
d =
```

```
Columns 1 through 10
```

```
1    10    7    4    5    2    11    8    9    6
```

Columns 11 through 12

```
3 12
```

Internally, the function creates the 3-by-4 temporary matrix

```
[1 10 7 4;  
5 2 11 8;  
9 6 3 12]
```

using length-four diagonals. The function then sends the elements, row by row, to the output `d`.

See Also

`helscanintrlv`, “Interleaving”

helscanintrlv

Purpose Reorder symbols in helical pattern

Syntax `intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)`

Description `intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents to the output in a helical fashion. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function selects elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples The command below rearranges a vector using diagonals of two different slopes.

```
i1 = helscanintrlv(1:12,3,4,1) % Slope of diagonal is 1.  
i2 = helscanintrlv(1:12,3,4,2) % Slope of diagonal is 2.
```

The output is below.

```
i1 =  
  
Columns 1 through 10  
    1     6    11     4     5    10     3     8     9    2  
  
Columns 11 through 12
```

```
7 12
```

```
i2 =
```

```
Columns 1 through 10
```

```
1 10 7 4 5 2 11 8 9 6
```

```
Columns 11 through 12
```

```
3 12
```

In each case, the function internally creates the temporary 3-by-4 matrix

```
[1 2 3 4;
 5 6 7 8;
 9 10 11 12]
```

To form `i1`, the function forms each slope-one diagonal by moving one row down and one column to the right. The first diagonal contains 1, 6, 11, and 4, while the second diagonal starts with 5 because that is beneath 1 in the temporary matrix.

To form `i2`, the function forms each slope-two diagonal by moving two rows down and one column to the right. The first diagonal contains 1, 10, 7, and 4, while the second diagonal starts with 5 because that is beneath 1 in the temporary matrix.

See Also

`helscandeintrlv`, “Interleaving”

hilbiir

Purpose Design Hilbert transform IIR filter

Syntax

```
hilbiir
hilbiir(ts)
hilbiir(ts,dly)
hilbiir(ts,dly,bandwidth)
hilbiir(ts,dly,bandwidth,tol)
[num,den] = hilbiir(...)
[num,den,sv] = hilbiir(...)
[a,b,c,d] = hilbiir(...)
[a,b,c,d,sv] = hilbiir(...)
```

Description The function `hilbiir` designs a Hilbert transform filter. The output is either

- A plot of the filter's impulse response, or
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Background Information

An ideal Hilbert transform filter has the transfer function $H(s) = -j\text{sgn}(s)$, where $\text{sgn}(\cdot)$ 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

$$h(t) = \frac{1}{\pi(t - \text{dly})}$$

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 $\text{rem}(dly, ts) = ts/2$. 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, $-\text{Inf}$, 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 2-345 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 $\text{bandwidth} = 0$ or $\text{bandwidth} > 1/(2 \cdot \text{ts})$, `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 $\text{tol} < 1$, the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < \text{tol}$$

If $\text{tol} > 1$, 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 2-345.

`[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.

Algorithm

The `hilbiir` function calculates the impulse response of the ideal Hilbert transform filter response with a group delay. It fits the response curve using a singular-value decomposition method. See the book by Kailath [1].

Examples

For an example using the function's default values, type one of the following commands at the MATLAB prompt.

```
hilbiir  
[num,den] = hilbiir
```

See Also

grpdelay, rcosiir, "Special Filters"

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1980.

huffmandeco

Purpose Huffman decoder

Syntax `dsig = huffmandeco(comp,dict)`

Description `dsig = huffmandeco(comp,dict)` decodes the numeric Huffman code vector `comp` using the code dictionary `dict`. The argument `dict` is an N-by-2 cell array, where N is the number of distinct possible symbols in the original signal that was encoded as `comp`. The first column of `dict` represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` is allowed to be the prefix of any other codeword in `dict`. You can generate `dict` using the `huffmandict` function and `comp` using the `huffmanenco` function. If all signal values in `dict` are numeric, `dsig` is a vector; if any signal value in `dict` is alphabetical, `dsig` is a one-dimensional cell array.

Examples The example below encodes and then decodes a vector of random data that has a prescribed probability distribution.

```
symbols = [1:6]; % Distinct symbols that data source can produce
p = [.5 .125 .125 .125 .0625 .0625]; % Probability distribution
[dict,avglen] = huffmandict(symbols,p); % Create dictionary.
actualsig = randsrc(1,100,[symbols; p]); % Create data using p.
comp = huffmanenco(actualsig,dict); % Encode the data.
dsig = huffmandeco(comp,dict); % Decode the Huffman code.
isequal(actualsig,dsig) % Check whether the decoding is correct.
```

The output below indicates that the decoder successfully recovered the data in `actualsig`.

```
ans =
     1
```

See Also `huffmandict`, `huffmanenco`, “Huffman Coding”

References

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

huffmandict

Purpose Generate Huffman code dictionary for source with known probability model

Syntax

```
[dict,avglen] = huffmandict(symbols,p)
[dict,avglen] = huffmandict(symbols,p,N)
[dict,avglen] = huffmandict(symbols,p,N,variance)
```

Description **For All Syntaxes**

The `huffmandict` function generates a Huffman code dictionary corresponding to a source with a known probability model. The required inputs are

- `symbols`, which lists the distinct signal values that the source produces. It can have the form of a numeric vector, numeric cell array, or alphanumeric cell array. If it is a cell array, it must be either a row or a column.
- `p`, a probability vector whose k th element is the probability with which the source produces the k th element of `symbols`. The length of `p` must equal the length of `symbols`.

The outputs of `huffmandict` are

- `dict`, a two-column cell array in which the first column lists the distinct signal values from `symbols` and the second column lists the corresponding Huffman codewords. In the second column, each Huffman codeword is represented as a numeric row vector.
- `avglen`, the average length among all codewords in the dictionary, weighted according to the probabilities in the vector `p`.

For Specific Syntaxes

`[dict,avglen] = huffmandict(symbols,p)` generates a binary Huffman code dictionary using the maximum variance algorithm.

`[dict,avglen] = huffmandict(symbols,p,N)` generates an N -ary Huffman code dictionary using the maximum variance algorithm. N is

an integer between 2 and 10 that must not exceed the number of source symbols whose probabilities appear in the vector *p*.

`[dict,avglen] = huffmandict(symbols,p,N,variance)` generates an N-ary Huffman code dictionary with the minimum variance if *variance* is 'min' and the maximum variance if *variance* is 'max'. N is an integer between 2 and 10 that must not exceed the length of the vector *p*.

Examples

```
symbols = [1:5];  
p = [.3 .3 .2 .1 .1];  
[dict,avglen] = huffmandict(symbols,p)  
samplecode = dict{5,2} % Codeword for fifth signal value
```

The output is below, where the first column of `dict` lists the values in `symbols` and the second column lists the corresponding codewords.

```
dict =  
  
[1]    [1x2 double]  
[2]    [1x2 double]  
[3]    [1x2 double]  
[4]    [1x3 double]  
[5]    [1x3 double]
```

```
avglen =  
  
2.2000
```

```
samplecode =  
  
1    1    0
```

See Also

`huffmanenco`, `huffmandeco`, “Huffman Coding”

References

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

Purpose Huffman encoder

Syntax `comp = huffmanenco(sig,dict)`

Description `comp = huffmanenco(sig,dict)` encodes the signal `sig` using the Huffman codes described by the code dictionary `dict`. The argument `sig` can have the form of a numeric vector, numeric cell array, or alphanumeric cell array. If `sig` is a cell array, it must be either a row or a column. `dict` is an N-by-2 cell array, where N is the number of distinct possible symbols to be encoded. The first column of `dict` represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` can be the prefix of any other codeword in `dict`. You can generate `dict` using the `huffmandict` function.

Examples The example below encodes a vector of random data that has a prescribed probability distribution.

```
symbols = [1:6]; % Distinct symbols that data source can produce
p = [.5 .125 .125 .125 .0625 .0625]; % Probability distribution
[dict,avglen] = huffmandict(symbols,p); % Create dictionary.
actualsig = randsrc(100,1,[symbols; p]); % Create data using p.
comp = huffmanenco(actualsig,dict); % Encode the data.
```

See Also `huffmandict`, `huffmandeco`, “Huffman Coding”

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

ifft

Purpose	Inverse discrete Fourier transform
Syntax	<code>ifft(x)</code>
Description	<code>ifft(x)</code> 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 <code>ifft</code> , see the reference page for <code>fft</code> .
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.
Algorithm	If x is a column vector, <code>ifft</code> applies <code>dftmtx</code> to the multiplicative inverse of the primitive element of the Galois field and multiplies the resulting matrix by x .
See Also	<code>fft</code> , <code>dftmtx</code> , “Signal Processing Operations in Galois Fields”

Purpose Integrate and dump

Syntax `y = intdump(x,nsamp)`

Description `y = intdump(x,nsamp)` integrates the signal `x` over a symbol period and outputs one value for that symbol period. A symbol period consists of `nsamp` samples. If `x` contains multiple symbols, the function processes the symbols independently. If `x` is a matrix with multiple rows, the function treats each column as a channel and processes the columns independently.

Examples An example in “Combining 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. Each symbol contains four samples.

```
nsamp = 4; % Number of samples per symbol
ch1 = randint(3*nsamp,1,2,68521); % Random binary channel
ch2 = rectpulse([1 2 3]',nsamp); % Rectangular pulses
x = [ch1 ch2]; % Two-channel signal
y = intdump(x,nsamp)
```

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`

intrlv

Purpose Reorder sequence of symbols

Syntax `intrlvd = intrlv(data,elements)`

Description `intrlvd = intrlv(data,elements)` rearranges the elements of `data` without repeating or omitting any elements. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. The sequence in `elements` is the sequence in which elements from `data` or its columns appear in `intrlvd`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

Examples The command below rearranges the elements of a vector. Your output might differ because the permutation vector is random in this example.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p)
```

The output is below.

```
a =
    10    90    60    30    50    80   100    20    70    40
```

The command below rearranges each of two columns of a matrix.

```
b = intrlv([.1 .2 .3 .4 .5; .2 .4 .6 .8 1]',[2 4 3 5 1])
b =
    0.2000    0.4000
    0.4000    0.8000
    0.3000    0.6000
    0.5000    1.0000
    0.1000    0.2000
```

See Also `deintrlv`, “Interleaving”

Purpose	True for trellis corresponding to catastrophic convolutional code
Syntax	<code>iscatastrophic(s)</code>
Description	<code>iscatastrophic(s)</code> returns true if the trellis <code>s</code> corresponds to a convolutional code that causes catastrophic error propagation. Otherwise, it returns false.
See Also	<code>convenc</code> , <code>istrellis</code> , <code>poly2trellis</code> , <code>struct</code> , “Convolutional Coding”

isprimitive

Purpose True for primitive polynomial for Galois field

Syntax `isprimitive(a)`

Description `isprimitive(a)` returns 1 if the polynomial that `a` represents is primitive for the Galois field $GF(2^m)$, and 0 otherwise. The input `a` can represent the polynomial using one of these formats:

- A nonnegative integer less than 2^{17} . The binary representation of this integer indicates the coefficients of the polynomial. In this case, `m` is `floor(log2(a))`.
- A Galois row vector in $GF(2)$, listing the coefficients of the polynomial in order of descending powers. In this case, `m` is the order of the polynomial represented by `a`.

Examples

The example below finds all primitive polynomials for $GF(8)$ and then checks using `isprimitive` whether specific polynomials are primitive.

```
a = primpoly(3,'all','nodisplay'); % All primitive polys for GF(8)

isp1 = isprimitive(13) % 13 represents a primitive polynomial.

isp2 = isprimitive(14) % 14 represents a nonprimitive polynomial.
```

The output is below. If you examine the vector `a`, notice that `isp1` is true because 13 is an element in `a`, while `isp2` is false because 14 is not an element in `a`.

```
isp1 =
      1

isp2 =
      0
```

See Also `primpoly`, “Galois Field Computations”

istrellis

Purpose True for valid trellis structure

Syntax `[isok,status] = istrellis(s)`

Description `[isok,status] = istrellis(s)` checks if the input `s` is a valid trellis structure. If the input is a valid trellis structure, `isok` is 1 and `status` is an empty string. Otherwise, `isok` is 0 and `status` is a string that indicates why `s` is not a valid trellis structure.

A valid trellis structure is a MATLAB structure whose fields are as in the table below.

Fields of a Valid Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
<code>numInputSymbols</code>	Scalar	Number of input symbols to the encoder: 2^k
<code>numOutputSymbols</code>	Scalar	Number of output symbols from the encoder: 2^n
<code>numStates</code>	Scalar	Number of states in the encoder
<code>nextStates</code>	<code>numStates-by-2^k</code> matrix	Next states for all combinations of current state and current input
<code>outputs</code>	<code>numStates-by-2^k</code> 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 `s`th row and `u`th 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 $\{0, \dots, 0, 1\}$.

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 `s`th row and `u`th 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 =
```

istrellis

..

Another example of a trellis is in “Trellis Description of a Convolutional Encoder”.

See Also

poly2trellis, struct, convenc, vitdec, “Convolutional Coding”

Purpose Construct linear equalizer object

Syntax

```
eqobj = lineareq(nweights,alg)
eqobj = lineareq(nweights,alg,sigconst)
eqobj = lineareq(nweights,alg,sigconst,nsamp)
```

Description The `lineareq` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Using Adaptive Equalizer Functions and Objects”.

`eqobj = lineareq(nweights,alg)` constructs a symbol-spaced linear equalizer object. The equalizer has `nweights` complex weights, which are initially all zeros. `alg` describes the adaptive algorithm that the equalizer uses; you should create `alg` using any of these functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`. The signal constellation of the desired output is `[-1 1]`, which corresponds to binary phase shift keying (BPSK).

`eqobj = lineareq(nweights,alg,sigconst)` specifies the signal constellation vector of the desired output.

`eqobj = lineareq(nweights,alg,sigconst,nsamp)` constructs a fractionally spaced linear equalizer object. The equalizer has `nweights` complex weights spaced at $T/nsamp$, where T is the symbol period and `nsamp` is a positive integer. `nsamp = 1` corresponds to a symbol-spaced equalizer.

Properties

The table below describes the properties of the linear equalizer object. To learn how to view or change the values of a linear equalizer object, see “Accessing Properties of an Equalizer”.

Tip To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
EqType	Fixed value, 'Linear Equalizer'
AlgType	Name of the adaptive algorithm represented by alg
nWeights	Number of weights
nSampPerSym	Number of input samples per symbol (equivalent to nsamp input argument). This value relates to both the equalizer structure (see the use of K in “Fractionally Spaced Equalizers”) and an assumption about the signal to be equalized.
RefTap (except for CMA equalizers)	Reference tap index, between 1 and nWeights. Setting this to a value greater than 1 effectively delays the reference signal and the output signal by RefTap-1 with respect to the equalizer’s input signal.
SigConst	Signal constellation, a vector whose length is typically a power of 2
Weights	Vector of complex coefficients. This is the set of w_i values in the schematic in “Symbol-Spaced Equalizers”.
WeightInputs	Vector of tap weight inputs. This is the set of u_i values in the schematic in “Symbol-Spaced Equalizers”.

Property	Description
ResetBeforeFiltering	If 1, each call to <code>equalize</code> resets the state of <code>eqobj</code> 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 <code>eqobj</code> , this property value is 0.
Properties specific to the adaptive algorithm represented by <code>alg</code>	See reference page for the adaptive algorithm function that created <code>alg</code> : <code>lms</code> , <code>signlms</code> , <code>normlms</code> , <code>varlms</code> , <code>rls</code> , or <code>cma</code> .

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
<code>Weights</code>	<code>zeros(1,nWeights)</code>
<code>WeightInputs</code>	<code>zeros(1,nWeights)</code>
<code>StepSize</code> (Variable-step-size LMS equalizers)	<code>InitStep*ones(1,nWeights)</code>
<code>InvCorrMatrix</code> (RLS equalizers)	<code>InvCorrInit*eye(nWeights)</code>

An example illustrating relationships among properties is in “Linked Properties of an Equalizer Object”.

lineareq

Examples

For examples that use this function, see “Equalizing Using a Training Sequence”, “Example: Equalizing Multiple Times, Varying the Mode”, and “Example: Adaptive Equalization Within a Loop”.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `dfe`, `equalize`, “Equalizers”

Purpose

Optimize quantization parameters using Lloyd algorithm

Syntax

```
[partition,codebook] = lloyds(training_set,initcodebook)
[partition,codebook] = lloyds(training_set,len)
[partition,codebook] = lloyds(training_set,...,tol)
[partition,codebook,distor] = lloyds(...)
[partition,codebook,distor,reldistor] = lloyds(...)
```

Description

`[partition,codebook] = lloyds(training_set,initcodebook)` optimizes the scalar quantization parameters `partition` and `codebook` for the training data in the vector `training_set`. `initcodebook`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `initcodebook`. The output `partition` is a vector whose length is one less than the length of `codebook`.

See “Representing Partitions”, “Representing 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
```

Algorithm

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 $\text{eps} * \max(\text{training_set})$, where eps is the MATLAB floating-point relative accuracy.

See Also

quantiz, dpcmopt, “Source Coding”

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.

Ims

Purpose Construct least mean square (LMS) adaptive algorithm object

Syntax

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

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

`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 “Accessing Properties of an Adaptive Algorithm”.

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

Examples For examples that use this function, see “Equalizing Using a Training Sequence”, “Example: Equalizing Multiple Times, Varying the Mode”, and “Example: Adaptive Equalization Within a Loop”.

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

$$(\text{LeakageFactor}) w + (\text{StepSize}) u^*$$

where the $*$ operator denotes the complex conjugate.

See Also

signlms, normlms, varlms, rls, cma, lineareq, dfe, equalize, “Equalizers”

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.

log

Purpose Logarithm in Galois field

Syntax `y = log(x)`

Description `y = log(x)` computes the logarithm of each element in the Galois array `x`. `y` is an integer array that solves the equation $A.^y = x$, where `A` is the primitive element used to represent elements in `x`. More explicitly, the base `A` of the logarithm is `gf(2,x.m)` or `gf(2,x.m,x.prim_poly)`. All elements in `x` must be nonzero because the logarithm of zero is undefined.

Examples The code below illustrates how the logarithm operation inverts exponentiation.

```
m = 4; x = gf([8 1 6; 3 5 7; 4 9 2],m);
y = log(x);
primel = gf(2,m); % Primitive element in the field
z = primel.^y; % This is now the same as x.
ck = isequal(x,z)
```

The output is

```
ck =
```

```
1
```

The code below shows that the logarithm of 1 is 0 and that the logarithm of the base (`primel`) is 1.

```
m = 4; primel = gf(2,m);
yy = log([1, primel])
```

The output is

```
yy =
```

```
0    1
```

Purpose Generalized Marcum Q function

Syntax
 $Q = \text{marcumq}(a, b)$
 $Q = \text{marcumq}(a, b, m)$

Description $Q = \text{marcumq}(a, b)$ 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.

$Q = \text{marcumq}(a, b, m)$ computes the generalized Marcum Q, defined by

$$Q(a, b) = \frac{1}{a^{m-1}} \int_b^{\infty} x^m \exp\left(-\frac{x^2 + a^2}{2}\right) I_{m-1}(ax) dx$$

where a and b are nonnegative real numbers, and m is a positive integer. In this expression, I_{m-1} is the modified Bessel function of the first kind of order $m-1$.

If any of the inputs is a scalar, it is expanded to the size of the other inputs.

See Also `besseli`

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.

Purpose	Convert mask vector to shift for shift register configuration
Syntax	<code>shift = mask2shift(prpoly,mask)</code>
Description	<p><code>shift = mask2shift(prpoly,mask)</code> returns the shift that is equivalent to a mask, for a linear feedback shift register whose connections are specified by the primitive polynomial <code>prpoly</code>. The <code>prpoly</code> input can have one of these formats:</p> <ul style="list-style-type: none">• 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 <p>The <code>mask</code> input is a binary vector whose length is the degree of the primitive polynomial.</p>

Note To save time, `mask2shift` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

For more information about how masks and shifts are related to pseudonoise sequence generators, see `shift2mask`.

Definition of Equivalent Shift

If A is a root of the primitive polynomial and $m(A)$ is the mask polynomial evaluated at A , the equivalent shift s solves the equation $A^s = m(A)$. To interpret the vector `mask` as a polynomial, treat `mask` as a list of coefficients in order of descending powers.

Examples	The first command below converts a mask of $x^3 + 1$ into an equivalent shift for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$. The second command
-----------------	---

mask2shift

shows that a mask of 1 is equivalent to a shift of 0. In both cases, notice that the length of the mask vector is one less than the length of the prpoly vector.

```
s = mask2shift([1 1 0 0 1],[1 0 0 1])
s2 = mask2shift([1 1 0 0 1],[0 0 0 1])
```

The output is below.

```
s =
```

```
4
```

```
s2 =
```

```
0
```

See Also

shift2mask, log, isprimitive, primpoly

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.

Purpose Restore ordering of symbols by filling matrix by columns and emptying it by rows

Syntax `deintrlvd = matdeintrlv(data,Nrows,Ncols)`

Description `deintrlvd = matdeintrlv(data,Nrows,Ncols)` rearranges the elements in `data` by filling a temporary matrix with the elements column by column and then sending the matrix contents, row by row, to the output. `Nrows` and `Ncols` are the dimensions of the temporary matrix. If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `matintrlv` function, use the same `Nrows` and `Ncols` inputs in both functions. In that case, the two functions are inverses in the sense that applying `matintrlv` followed by `matdeintrlv` leaves `data` unchanged.

Examples The code below illustrates the inverse relationship between `matintrlv` and `matdeintrlv`.

```
Nrows = 2; Ncols = 3;
data = [1 2 3 4 5 6; 2 4 6 8 10 12]';
a = matintrlv(data,Nrows,Ncols); % Interleave.
b = matdeintrlv(a,Nrows,Ncols) % Deinterleave.
```

The output below shows that `b` is the same as `data`.

```
b =

     1     2
     2     4
     3     6
     4     8
     5    10
     6    12
```

matdeintrlv

See Also

matintrlv, “Interleaving”

Purpose Reorder symbols by filling matrix by rows and emptying it by columns

Syntax `intrlvd = matintrlv(data,Nrows,Ncols)`

Description `intrlvd = matintrlv(data,Nrows,Ncols)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents, column by column, to the output. `Nrows` and `Ncols` are the dimensions of the temporary matrix. If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples The command below rearranges each of two columns of a matrix.

```
b = matintrlv([1 2 3 4 5 6; 2 4 6 8 10 12]',2,3)
b =
```

```
1     2
4     8
2     4
5    10
3     6
6    12
```

To form the first column of the output, the function creates the temporary 2-by-3 matrix `[1 2 3; 4 5 6]`. Then the function reads down each column of the temporary matrix to get `[1 4 2 5 3 6]`.

See Also `matdeintrlv`, “Interleaving”

mimochan

Purpose Create MIMO fading channel object

Syntax

```
chan = mimochan(nt, nr, ts, fd)
chan = mimochan(nt, nr, ts, fd, tau)
chan = mimochan(nt, nr, ts, fd, tau, pdb)
```

Description `chan = mimochan(nt, nr, ts, fd)` constructs a multiple-input multiple-output (MIMO) Rayleigh fading channel object with a single path link.

- *nt* is the number of transmit antennas.
- *nr* is the number of receive antennas.
- *nt* and *nr* can be integer values from 1 to 8.
- *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, *chan*, on a signal *x* by using the syntax `y = filter(chan, x)`, where the input signal *x* has *nt* columns, and the output signal *y* has *nr* columns. See `filter (mimo)` for more information.

`chan = mimochan(nt, nr, ts, fd, tau)` constructs a MIMO fading channel object with a frequency selective multipath link that models each discrete path as an independent Rayleigh fading process with the same average gain. *tau* represents a row vector of path delays, each specified in seconds

`chan = mimochan(nt, nr, ts, fd, tau, pdb)` specifies *pdb* as a row vector of average path gains, each in dB.

Properties A MIMO fading channel object has the properties shown in the following table. You can write to all properties except for the ones explicitly noted otherwise.

Property	Description
NumTxAntennas	Number of transmit antennas, between 1 and 8.
NumRxAntennas	Number of receive antennas, between 1 and 8.
InputSamplePeriod	Sample period of the signal on which the channel acts, measured in seconds.
DopplerSpectrum	Any Doppler spectrum objects. This property defaults to a Jakes Doppler object. Must either be a single Doppler object or a vector of Doppler objects with the same length as PathDelays.
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. This value defaults to 0.
AvgPathGaindB	Vector listing the average gain of the discrete paths, in dB. Must be of the same size as PathDelays. This value defaults to 0.
TxCorrelationMatrix	Transmit correlation matrix, hermitian, of size NumTxAntennas by NumTxAntennas (or 3-D array, with one correlation matrix per path). The default is an identity matrix.
RxCorrelationMatrix	Receive correlation matrix, hermitian, of size NumRxAntennas by NumRxAntennas (or 3-D array, with one correlation matrix per path). This value defaults to an identity matrix.

Property	Description
KFactor	Rician K-factor (scalar or vector). This value defaults to 0 (Rayleigh fading).
DirectPathDopplerShift	Any Doppler shifts of the line-of-sight components in hertz. This value defaults to 0.
DirectPathInitPhase	Any Initial phases of line-of-sight components in radians. This value defaults to 0.
ResetBeforeFiltering	If this value is 1, each call to filter resets the state of the channel object before filtering. If it is 0, the fading process maintains continuity from one call to the next. This value defaults to 1.
NormalizePathGains	If this value is 1, the fading process is normalized such that the expected value of the path gains' total power is 1. This value defaults to 1.
StorePathGains	If this value is 1, the complex path gain array is stored as the channel filter function processes the signal. This value defaults to 0.
ChannelType	Fixed value, 'MIMO'. This property is not writable.
PathGains	Complex array of size N_s by L by $NumTxAntennas$ by $NumRxAntennas$ (where N_s is the number of samples and L is the length of <i>PathDelays</i>), listing the current gains of the discrete paths for each combination of transmit/receive antennas. This property is not writable.

Property	Description
ChannelFilterDelay	Delay of the channel filter, measured in samples. This property is not writable.
NumSamplesProcessed	Number of samples the channel processed since the last reset. When you create or reset <i>chan</i> , this property value is 0. This property is not writable.

Relationships Among Properties

Changing the length of PathDelays also changes:

- the length of AvgPathGaindB
- the lengths of KFactor, DirectPathDopplerShift and DirectPathInitPhase, if KFactor is a vector (no changes if KFactor is a scalar).
- the length of DopplerSpectrum, if DopplerSpectrum is a vector (no change if DopplerSpectrum is a single object)
- the size of TxCorrelationMatrix and/or RxCorrelationMatrix, if either property is a 3-D array

Changing the length of PathDelays also changes:

- MATLAB software truncates or zero-pads the value of AvgPathGaindB to adjust its vector length
- If KFactor is a vector, MATLAB software truncates or zero-pads the value of KFactor to adjust its vector length. DirectPathDopplerShift and DirectPathInitPhase both follow changes in KFactor
- If DopplerSpectrum is a vector of Doppler objects, MATLAB software adds Jakes Doppler objects or removes elements from DopplerSpectrum, to make it the same length as PathDelays

- If `TxCorrelationMatrix` and/or `RxCorrelationMatrix` is a 3-D array, will add identity matrices or remove matrices from the 3-D array, so that the 3rd dimension equals the length of `PathDelays`
- MATLAB software may also change the values of read-only properties, such as `PathGains` and `ChannelFilterDelay`

Rayleigh and Rician Fading

If `KFactor` is a scalar value, then the first discrete path indicates a Rician fading process (it contains a line-of-sight component) with a K-factor (defined by the value for `KFactor`). The remaining discrete paths are independent Rayleigh fading processes (no line-of-sight component). If `KFactor` is a vector of the same size as `PathDelays`, then each discrete path indicates a Rician fading process with a K-factor given by the corresponding element of the vector `KFactor`.

`DirectPathDopplerShift` must be the same size as `KFactor`. If `KFactor` and `DirectPathDopplerShift` are scalar values, the line-of-sight component of the first discrete path has a Doppler shift of `DirectPathDopplerShift`, while the remaining discrete paths indicate independent Rayleigh fading processes. If `DirectPathDopplerShift` is a vector of the same size as `KFactor`, the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector `DirectPathDopplerShift`.

You can set any initial phases of the line-of-sight components through the property `DirectPathInitPhase`.

Methods

A MIMO fading channel object has the following methods.

filter

This method filters data through the MIMO fading channel object.

`y = filter(chan, x)` filters data through the MIMO channel, where the input x is a matrix of size N_s by nt . The output, y , is a matrix of size N_s by nr . For these matrices, N_s is the number of samples, nt is the number of transmit antennas, and nr is the number of receive antennas.

reset

This method resets the MIMO channel object.

`reset(chan)` resets the MIMO channel object, *chan*, initializing the `PathGains` and `NumSamplesProcessed` properties as well as internal filter states. This is useful when you want the effect of creating a new channel.

`reset(chan, randstate)` resets the MIMO channel object, *chan* and initializes the state of the random number generator, *randstate*, that the channel uses. *randstate* is a two-element column vector or a scalar integer. This is useful when you want to repeat previous numerical results that started from a particular state.

Examples

The following example creates a MIMO channel object with two transmit and receive antennas, three paths, Rician K-factor greater than zero, and specific correlation matrices. First create the channel object using the `mimochan` function, as shown below:

```
chan = mimochan(2, 2, 1e-4, 60, [0 2.5e-4 3e-4], [0 -2 -3])
```

`mimochan` returns the following channel:

```
chan =
```

```

        ChannelType: 'MIMO'
        NumTxAntennas: 2
        NumRxAntennas: 2
        InputSamplePeriod: 1.0000e-004
        DopplerSpectrum: [1x1 doppler.jakes]
        MaxDopplerShift: 60
        PathDelays: [0 2.5000e-004 3.0000e-004]
        AvgPathGaindB: [0 -2 -3]
        TxCorrelationMatrix: [2x2 double]
        RxCorrelationMatrix: [2x2 double]
        KFactor: 0
        DirectPathDopplerShift: 0
        DirectPathInitPhase: 0

```

```
ResetBeforeFiltering: 1
  NormalizePathGains: 1
    StorePathGains: 0
      PathGains: [4-D double]
    ChannelFilterDelay: 4
  NumSamplesProcessed: 0
```

Now assign the Rician K-factor to the first path, and set the transmit and receive correlation matrices of the MIMO channel object.

```
chan.KFactor = 2;
chan.TxCorrelationMatrix = [1 0.6; 0.6 1];
chan.RxCorrelationMatrix = [1 0.5*j; -0.5*j 1];
```

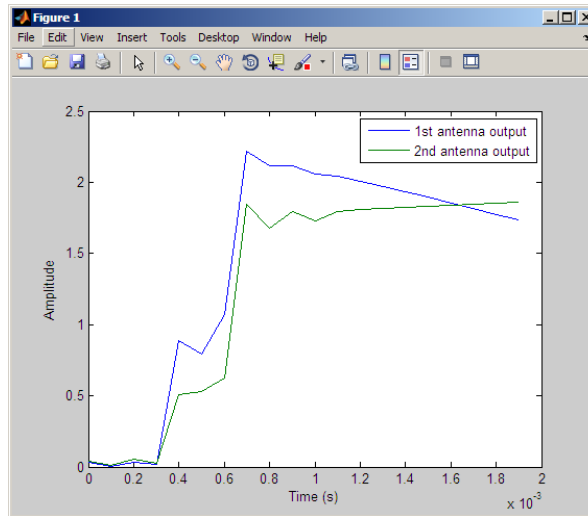
Now we can use this channel to filter data signals. This example sends a series of ones through the channel and stores the output.

```
y = filter(chan, ones(20, 2));
```

Plot the channel output signals:

```
t=(0:19)*chan.InputSamplePeriod;
plot(t,abs(y))
xlabel('Time (s)')
ylabel('Amplitude')
legend('1st antenna output','2nd antenna output')
```

The figure window displays.



See Also

doppler.bell

References

- [1] Jeruchim, M., Balaban, P., and Shanmugan, K., *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.
- [2] J. P. Kermoal, 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 Commun.*, vol. 20, no. 6, pp. 1211—1226, Aug. 2002.
- [3] C. Oestges and B. Clerckx, *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.
- [4] L. M. Correira, Ed., *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

minpol

Purpose Find minimal polynomial of Galois field element

Syntax `p1 = minpol(x)`

Description `p1 = minpol(x)` finds the minimal polynomial of each element in the Galois column vector, `x`. The output `p1` is an array in $\text{GF}(2)$. The k th row of `p1` lists the coefficients, in order of descending powers, of the minimal polynomial of the k th element of `x`.

Note The output is in $\text{GF}(2)$ 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 $\text{GF}(2^m)$. This is true for any value of m , not just the value used in the example.

```
m = 4;  
A = gf(2,m)  
p1 = 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
```

```
p1 = GF(2) array.
```

```
Array elements =
```

```
1 0 0 1 1
```

Another example is in “Minimal Polynomials”.

See Also

cosets, “Polynomials over Galois Fields”

mldivide

Purpose Matrix left division `\` of Galois arrays

Syntax `x = A\B`

Description `x = A\B` divides the Galois array `A` into `B` to produce a particular solution of the linear equation $A*x = B$. In the special case when `A` is a nonsingular square matrix, `x` is the unique solution, `inv(A)*B`, to the equation.

Examples The code below shows that `A \ eye(size(A))` is the inverse of the nonsingular square matrix `A`.

```
m = 4; A = gf([8 1 6; 3 5 7; 4 9 2],m);
Id = gf(eye(size(A)),m);
X = A \ Id;
ck1 = isequal(X*A, Id)
ck2 = isequal(A*X, Id)
```

The output is below.

```
ck1 =
```

```
1
```

```
ck2 =
```

```
1
```

Other examples are in “Solving Linear Equations”.

Limitations The matrix `A` must be one of these types:

- A nonsingular square matrix
- A tall matrix such that $A'*A$ is nonsingular
- A wide matrix such that $A*A'$ is nonsingular

Algorithm

If A is an M -by- N tall matrix where $M > N$, $A \setminus B$ is the same as $(A' * A) \setminus (A' * B)$.

If A is an M -by- N wide matrix where $M < N$, $A \setminus B$ is the same as $A' * ((A * A') \setminus B)$. This solution is not unique.

See Also

“Linear Algebra in Galois Fields”

mlseeq

Purpose Equalize linearly modulated signal using Viterbi algorithm

Syntax

```
y = mlseeq(x, chcffs, const, tbleen, opmode)
y = mlseeq(x, chcffs, const, tbleen, 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, tbleen, 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. `tbleen` 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 <code>opmode</code>	Typical Usage
'rst'	Enables you to specify a preamble and postamble that accompany your data. The function processes <code>x</code> 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 <code>tbleen</code> symbols.

`y = mlseeq(x, chcffs, const, tbleen, 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 $chcfft$ 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
<code>init_metric</code>	State metrics	1 row, <code>numStates</code> columns	Real numbers
<code>init_states</code>	Traceback states	<code>numStates</code> rows, <code>tblen</code> columns	Integers between 0 and <code>numStates-1</code>
<code>init_inputs</code>	Traceback inputs	<code>numStates</code> rows, <code>tblen</code> 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.

```
M = 2; % Use 2-PAM.
const = pammod([0:M-1],M); % PAM constellation
tblen = 10; % Traceback depth for equalizer
nsamp = 2; % Number of samples per symbol

msgIdx = randint(1000,1,M); % Random bits
msg = upsample(pammod(msgIdx,M),nsamp); % Modulated message
chcoeffs = [.986; .845; .237; .12345+.31i]; % Channel coefficients
```



```
chanest = chcoeffs; % Channel estimate
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
msgRx = awgn(filtmsg,5); % Add Gaussian noise.
msgEq = mlseq(msgRx,chanest,const,tblen,'rst',nsamp); % Equalize.
msgEqIdx = pandemod(msgEq,M); % Demodulate.

[nerrs ber] = biterr(msgIdx, msgEqIdx) % Bit error rate
```

The output is shown below. Your results might vary because this example uses random numbers.

```
nerrs =
```

```
1
```

```
ber =
```

```
0.0010
```

The example in “Example: Continuous Operation Mode” illustrates how to use the final state and initial state arguments when invoking `mlseq` repeatedly.

The example in “Example: Using a Preamble” illustrates how to use a preamble.

See Also

`equalize`, “Using MLSE Equalizers”

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.

modem

Purpose

Package of modem classes

Description

This package contains the modulator and demodulator objects for performing the following modulations:

- DPSK
- MSK
- OQPSK
- PSK
- PAM
- QAM
- General QAM

Properties and Methods

Each modem object has a method `disp` to display its properties.

The methods `modulate` and `demodulate` are available depending on whether the class is a modulator or a demodulator.

See Also

`modem.dpskdemod`, `modem.dpskmod`, `modem.genqamdemod`,
`modem.genqammod`, `modem.msksdemod`, `modem.msksmod`,
`modem.oqpskdemod`, `modem.oqpskmod`, `modem.pamdemod`,
`modem.pammod`, `modem.pskdemod`, `modem.pskmod`, `modem.qamdemod`, and
`modem.qammod`

Purpose

Construct DPSK demodulator object

Syntax

```
h = modem.dpskdemod(property1, value1, ...)
h = modem.dpskdemod(DPSKmod_object)
h = modem.dpskdemod(DPSKmod_object, property1, value1, ...)
h = modem.dpskdemod
```

Description

The `modem.dpskdemod` function creates a demodulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.dpskdemod(property1, value1, ...)` constructs a DPSK demodulator object `h` with properties as specified by the property/value pairs.

`h = modem.dpskdemod(DPSKmod_object)` constructs a DPSK demodulator object `h` by reading the property values from the DPSK modulator object, `DPSKmod_object`. The properties that are unique to the DPSK demodulator object are set to default values.

`h = modem.dpskdemod(DPSKmod_object, property1, value1, ...)` constructs a DPSK demodulator object `h` by reading the property values from the DPSK modulator object, `DPSKmod_object`. Additional properties are specified using property/value pairs.

`h = modem.dpskdemod` constructs a DPSK demodulator object `h` with default properties. It constructs a demodulator object for binary DPSK demodulation, and is equivalent to:

```
h = modem.dpskdemod('M', 2, 'PhaseRotation', 0, 'SymbolOrder', ...
                    'binary', 'OutputType', 'integer', ...
                    'InitialPhase', 0)
```

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the

baseband signal (complex envelope) x with the demodulator object and outputs binary words (bits) or symbols (integers) in signal y .

x can be a multichannel signal. The columns of x are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to 'bit'.

For `h.outputtype = 'bit'`, an output y of size $R \times (nBits \times C)$ is computed for an input x of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output y of size $R \times C$ is computed for an input x of size $R \times C$.

See Using Modem Objects for usage examples.

Properties

A DPSK demodulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'DPSK Demodulator'.
M	Constellation size.
PhaseRotation	Specifies the phase rotation (rad) of the modulation. In this case, the total per-symbol phase shift is the sum of <code>PhaseRotation</code> and the phase generated by the differential modulation.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on <code>M</code> .

Property	Description
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping).
SymbolMapping	A list of integer values from 0 to M-1 that correspond to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Otherwise, it is automatically computed.
OutputType	Type of output to be computed by the DPSK demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).
DecisionType	Type of output values to be computed by DPSK demodulator object. This property is set to 'hard decision' and is not writable.
InitialPhase	Initial phase state of the DPSK demodulator. InitialPhase is used to calculate the first demodulated symbol.

Methods

A DPSK demodulator object has the following four functions for inspection, management, and simulation:

- demodulate
- disp
- copy
- reset

See Using Modem Objects for details and examples of their use.

modem.dpskdemod

Examples

```
% Construct a demodulator object for 4-DPSK demodulation
% with initial phase pi/4.
h = modem.dpskdemod('M', 4, 'InitialPhase', pi/4)

% Construct an object to compute hard bit decisions of a
% baseband signal using 16-DPSK modulation. The modulated
% signal has a minimum phase rotation of pi/8 per symbol.
% The constellation has Gray mapping.
h = modem.dpskdemod('M', 16, 'SymbolOrder', 'Gray', ...
    'PhaseRotation', pi/8, 'OutputType', 'Bit')

% Construct a demodulator object from an existing modulator
% object for DPSK modulation in order to make bit decision.
modObj = modem.dpskmod('M', 8, 'InputType', 'Bit')
demodObj = modem.dpskdemod(modObj)
```

See Also

modem, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.msksdemod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct DPSK modulator object

Syntax

```
h = modem.dpskmod(property1, value1, ...)  
h = modem.dpskmod(DPSKdemod_object)  
h = modem.dpskmod(DPSKdemod_object, property1, value1, ...)  
h = modem.dpskmod
```

Description

The `modem.dpskmod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.dpskmod(property1, value1, ...)` constructs a DPSK modulator object `h` with properties as specified by the property/value pairs.

`h = modem.dpskmod(DPSKdemod_object)` constructs a DPSK modulator object `h` by reading the property values from the DPSK demodulator object, `DPSKdemod_object`. The properties that are unique to the DPSK modulator object are set to default values.

`h = modem.dpskmod(DPSKdemod_object, property1, value1, ...)` constructs a DPSK modulator object `h` by reading the property values from the DPSK demodulator object, `DPSKdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.dpskmod` constructs a DPSK modulator object `h` with default properties. It constructs a modulator object for binary DPSK modulation, and is equivalent to:

```
h = modem.dpskmod('M', 2, 'PhaseRotation', 0, ...  
                  'SymbolOrder', 'binary', ...  
                  'InputType', 'integer', 'InitialPhase', 0)
```

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

x can be a multichannel signal. The columns of x are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., x represents binary input), $nBits$ consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of $nBits$, and elements of x must be 0 or 1. For an input x of size $R \times C$, an output y of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., x represents symbol input), elements of x must be in the range $[0, h.M-1]$. For an input x of size $R \times C$, an output y of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

A DPSK modulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'DPSK Modulator'.
M	Constellation size.
PhaseRotation	Specifies the phase rotation (rad) of the modulation. In this case, the total per-symbol phase shift is the sum of PhaseRotation and the phase generated by the differential modulation.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on M.

Property	Description
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping).
SymbolMapping	A list of integer values from 0 to M-1 that correspond to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Otherwise, it is automatically computed.
InputType	Type of input to be processed by the DPSK modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).
InitialPhase	Initial phase state of the DPSK modulator. InitialPhase is used to calculate the first modulated symbol.

Methods

A DPSK demodulator object has the following four functions for inspection, management, and simulation:

- disp
- copy
- modulate
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for 4-DPSK modulation
% with initial phase pi/4.
h = modem.dpskmod('M', 4, 'InitialPhase', pi/4)
```

modem.dpskmod

```
% Construct an object to modulate binary data using
% 16-DPSK modulation with pi/8 degrees minimum phase
% rotation per symbol. The constellation has Gray mapping.
h = modem.dpskmod('M', 16, 'SymbolOrder', 'Gray', ...
                 'PhaseRotation', pi/8, 'InputType', 'Bit')

% Construct a modulator object from an existing demodulator
% object for DPSK demodulation in order to modulate binary
% inputs.
demodObj = modem.dpskdemod('M', 8) % existing DPSK
                                     % demodulator object
modObj = modem.dpskmod(demodObj)
```

See Also

modem, modem.dpskdemod, modem.genqamdemod, modem.genqammod, modem.msksdemod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct General QAM demodulator object

Syntax

```
h = modem.genqamdemod(property1, value1, ...)
h = modem.genqamdemod(GENQAMmod_object)
h = modem.genqamdemod(GENQAMmod_object, property1, value1,
    ...)
h = modem.genqamdemod
```

Description

The `modem.genqamdemod` function creates a modulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.genqamdemod(property1, value1, ...)` constructs a General QAM demodulator object `h` with properties as specified by the property/value pairs.

`h = modem.genqamdemod(GENQAMmod_object)` constructs a General QAM demodulator object `h` by reading the property values from the General QAM modulator object, `GENQAMmod_object`. The properties that are unique to the General QAM demodulator object are set to default values.

`h = modem.genqamdemod(GENQAMmod_object, property1, value1, ...)` constructs a General QAM demodulator object `h` by reading the property values from the General QAM modulator object, `GENQAMmod_object`. Additional properties are specified using property/value pairs.

`h = modem.genqamdemod` constructs a General QAM demodulator object `h` with default properties. It constructs a demodulator object for 16-QAM modulation and is equivalent to:

```
h = modem.genqamdemod(['Constellation', [-3+j*3, -3+j*1, ...
    -3-j*1, -3-j*3, -1+j*3, -1+j*1, -1-j*1, -1-j*3, ...
    1+j*3, 1+j*1, 1-j*1, 1-j*3, 3+j*3, 3+j*1, 3-j*1, ...
    3-j*3], 'OutputType', 'integer', ...
    'DecisionType', 'hard decision')
```

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

See [Using Modem Objects](#) for usage examples.

Properties

A General QAM demodulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to <code>'General QAM Demodulator'</code> .
M	M-ary value. This property is not writable, and is automatically computed based on <code>Constellation</code> .
Constellation	Signal constellation.

Property	Description
OutputType	Type of output to be computed by the General QAM demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).
DecisionType	Type of output values to be computed by the General QAM demodulator object. The choices are 'hard decision' (hard decision values), 'llr' (log-likelihood ratio), and 'approximate llr' (approximate log-likelihood ratio).
NoiseVariance	Noise variance of the received signal to be processed by the General QAM demodulator object. This is used to compute only the LLR or approximate LLR. Hence, NoiseVariance is visible only when DecisionType is set to 'llr' or 'approximate llr'. If the NoiseVariance value is very small, LLR computations may yield Inf, -Inf, or NaN because the LLR algorithm would involve computing exponentials of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Methods

A General QAM demodulator object has the following functions for inspection, management, and simulation:

- demodulate
- disp
- copy

See “Using Modem Objects” for details and examples of their use.

modem.genqamdemod

Examples

```
% Construct a General QAM demodulator object with an
% equidistant 3-point constellation on the unit circle.
M = 3;
h = modem.genqamdemod('Constellation', exp(j*2*pi*[0:M-1]/M))

% Construct a General QAM demodulator object to compute
% log-likelihood ratio of a baseband signal using a two-tiered
% constellation. The estimated noise variance of input signal
% is 1.2.
h = modem.genqamdemod('Constellation', [exp(j*2*pi*[0:3]/4) ...
                                         2*exp(j*(2*pi*[0:3]/4+pi/4))], ...
                     'OutputType', 'Bit', 'DecisionType', ...
                     'LLR', 'NoiseVariance', 1.2)
plot(h.Constellation, '*');grid on;axis('equal',[-2 2 -2 2]);

% Construct a demodulator object from an existing modulator
% object for general QAM modulation in order to compute
% approximate log-likelihood ratio for a baseband signal
% whose estimated noise variance is 0.81.
modObj = modem.genqammod('Constellation', [-1 1 2*j -2*j], ...
                        'InputType', 'Bit') % existing general QAM modulator object
demodObj = modem.genqamdemod(modObj, 'DecisionType', ...
                             'Approximate LLR', 'NoiseVariance', 0.81)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqammod,
modem.msksdemod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod,
modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod,
modem.qamdemod, and modem.qammod

Purpose

Construct General QAM modulator object

Syntax

```
h = modem.genqammod(property1, value1, ...)  
h = modem.genqammod(GENQAMdemod_object)  
h = modem.genqammod(GENQAMdemod_object, property1, value1,  
    ...)  
h = modem.genqammod
```

Description

The `modem.genqammod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.genqammod(property1, value1, ...)` constructs a General QAM modulator object `h` with properties as specified by the property/value pairs.

`h = modem.genqammod(GENQAMdemod_object)` constructs a General QAM modulator object `h` by reading the property values from the General QAM demodulator object, `GENQAMdemod_object`. The properties that are unique to the General QAM modulator object are set to default values.

`h = modem.genqammod(GENQAMdemod_object, property1, value1, ...)` constructs a General QAM modulator object `h` by reading the property values from the General QAM demodulator object, `GENQAMdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.genqammod` constructs a General QAM modulator object `h` with default properties. It constructs a modulator object for 16-QAM modulation, and is equivalent to:

```
h = modem.genqammod('Constellation', [-3+j*3, -3+j*1, ...  
    -3-j*1, -3-j*3, -1+j*3, -1+j*1, -1-j*1, -1-j*3, ...  
    1+j*3, 1+j*1, 1-j*1, 1-j*3, 3+j*3, 3+j*1, 3-j*1, ...  
    3-j*3], 'InputType', 'integer')
```

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., `x` represents binary input), `nBits` consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of `nBits`, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range $[0, h.M-1]$. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

“Using Modem Objects”

A General QAM modulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'General QAM Modulator'.
M	M-ary value. This property is not writable, and is automatically computed based on Constellation.

Property	Description
Constellation	Signal constellation..
InputType	Type of input to be processed by the General QAM modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).

Methods

A General QAM modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a General QAM modulator object with an
% equidistant 3-point constellation on the unit circle.
M = 3;
h = modem.genqammod('Constellation', exp(j*2*pi*[0:M-1]/M))

% Construct a General QAM object to modulate binary data
% using a two-tiered constellation.
h = modem.genqammod('Constellation', [exp(j*2*pi*[0:3]/4) ...
    2*exp(j*(2*pi*[0:3]/4+pi/4))], 'InputType', 'Bit')
plot(h.Constellation, '*');grid on;axis('equal',[-2 2 -2 2]);

% Construct a modulator object from an existing
% demodulator object for general QAM demodulation in order
% to compute approximate log-likelihood ratio for a baseband
% signal whose estimated noise variance is 0.81.
demodObj = modem.genqamdemod('Constellation', [-1 1 2*j -2*j], ...
```

modem.genqammod

```
'OutputType', 'Bit')  
modObj = modem.genqammod(demodObj)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.msksdemod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct MSK demodulator object

Syntax

```
h = modem.mskdemod(property1, value1, ...)  
h = modem.mskdemod(MSKmod_object)  
h = modem.mskdemod(MSKmod_object, property1, value1, ...)  
h = modem.mskdemod
```

Description

The `modem.mskdemod` function creates a modulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.mskdemod(property1, value1, ...)` constructs an MSK demodulator object `h` with properties as specified by the property/value pairs.

`h = modem.mskdemod(MSKmod_object)` constructs an MSK demodulator object `h` by reading the property values from the MSK modulator object, `MSKmod_object`. The properties that are unique to the MSK demodulator object are set to default values.

`h = modem.mskdemod(MSKmod_object, property1, value1, ...)` constructs an MSK demodulator object `h` by reading the property values from the MSK modulator object, `MSKmod_object`. Additional properties are specified using property/value pairs.

`h = modem.mskdemod` constructs an MSK demodulator object `h` with default properties. This syntax is equivalent to:

```
h = modem.mskdemod('Precoding', 'off', ...  
    'SamplesPerSymbol', 8, 'OutputType', 'bit')
```

Note The MSK demodulator has a 2-bit delay.

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

See [Using Modem Objects](#) for usage examples.

Properties

An MSK demodulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to <code>'MSK Demodulator'</code> .
M	Constellation size. This is a fixed value, set to 2.
Precoding	Specifies the type of the coherent MSK demodulator. The choices are <code>'off'</code> for conventional coherent MSK, and <code>'on'</code> for precoded coherent MSK.
SamplesPerSymbol	Number of samples used to represent an MSK symbol.

Property	Description
OutputType	Type of input to be processed by the MSK demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output). Because the MSK constellation size is two, 'bit' and 'integer' are equivalent.
DecisionType	Type of output values to be computed by MSK demodulator object. This property is set to 'hard decision' and is not writable.

Methods

An MSK demodulator object has the following four functions for inspection, management, and simulation:

- demodulate
- disp
- copy
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct an MSK demodulator object with five samples
% per symbol.
h = modem.msksdemod('SamplesPerSymbol', 5)

% Construct an MSK demodulator object with precoding.
h = modem.msksdemod('Precoding', 'on')

% Construct an MSK demodulator object from an existing
% MSK modulator object.
modObj = modem.msksmod('SamplesPerSymbol', 6, ...
    'Precoding', 'on') % existing MSK modulator object
demodObj = modem.msksdemod(modObj)
```

modem.mskdemod

```
% Modulate and demodulate a bit stream.  
% Note the 2-bit delay.  
demodulate(demodObj, modulate(modObj, ...  
    [1 1 1 0 0 0 1 0 1 0]'))
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose Construct MSK modulator object

Syntax

```
h = modem.mskmod(property1, value1, ...)
h = modem.mskmod(MSKdemod_object)
h = modem.mskmod(MSKdemod_object, property1, value1, ...)
h = modem.mskmod
```

Description The `modem.mskmod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.mskmod(property1, value1, ...)` constructs an MSK modulator object `h` with properties as specified by the property/value pairs.

`h = modem.mskmod(MSKdemod_object)` constructs an MSK modulator object `h` by reading the property values from the MSK demodulator object, `MSKdemod_object`. The properties that are unique to the MSK modulator object are set to default values.

`h = modem.mskmod(MSKdemod_object, property1, value1, ...)` constructs an MSK modulator object `h` by reading the property values from the MSK demodulator object, `MSKdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.mskmod` constructs an MSK modulator object `h` with default properties. This syntax is equivalent to:

```
h = modem.mskmod('Precoding', 'off', ...
    'SamplesPerSymbol', 8, 'InputType', 'bit')
```

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., `x` represents binary input), `nBits` consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of `nBits`, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range $[0, h.M-1]$. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

An MSK modulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'MSK Modulator'.
M	Constellation size. This is a fixed value, set to 2.
Precoding	Specifies the type of the coherent MSK modulator. The choices are 'off' for conventional coherent MSK, and 'on' for precoded coherent MSK.
SamplesPerSymbol	Number of samples used to represent an MSK symbol.
InputType	Type of input to be processed by the MSK modulator object. The choices are 'bit' (bit/binary input), and 'integer' (integer/symbol input). Because the MSK constellation size is two, 'bit' and 'integer' are equivalent.

Methods

An MSK modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for MSK modulation with
% five samples per symbol.
h = modem.mskmod('SamplesPerSymbol', 5)

% Construct an MSK modulator object with precoding and
% 10 samples per symbol.
h = modem.mskmod('Precoding', 'on', 'SamplesPerSymbol', 10)

% Construct a modulator object from an existing demodulator
% object for MSK demodulation in order to modulate binary
% inputs.
demodObj = modem.mskdemod('SamplesPerSymbol', 6) % existing
% MSK demodulator object
modObj = modem.mskmod(demodObj)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.mskdemod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

modem.oqpskdemod

Purpose Construct OQPSK demodulator object

Syntax

```
h = modem.oqpskdemod(property1, value1, ...)  
h = modem.oqpskdemod(OQPSKmod_object)  
h = modem.oqpskdemod(OQPSKmod_object, property1, value1, ...)  
h = modem.oqpskdemod
```

Description The `modem.oqpskdemod` function creates a demodulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.oqpskdemod(property1, value1, ...)` constructs an OQPSK demodulator object `h` with properties as specified by the property/value pairs.

`h = modem.oqpskdemod(OQPSKmod_object)` constructs an OQPSK demodulator object `h` by reading the property values from the OQPSK modulator object, `OQPSKmod_object`. The properties that are unique to the OQPSK demodulator object are set to default values.

`h = modem.oqpskdemod(OQPSKmod_object, property1, value1, ...)` constructs an OQPSK demodulator object `h` by reading the property values from the OQPSK modulator object, `OQPSKmod_object`. Additional properties are specified using property/value pairs.

`h = modem.oqpskdemod` constructs an OQPSK demodulator object `h` with default properties. This syntax is equivalent to:

```
h = modem.oqpskdemod('PhaseOffset', 0, 'SymbolOrder', ...  
    'binary', 'OutputType', 'integer', ...  
    'DecisionType', 'hard decision')
```

Note OQPSK demodulators have a 1 symbol delay.

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

See Using Modem Objects for usage examples.

Properties

An OQPSK demodulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to <code>'OQPSK Demodulator'</code> .
M	M-ary value. This property is set to four and is not writable.
PhaseOffset	Phase offset of ideal signal constellation in radians.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on <code>M</code> and <code>PhaseOffset</code> .

modem.oqpskdemod

Property	Description
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping).
SymbolMapping	A list of integer values from 0 to M-1 that correspond to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Otherwise, it is automatically computed.
OutputType	Type of output to be computed by the OQPSK demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).
DecisionType	Type of output values to be computed by the OQPSK demodulator object. The choices are 'hard decision' (hard decision values), 'llr' (log-likelihood ratio), and 'approximate llr' (approximate log-likelihood ratio).
NoiseVariance	Noise variance of the received signal to be processed by the OQPSK demodulator object. This property is used to compute only the LLR or approximate LLR. Hence, NoiseVariance is visible only when DecisionType is set to 'llr' or 'approximate llr'. If the NoiseVariance value is very small, LLR computations may yield Inf, -Inf, or NaN because the LLR algorithm would involve computing exponentials of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Methods

An OQPSK demodulator object has the following four functions for inspection, management, and simulation:

- demodulate
- disp
- copy
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a demodulator object for OQPSK demodulation
% with default constellation.
h = modem.oqpskdemod

% Construct an object to compute log-likelihood ratio of
% a baseband signal using OQPSK modulation. The
% constellation has Gray mapping and is shifted by -pi/16
% radians. The estimated noise variance of input signal
% is 1.2.
h = modem.oqpskdemod('PhaseOffset', -pi/16, ...
    'SymbolOrder', 'Gray', 'OutputType', 'Bit', ...
    'DecisionType', 'LLR', 'NoiseVariance', 1.2)

% Construct a demodulator object from an existing
% modulator object for OQPSK modulation in order to
% compute approximate log-likelihood ratio for a
% baseband signal whose estimated noise variance is 0.81.
modObj = modem.oqpskmod('InputType', 'Bit')
demodObj = modem.oqpskdemod(modObj, 'DecisionType', ...
    'Approximate LLR', 'NoiseVariance', 0.81)

% Modulate and demodulate a number of symbols.
% Note that there is a one symbol delay.
demodObj = modem.oqpskdemod(modObj);
demodulate(demodObj, modulate(modObj, [0 1 2 3 0 1 2 3]))
```

modem.oqpskdemod

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.msksdemod, modem.msksmod, modem.oqpskmod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct OQPSK modulator object

Syntax

```
h = modem.oqpskmod(property1, value1, ...)  
h = modem.oqpskmod(OQPSKdemod_object)  
h = modem.oqpskmod(OQPSKdemod_object, property1, value1, ...)  
h = modem.oqpskmod
```

Description

The `modem.oqpskmod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.oqpskmod(property1, value1, ...)` constructs an OQPSK modulator object `h` with properties as specified by the property/value pairs.

`h = modem.oqpskmod(OQPSKdemod_object)` constructs an OQPSK modulator object `h` by reading the property values from the OQPSK demodulator object, `OQPSKdemod_object`. The properties that are unique to the OQPSK modulator object are set to default values.

`h = modem.oqpskmod(OQPSKdemod_object, property1, value1, ...)` constructs an OQPSK modulator object `h` by reading the property values from the OQPSK demodulator object, `OQPSKdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.oqpskmod` constructs an OQPSK modulator object `h` with default properties. This syntax is equivalent to:

```
h = modem.oqpskmod('PhaseOffset', 0, 'SymbolOrder', ...  
                  'binary', 'InputType', 'integer')
```

Note OQPSK modulators upsample by 2.

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., `x` represents binary input), `nBits` consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of `nBits`, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range $[0, h.M-1]$. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

An OQPSK modulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'OQPSK Modulator'.
M	M-ary value that is set to four and is not writable.
PhaseOffset	Phase offset of ideal signal constellation in radians.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on M and PhaseOffset.

Property	Description
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping).
SymbolMapping	A list of integer values from 0 to M-1 that correspond to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Otherwise, it is automatically computed.
InputType	Type of input to be processed by the OQPSK modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).

Methods

An OQPSK modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate
- reset

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for OQPSK modulation
% with default constellation .
h = modem.oqpskmod
```

```
% Construct an object to modulate binary data using
% OQPSK modulation. The constellation has Gray mapping
% and is shifted by -pi/16 radians.
```

modem.oqpskmod

```
h = modem.oqpskmod('PhaseOffset', -pi/16, ...
    'SymbolOrder', 'Gray', 'InputType', 'Bit')

% Construct a modulator object from an existing demodulator
% object for OQPSK demodulation in order to modulate binary
% inputs.
demodObj = modem.oqpskdemod('PhaseOffset', pi/3)
modObj = modem.oqpskmod(demodObj, 'InputType', 'Bit')
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.msksdemod, modem.msksmod, modem.oqpskdemod, modem.pamdemod, modem.pammod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct PAM demodulator object

Syntax

```
h = modem.pamdemod(property1, value1, ...)  
h = modem.pamdemod(PAMmod_object)  
h = modem.pamdemod(PAMmod_object, property1, value1, ...)  
h = modem.pamdemod
```

Description

The `modem.pamdemod` function creates a modulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.pamdemod(property1, value1, ...)` constructs a PAM demodulator object `h` with properties as specified by the property/value pairs.

`h = modem.pamdemod(PAMmod_object)` constructs a PAM demodulator object `h` by reading the property values from the PAM modulator object, `PAMmod_object`. The properties that are unique to the PAM demodulator object are set to default values.

`h = modem.pamdemod(PAMmod_object, property1, value1, ...)` constructs a PAM demodulator object `h` by reading the property values from the PAM modulator object, `PAMmod_object`. Additional properties are specified using property/value pairs.

`h = modem.pamdemod` constructs a PAM demodulator object `h` with default properties. It constructs a demodulator object for BPAM demodulation, and is equivalent to:

```
h = modem.pamdemod('M', 2, 'SymbolOrder', 'binary', ...  
                  'OutputType', 'integer', 'DecisionType', 'hard decision')
```

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

x can be a multichannel signal. The columns of x are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output y of size $R \times (nBits \times C)$ is computed for an input x of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output y of size $R \times C$ is computed for an input x of size $R \times C$.

See Using Modem Objects for usage examples.

Properties

An PAM demodulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to <code>'PAM Demodulator'</code> .
M	M-ary value.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on M.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are <code>'binary'</code> (binary mapping), <code>'gray'</code> (Gray mapping), and <code>'user-defined'</code> (custom mapping).
OutputType	Type of output to be computed by the PAM demodulator object. The choices are <code>'bit'</code> (bit/binary output), and <code>'integer'</code> (integer/symbol output).

Property	Description
DecisionType	Type of output values to be computed by the PAM demodulator object. The choices are 'hard decision' (hard decision values), 'llr' (log-likelihood ratio), and 'approximate llr' (approximate log-likelihood ratio).
NoiseVariance	Noise variance of the received signal to be processed by the PAM demodulator object. This is used to compute only the LLR or approximate LLR. Hence, NoiseVariance is visible only when DecisionType is set to 'llr' or 'approximate llr'. If the NoiseVariance value is very small, LLR computations may yield Inf, -Inf, or NaN because the LLR algorithm would involve computing exponentials of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Methods

A PAM demodulator object has the following four functions for inspection, management, and simulation:

- demodulate
- disp
- copy

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a demodulator object for 4-PAM demodulation.
h = modem.pamdemod('M', 4)
```

```
% Construct an object to compute log-likelihood ratio of
```

modem.pamdemod

```
% a baseband signal using 16-PAM modulation. The
% constellation has Gray mapping.
% The estimated noise variance of input signal is 1.2.
h = modem.pamdemod('M', 16, 'SymbolOrder', 'Gray', ...
    'OutputType', 'Bit', 'DecisionType', 'LLR', ...
    'NoiseVariance', 1.2)

% Construct a demodulator object from an existing modulator
% object for PAM modulation in order to compute approximate
% log-likelihood ratio for a baseband signal whose estimated
% noise variance is 0.81.
modObj = modem.pammod('M', 8, 'InputType', 'Bit')
demodObj = modem.pamdemod(modObj, 'DecisionType', ...
    'Approximate LLR', 'NoiseVariance', 0.81)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod,
modem.genqammod, modem.msksdemod, modem.mskmod,
modem.oqpskdemod, modem.oqpskmod, , modem.pskdemod,
modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct PAM modulator object

Syntax

```
h = modem.pammod(property1, value1, ...)
h = modem.pammod(PAMdemod_object)
h = modem.pammod(PAMdemod_object, property1, value1, ...)
h = modem.pammod
```

Description

The `modem.pammod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.pammod(property1, value1, ...)` constructs a PAM modulator object `h` with properties as specified by the property/value pairs.

`h = modem.pammod(PAMdemod_object)` constructs a PAM modulator object `h` by reading the property values from the PAM demodulator object, `PAMdemod_object`. The properties that are unique to the PAM modulator object are set to default values.

`h = modem.pammod(PAMdemod_object, property1, value1, ...)` constructs a PAM modulator object `h` by reading the property values from the PAM demodulator object, `PAMdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.pammod` constructs a PAM modulator object `h` with default properties. It constructs a modulator object for BPAM modulation, and is equivalent to:

```
h = modem.pammod('M', 2, 'SymbolOrder', 'binary', ...
    'InputType', 'integer')
```

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

x can be a multichannel signal. The columns of x are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., x represents binary input), $nBits$ consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of $nBits$, and elements of x must be 0 or 1. For an input x of size $R \times C$, an output y of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., x represents symbol input), elements of x must be in the range $[0, h.M-1]$. For an input x of size $R \times C$, an output y of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

A PAM modulator object has the following properties. All the properties are writable unless explicitly noted otherwise.

Property	Description
Type	Type of modulation object. This is a fixed value, set to 'PAM Modulator'.
M	M-ary value.
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on M and PhaseOffset.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping).

Property	Description
SymbolMapping	A list of integer values from 0 to M-1 that correspond to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Otherwise, it is automatically computed.
InputType	Type of input to be processed by the PAM modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output).

Methods

A PAM modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for 4-PAM modulation.
h = modem.pammod('M', 4)

% Construct an object to modulate binary data using
% 16-PAM modulation.
% The constellation has Gray mapping.
h = modem.pammod('M', 16, 'SymbolOrder', 'Gray', ...
    'InputType', 'Bit')

% Construct a modulator object from an existing
% demodulator object for PAM demodulation in order to
% modulate binary inputs.
demodObj = modem.pamdmod('M', 8)
modObj = modem.pammod(demodObj, 'InputType', 'Bit')
```

modem.pammod

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod, modem.genqammod, modem.msksdemod, modem.mskmod, modem.oqpskdemod, modem.oqpskmod, modem.pamdemod, modem.pskdemod, modem.pskmod, modem.qamdemod, and modem.qammod

Purpose

Construct PSK demodulator object

Syntax

```
h = modem.pskdemod(M)
h = modem.pskdemod(M, phaseoffset)
h = modem.pskdemod(property1, value1, ...)
h = modem.pskdemod
h = modem.pskdemod(pskmod_object)
h = modem.pskdemod(pskmod_object, property1, value1, ...)
```

Description

The `modem.pskdemod` function creates a demodulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.pskdemod(M)` constructs a PSK demodulator object `h` for `M`-ary demodulation.

`h = modem.pskdemod(M, phaseoffset)` constructs a PSK demodulator object `h` whose constellation has a phase offset of `phaseoffset` radians.

`h = modem.pskdemod(property1, value1, ...)` constructs a PSK demodulator object `h` with properties as specified by the property/value pairs. If a property is not specified, it is assigned a default value. See the following section on properties.

`h = modem.pskdemod` constructs a PSK demodulator object `h` with default properties. It constructs a demodulator object for BPSK demodulation and is equivalent to:

```
h = modem.pskdemod('M', 2, 'PhaseOffset', 0, ...
    'SymbolOrder', 'binary', 'OutputType', 'integer', ...
    'DecisionType', 'hard decision')
```

`h = modem.pskdemod(pskmod_object)` constructs a PSK demodulator object `h` by reading the property values from the `pskmod_object` PSK modulator object. The properties that are unique to the PSK demodulator object are set to default values.

`h = modem.pskdemod(pskmod_object, property1, value1, ...)` constructs a PSK demodulator object `h` by reading the property values

from the `pskmod_object` PSK modulator object. Additional properties are specified by the property/value pairs.

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

See [Using Modem Objects](#) for usage examples.

Properties

The following table describes the properties of the PSK demodulator object.

Property	Description
Type	Type of modulation object. This property is a fixed value, set to <code>'PSK Demodulator'</code> .
M	M-ary value. Default is 2.
PhaseOffset	Phase offset of ideal signal constellation in radians. Default is 0.

Property	Description
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on the <code>M</code> and <code>PhaseOffset</code> properties.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping). Default is 'binary'.
SymbolMapping	Symbol mapping values corresponding to ideal constellation points. This property is writable only when <code>SymbolOrder</code> is set to 'user-defined'. Each element of the symbol mapping vector contains the symbol mapped to the corresponding element of the constellation vector. Thus, the first element of the symbol mapping vector contains the symbol mapped to the first element of the constellation vector, the second element contains the symbol mapped to the second element of the constellation vector, and so on.
OutputType	Type of output to be computed by the PSK demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output). Default is 'integer'.

Property	Description
DecisionType	Type of output values to be computed by the PSK demodulator object. The choices are 'hard decision' (hard-decision values), 'llr' (log-likelihood ratio), and 'approximate llr' (approximate log-likelihood ratio). Default is 'hard decision'.
NoiseVariance	Noise variance of the channel or equalized signal to be processed by the PSK demodulator object. The noise variance is used to compute LLR or Approximate LLR, hence NoiseVariance is visible only when DecisionType is set to 'llr' or 'approximate llr'. If the NoiseVariance value is very small, LLR computations may yield Inf, -Inf, or NaN because the LLR algorithm would involve computing exponentials of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Methods

A PSK demodulator object has the following four functions for inspection, management, and simulation:

- copy
- disp
- reset

See “Using Modem Objects” for details and examples of their use.

Algorithms

See “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Examples

```
% Construct a demodulator object for QPSK demodulation.
h = modem.pskdemod(4)

% Construct an object to compute log-likelihood ratio of
% a baseband signal using 16-PSK modulation. The
% constellation has Gray mapping and is shifted by -pi/16
% radians. The estimated noise variance of input
% signal is 1.2.
h = modem.pskdemod('M', 16, 'PhaseOffset', -pi/16, ...
    'SymbolOrder', 'Gray', 'OutputType', 'Bit', ...
    'DecisionType', 'LLR', 'NoiseVariance', 1.2)

% Construct a demodulator object from an existing
% modulator object for PSK modulation in order to
% compute approximate log-likelihood ratio for
% a baseband signal whose estimated noise variance is 0.81.
modObj = modem.pskmod('M', 8, 'InputType', 'Bit')
demodObj = modem.pskdemod(modObj, 'DecisionType', ...
    'Approximate LLR', 'NoiseVariance', 0.81)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod,
modem.genqammod, modem.msksdemod, modem.mskmod,
modem.oqpskdemod, modem.oqpskmod, modem.pamdemod,
modem.pammod, modem.pskmod, modem.qamdemod, and modem.qammod

modem.pskmod

Purpose Construct PSK modulator object

Syntax

```
h = modem.pskmod(M)
h = modem.pskmod(M, phaseoffset)
h = modem.pskmod(property1, value1, ...)
h = modem.pskmod(PSKdemod_object)
h = modem.pskmod(PSKdemod_object, property1, value1, ...)
h = modem.pskmod
```

Description The `modem.pskmod` function (constructor) creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.pskmod(M)` constructs a PSK modulator object `h` for M-ary modulation.

`h = modem.pskmod(M, phaseoffset)` constructs a PSK modulator object `h` whose constellation has a phase offset of `phaseoffset` radians.

`h = modem.pskmod(property1, value1, ...)` constructs a PSK modulator object `h` with properties as specified by the property/value pairs. If a property is not specified, it is assigned a default value. See the following section on properties.

`h = modem.pskmod(PSKdemod_object)` constructs a PSK modulator object `h` by reading the property values from the PSK demodulator object, `PSKdemod_object`. The properties that are unique to the PSK modulator object are set to default values.

`h = modem.pskmod(PSKdemod_object, property1, value1, ...)` constructs a PSK modulator object `h` by reading the property values from the PSK demodulator object, `PSKdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.pskmod` constructs a PSK modulator object `h` with default properties. It constructs a modulator object for BPSK modulation and is equivalent to:

```
h = modem.pskmod('M', 2, 'PhaseOffset', 0, 'SymbolOrder', ...
```


'binary', 'InputType', 'integer')

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., `x` represents binary input), *nBits* consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of *nBits*, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range $[0, h.M-1]$. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

The following table describes the properties of the PSK modulator object.

Property	Description
Type	Type of modulation object. This property is a fixed value, set to 'PSK Modulator'.
M	M-ary value. Default is 2.
PhaseOffset	Phase offset of ideal signal constellation in radians. Default is 0.

Property	Description
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on the M and PhaseOffset properties.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping). Default is 'binary'.
SymbolMapping	Symbol mapping values corresponding to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Each element of the symbol mapping vector contains the symbol mapped to the corresponding element of the constellation vector. Thus, the first element of the symbol mapping vector contains the symbol mapped to the first element of the constellation vector, the second element contains the symbol mapped to the second element of the constellation vector, and so on.
InputType	Type of input to be processed by the PSK modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output). Default is 'integer'.

Methods

A General QAM modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for QPSK modulation.
h = modem.pskmod(4)

% Construct a modulator object for 8-PSK modulation with
% constellation shifted by pi/8 radians.
h = modem.pskmod(8, pi/8)

% Construct an object to modulate binary data using 16-PSK .
% modulation. The constellation has Gray mapping and is
% shifted by -pi/16 radians.
h = modem.pskmod('M', 16, 'PhaseOffset', -pi/16, ...
                'SymbolOrder', 'Gray', 'InputType', 'Bit')
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod,
modem.genqammod, modem.msksdemod, modem.mskmod,
modem.oqpskdemod, modem.oqpskmod, modem.pamdemod,
modem.pammod, modem.pskdemod, modem.qamdemod, and modem.qammod

modem.qamdemod

Purpose Construct QAM demodulator object

Syntax

```
h = modem.qamdemod(M)
h = modem.qamdemod(M, phaseoffset)
h = modem.qamdemod(property1, value1, ...)
h = modem.qamdemod(qammod_object)
h = modem.qamdemod(qammod_object, property1, value1, ...)
h = modem.qamdemod
```

Description The `modem.qamdemod` function creates a demodulator object that you can use with the `demodulate` method to demodulate a signal. To learn more about the process for demodulating a signal, see “Using Modem Objects”.

`h = modem.qamdemod(M)` constructs a QAM demodulator object `h` for `M`-ary demodulation.

`h = modem.qamdemod(M, phaseoffset)` constructs a QAM demodulator object `h` whose constellation has a phase offset of `phaseoffset` radians.

`h = modem.qamdemod(property1, value1, ...)` constructs a QAM demodulator object `h` with properties as specified by the property/value pairs. If a property is not specified, it is assigned a default value. See the following section on properties.

`h = modem.qamdemod(qammod_object)` constructs a QAM demodulator object `h` by reading the property values from the `qammod_object` QAM modulator object. The properties that are unique to the QAM demodulator object are set to default values.

`h = modem.qamdemod(qammod_object, property1, value1, ...)` constructs a QAM demodulator object `h` by reading the property values from the `qammod_object` QAM modulator object. Additional properties are specified by the property/value pairs.

`h = modem.qamdemod` constructs a QAM demodulator object `h` with default properties. It constructs a demodulator object for 16-QAM demodulation and is equivalent to:

```
h = modem.qamdemod('M', 16, 'PhaseOffset', 0, 'SymbolOrder', ...
```

```
'binary', 'OutputType', 'integer', 'DecisionType',...
'hard decision')
```

Modem Demodulation Method

This object has a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the baseband signal (complex envelope) `x` with the demodulator object and outputs binary words (bits) or symbols (integers) in signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = 'bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(h.M)$.

For `h.outputtype = 'integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

See [Using Modem Objects](#) for usage examples.

Properties

The following table describes the properties of the QAM demodulator object.

Property	Description
Type	Type of modulation object. This property is a fixed value, set to <code>'QAM Demodulator'</code> .
M	M-ary value. Default is 2.
PhaseOffset	Phase offset of ideal signal constellation in radians. Default is 0.

modem.qamdemod

Property	Description
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on the M and PhaseOffset properties.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping). Default is 'binary'.
SymbolMapping	Symbol mapping values corresponding to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Each element of the symbol mapping vector contains the symbol mapped to the corresponding element of the constellation vector. Thus, the first element of the symbol mapping vector contains the symbol mapped to the first element of the constellation vector, the second element contains the symbol mapped to the second element of the constellation vector, and so on.
OutputType	Type of output to be computed by the QAM demodulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output). Default is 'integer'.

Property	Description
DecisionType	Type of output values to be computed by the QAM demodulator object. The choices are 'hard decision' (hard-decision values), 'llr' (log-likelihood ratio), and 'approximate llr' (approximate log-likelihood ratio). Default is 'hard decision'.
NoiseVariance	Noise variance of the channel or equalized signal to be processed by the QAM demodulator object. The noise variance is used to compute LLR or Approximate LLR, hence NoiseVariance is visible only when DecisionType is set to 'llr' or 'approximate llr'. If the NoiseVariance value is very small (i.e., SNR is very high), LLR computations may yield Inf or -Inf because the LLR algorithm would involve computing exponentials of very large or very small numbers using finite precision arithmetic. In such cases, use of approximate LLR is recommended, as its algorithm does not involve computing exponentials.

Methods

A QAM demodulator object has the following functions for inspection, management, and simulation:

- copy
- demodulate
- disp

See “Using Modem Objects” for details and examples of their use.

Algorithms

See “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Examples

```
% Construct a demodulator object for 16-QAM demodulation.
h = modem.qamdemod % note that default value of M is 16
```

modem.qamdemod

```
% Construct an object to compute log-likelihood ratio of a
% baseband signal using 64-QAM modulation. The constellation
% has Gray mapping.
% The estimated noise variance of input signal is 12.2.
h = modem.qamdemod('M', 64, 'SymbolOrder', 'Gray', ...
    'OutputType', 'Bit', 'DecisionType', 'LLR', ...
    'NoiseVariance', 12.2)

% Construct a demodulator object from an existing modulator
% object for QAM modulation in order to compute approximate
% log-likelihood ratio for a baseband signal whose estimated
% noise variance is 3.81.
modObj = modem.qammod('M', 8, 'InputType', 'Bit')
demodObj = modem.qamdemod(modObj, 'DecisionType', ...
    'Approximate LLR', 'NoiseVariance', 3.81)
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod,
modem.genqammod, modem.msksdemod, modem.mskmod,
modem.oqpskdemod, modem.oqpskmod, modem.pamdemod,
modem.pammod, modem.pskdemod, modem.pskmod, and modem.qammod

Purpose Construct QAM modulator object

Syntax

```
h = modem.qammod(M)
h = modem.qammod(M, phaseoffset)
h = modem.qammod(property1, value1, ...)
h = modem.qammod(QAMdemod_object)
h = modem.qammod(QAMdemod_object, property1, value1, ...)
h = modem.qammod
```

Description The `modem.qammod` function creates a modulator object that you can use with the `modulate` method to modulate a signal. To learn more about the process for modulating a signal, see “Using Modem Objects”.

`h = modem.qammod(M)` constructs a QAM modulator object `h` for `M`-ary modulation.

`h = modem.qammod(M, phaseoffset)` constructs a QAM modulator object `h` whose constellation has a phase offset of `phaseoffset` radians.

`h = modem.qammod(property1, value1, ...)` constructs a QAM modulator object `h` with properties as specified by the property/value pairs. See the following section on properties.

`h = modem.qammod(QAMdemod_object)` constructs a QAM modulator object `h` by reading the property values from the QAM demodulator object, `QAMdemod_object`. The properties that are unique to the QAM modulator object are set to default values.

`h = modem.qammod(QAMdemod_object, property1, value1, ...)` constructs a QAM modulator object `h` by reading the property values from the QAM demodulator object, `QAMdemod_object`. Additional properties are specified using property/value pairs.

`h = modem.qammod` constructs a QAM modulator object `h` with default properties. It constructs a modulator object for 16-QAM modulation and is equivalent to:

```
h = modem.qammod('M', 16, 'PhaseOffset', 0, 'SymbolOrder',...
    'binary', 'InputType', 'integer')
```

Modem Modulation Method

This object has a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal (complex envelope) `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = 'bit'` (i.e., `x` represents binary input), `nBits` consecutive elements in each channel or column represent a symbol, where $nBits = \log_2(h.M)$. The number of elements in each channel must be an integer multiple of `nBits`, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = 'integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range $[0, h.M-1]$. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

See Using Modem Objects for usage examples.

Properties

The following table describes the properties of the QAM modulator object.

Property	Description
Type	Type of modulation object. This property is a fixed value, set to 'QAM Modulator'.
M	M-ary value. Default is 16.
PhaseOffset	Phase offset of ideal signal constellation in radians. Default is 0.

Property	Description
Constellation	Ideal signal constellation. This property is not writable and is automatically computed based on the M and PhaseOffset properties.
SymbolOrder	Type of mapping employed for mapping symbols to ideal constellation points. The choices are 'binary' (binary mapping), 'gray' (Gray mapping), and 'user-defined' (custom mapping). Default is 'binary'.
SymbolMapping	Symbol mapping values corresponding to ideal constellation points. This property is writable only when SymbolOrder is set to 'user-defined'. Each element of the symbol mapping vector contains the symbol mapped to the corresponding element of the constellation vector. 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. Note that when the alphabet size is 4, this top-down mapping in binary mode effectively creates a gray-mapped constellation.
InputType	Type of input to be processed by the QAM modulator object. The choices are 'bit' (bit/binary output), and 'integer' (integer/symbol output). Default is 'integer'.

Methods

A QAM modulator object has the following functions for inspection, management, and simulation:

- copy
- disp
- modulate

modem.qammod

See “Using Modem Objects” for details and examples of their use.

Examples

```
% Construct a modulator object for 32-QAM modulation.
h = modem.qammod(32)

% Construct an object to modulate binary data using 64-QAM
% modulation. The constellation has Gray mapping.
h = modem.qammod('M', 64, 'SymbolOrder', 'Gray', ...
                'InputType', 'Bit')

% Construct a modulator object from an existing demodulator
% object for QAM demodulation in order to modulate binary
% inputs.
demodObj = modem.qamdemod('M', 8)
modObj = modem.qammod(demodObj, 'InputType', 'Bit')
```

See Also

modem, modem.dpskdemod, modem.dpskmod, modem.genqamdemod,
modem.genqammod, modem.msksdemod, modem.mskmod,
modem.oqpskdemod, modem.oqpskmod, modem.pamdemod,
modem.pammod, modem.pskdemod, modem.pskmod, and modem.qamdemod

Purpose Scaling factor for normalizing modulation output

Syntax

```
scale = modnorm(const, 'avpow', avpow)
scale = modnorm(const, 'peakpow', peakpow)
```

Description `scale = modnorm(const, 'avpow', avpow)` returns a scale factor for normalizing a PAM or QAM modulator output such that its average power is `avpow` (watts). `const` is a vector specifying the reference constellation used to generate the scale factor. The function assumes that the signal to be normalized has a minimum distance of 2.

`scale = modnorm(const, 'peakpow', peakpow)` returns a scale factor for normalizing a PAM or QAM modulator output such that its peak power is `peakpow` (watts).

Examples The code below illustrates how to use `modnorm` to transmit a quadrature amplitude modulated signal having a peak power of one watt.

```
M = 16; % Alphabet size
const = qammod([0:M-1],M); % Generate the constellation.
x = randint(1,100,M);
scale = modnorm(const,'peakpow',1); % Compute scale factor.
y = scale * qammod(x,M); % Modulate and scale.

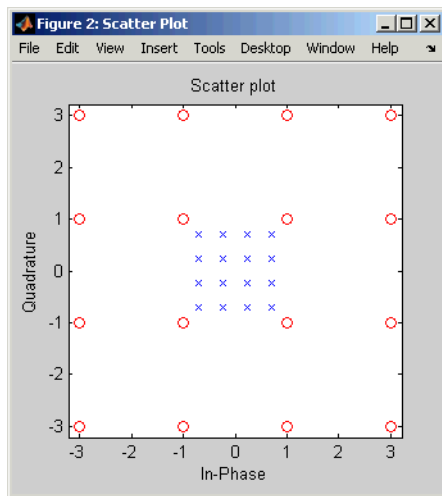
ynoisy = awgn(y,10); % Transmit along noisy channel.

ynoisy_unscaled = ynoisy/scale; % Unscale at receiver end.
z = qamdemod(ynoisy_unscaled,M); % Demodulate.

% See how scaling affects constellation.
h = scatterplot(const,1,0,'ro'); % Unscaled constellation
hold on; % Next plot will be in same figure window.
scatterplot(const*scale,1,0,'bx',h); % Scaled constellation
hold off;
```

In the plot below, the plotting symbol `o` marks points on the original QAM signal constellation, and the plotting symbol `x` marks points on

the signal constellation as scaled by the output of the modnorm function. The channel in this example carries points from the scaled constellation.



Additional examples using modnorm are in “Examples of Signal Constellation Plots”.

See Also

pammod, pamdemod, qammod, qamdemod, “Modulation”

Purpose

Minimum shift keying demodulation

Syntax

```
z = mskdemod(y,nsamp)
z = mskdemod(y,nsamp,dataenc)
z = mskdemod(y,nsamp,dataenc,ini_phase)
z = mskdemod(y,nsamp,dataenc,ini_phase,ini_state)
[z,phaseout] = mskdemod(...)
[z,phaseout,stateout] = mskdemod(...)
```

Description**Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.mskdemod` instead.

`z = mskdemod(y,nsamp)` demodulates the complex envelope `y` of a signal using the differentially encoded minimum shift keying (MSK) method. `nsamp` denotes the number of samples per symbol and must be a positive integer. The initial phase of the demodulator is 0. If `y` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`z = mskdemod(y,nsamp,dataenc)` specifies the method of encoding data for MSK. `dataenc` can be either 'diff' for differentially encoded MSK or 'nondiff' for nondifferentially encoded MSK.

`z = mskdemod(y,nsamp,dataenc,ini_phase)` specifies the initial phase of the demodulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of $\pi/2$. To avoid overriding the default value of `dataenc`, set `dataenc` to `[]`.

`z = mskdemod(y,nsamp,dataenc,ini_phase,ini_state)` specifies the initial state of the demodulator. `ini_state` contains the last half symbol of the previously received signal. `ini_state` is an `nsamp-by-C` matrix, where `C` is the number of channels in `y`.

`[z,phaseout] = mskdemod(...)` returns the final phase of `y`, which is important for demodulating a future signal. The output `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values 0, $\pi/2$, π , and $3\pi/2$.

[z,phaseout,stateout] = mskdemod(...) returns the final nsamp values of y, which is useful for demodulating the first symbol of a future signal. stateout has the same dimensions as the ini_state input.

Examples

The example below illustrates how to modulate and demodulate within a loop. To provide continuity from one iteration to the next, the syntaxes for mskmod and mskdemod use initial phases and/or state as both input and output arguments.

```
% Define parameters.
numbits = 99; % Number of bits per iteration
numchans = 2; % Number of channels (columns) in signal
nsamp = 16; % Number of samples per symbol

% Initialize.
numerrs = 0; % Number of bit errors seen so far
demod_ini_phase = zeros(1,numchans); % Modulator phase
mod_ini_phase = zeros(1,numchans); % Demodulator phase
ini_state = complex(zeros(nsamp,numchans)); % Demod. state

% Main loop
for iRuns = 1 : 10
    x = randint(numbits,numchans); % Binary signal
    [y,phaseout] = mskmod(x,nsamp,[],mod_ini_phase);
    mod_ini_phase = phaseout; % For next mskmod command
    [z, phaseout, stateout] = ...
        mskdemod(y,nsamp,[],demod_ini_phase,ini_state);
    ini_state = stateout; % For next mskdemod command
    demod_ini_phase = phaseout; % For next mskdemod command
    numerrs = numerrs + biterr(x,z); % Cumulative bit errors
end
disp(['Total number of bit errors = ' num2str(numerrs)])
```

The output is as follows.

```
Total number of bit errors = 0
```


References

[1] Pasupathy, Subbarayan, "Minimum Shift Keying: A Spectrally Efficient Modulation," *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

mskmod, fskmod, fskdemod, "Modulation"

mskmod

Purpose Minimum shift keying modulation

Syntax

```
y = mskmod(x,nsamp)
y = mskmod(x,nsamp,dataenc)
y = mskmod(x,nsamp,dataenc,ini_phase)
[y,phaseout] = mskmod(...)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.mskmod` instead.

`y = mskmod(x,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using differentially encoded minimum shift keying (MSK) modulation. The elements of `x` must be 0 or 1. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer. The initial phase of the MSK modulator is 0. If `x` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`y = mskmod(x,nsamp,dataenc)` specifies the method of encoding data for MSK. `dataenc` can be either 'diff' for differentially encoded MSK or 'nondiff' for nondifferentially encoded MSK.

`y = mskmod(x,nsamp,dataenc,ini_phase)` specifies the initial phase of the MSK modulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of $\pi/2$. To avoid overriding the default value of `dataenc`, set `dataenc` to `[]`.

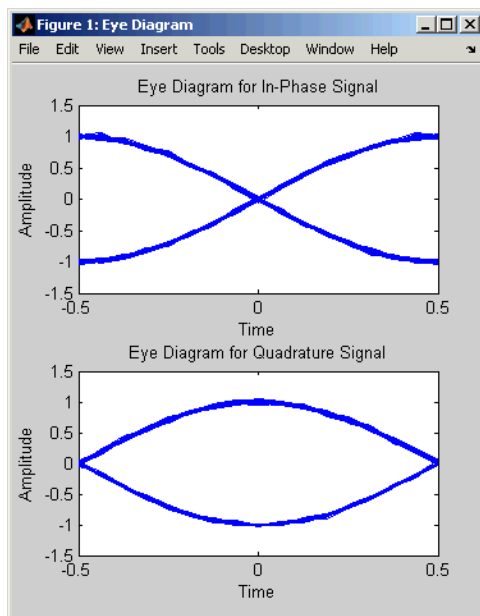
`[y,phaseout] = mskmod(...)` returns the final phase of `y`. This is useful for maintaining phase continuity when you are modulating a future bit stream with differentially encoded MSK. `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values 0, $\pi/2$, π , and $3\pi/2$.

Examples

The code below creates an eye diagram from an MSK signal.

```
x = randint(99,1); % Random signal
y = mskmod(x,8,[],pi/2);
```

```
y = awgn(y,30, 'measured' );  
eyediagram(y,16);
```



The example on the reference page for `mskdemod` also uses this function.

References

[1] Pasupathy, Subbarayan, "Minimum Shift Keying: A Spectrally Efficient Modulation," *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`mskdemod`, `fskmod`, `fskdemod`, "Modulation"

muxdeintrlv

Purpose Restore ordering of symbols using specified shift registers

Syntax

```
deintrlv = muxdeintrlv(data,delay)
[deintrlv,state] = muxdeintrlv(data,delay)
[deintrlv,state] = muxdeintrlv(data,delay,init_state)
```

Description `deintrlv = 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.

`[deintrlv,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.

`[deintrlv,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 `[deintrlv1; deintrlv2]` is the same as `deintrlv`.

```

delay = [0 4 8 12]; % Delays in shift registers
symbols = 100; % Number of symbols to process
% Interleave random data.
intrlved = muxintrlv(randint(symbols,1,2,123),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 “Example: Convolutional Interleavers”.

References

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

See Also

`muxintrlv`, “Interleaving”

muxintrlv

Purpose Permute symbols using shift registers with specified delays

Syntax

```
intrlved = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay,init_state)
```

Description `intrlved = muxintrlv(data,delay)` permutes the elements in `data` by using internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process `data`, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

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

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

Examples The examples in “Example: Convolutional Interleavers” and on the reference page for the `convintrlv` function use `muxintrlv`.

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

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

See Also `muxdeintrlv`, `convintrlv`, `helintrlv`, “Interleaving”

Purpose Equivalent noise bandwidth of filter

Syntax `bw = noisebw(num, den, numsamp, Fs)`

Description `bw = noisebw(num, den, numsamp, Fs)` returns the two-sided equivalent noise bandwidth, in Hz, of a digital lowpass filter given in descending powers of z by numerator vector `num` and denominator vector `den`. The bandwidth is calculated over `numsamp` samples of the impulse response. `Fs` is the sampling rate of the signal that the filter would process; this is used as a scaling factor to convert a normalized unitless quantity into a bandwidth in Hz.

Examples This example computes the equivalent noise bandwidth of a Butterworth filter over 100 samples of the impulse response.

```
Fs = 16; % Sampling rate
Fnyq = Fs/2; % Nyquist frequency
Fc = 0.5; % Carrier frequency
[num,den] = butter(2,Fc/Fnyq); % Butterworth filter
bw = noisebw(num,den,100,Fs)
```

The output is below.

```
bw =

    1.1049
```

Algorithm The two-sided equivalent noise bandwidth is

$$F_s \frac{\sum_{i=1}^N |h(i)|^2}{\left| \sum_{i=1}^N h(i) \right|^2}$$

where h is the impulse response of the filter described by `num` and `den`, and N is `numsamp`.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

Purpose Construct normalized least mean square (LMS) adaptive algorithm object

Syntax `alg = normlms(stepsize)`
`alg = normlms(stepsize,bias)`

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

`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 “Accessing 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”.

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

$$(\text{LeakageFactor})w + \frac{(\text{StepSize})u^* e}{u^H u + \text{Bias}}$$

where the $*$ operator denotes the complex conjugate and H denotes the Hermitian transpose.

See Also

lms, signlms, varlms, rls, cma, lineareq, dfe, equalize, “Equalizers”

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

Purpose Convert octal to decimal numbers

Syntax `d = oct2dec(c)`

Description `d = oct2dec(c)` converts an octal matrix `c` to a decimal matrix `d`, element by element. In both octal and decimal representations, the rightmost digit is the least significant.

Examples The command below converts a 2-by-2 octal matrix.

```
d = oct2dec([12 144;0 25])
```

```
d =
```

```
    10    100  
     0     21
```

For instance, the octal number 144 is equivalent to the decimal number 100 because $144 \text{ (octal)} = 1 \cdot 8^2 + 4 \cdot 8^1 + 4 \cdot 8^0 = 64 + 32 + 4 = 100$.

See Also `bi2de`

oqpskdemod

Purpose Offset quadrature phase shift keying demodulation

Syntax `z = oqpskdemod(y)`
`z = oqpskdemod(y, ini_phase)`

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.oqpskdemod` instead.

`z = oqpskdemod(y)` demodulates the complex envelope `y` of an OQPSK modulated signal. The function implicitly downsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `y` is a matrix with multiple rows, the function processes the columns independently.

`z = oqpskdemod(y, ini_phase)` specifies the phase offset of the modulated signal in radians.

See Also `oqpskmod`, `pskmod`, `pskdemod`, `qammod`, `qamdemod`, `modnorm`, “Modulation”

Purpose Offset quadrature phase shift keying modulation

Syntax
`y = oqpskmod(x)`
`y = oqpskmod(x, ini_phase)`

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.oqpskmod` instead.

`y = oqpskmod(x)` outputs the complex envelope `y` of the modulation of the message signal `x` using offset quadrature phase shift keying (OQPSK) modulation. The message signal must consist of integers between 0 and 3. The function implicitly upsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = oqpskmod(x, ini_phase)` specifies the phase offset of the modulated signal in radians.

See Also `oqpskdemod`, `pskmod`, `pskdemod`, `qammod`, `qamdemod`, `modnorm`, “Modulation”

pamdemod

Purpose Pulse amplitude demodulation

Syntax

```
z = pamdemod(y,M)
z = pamdemod(y,M,ini_phase)
z = pamdemod(y,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.pamdemod` instead.

`z = pamdemod(y,M)` demodulates the complex envelope `y` of a pulse amplitude modulated signal. `M` is the alphabet size. The ideal modulated signal should have a minimum Euclidean distance of 2.

`z = pamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = pamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples The example in “Comparing Theoretical and Empirical Error Rates” uses this function.

See Also `pammod`, `qamdemod`, `qammod`, `pskdemod`, `pskmod`, “Modulation”

Purpose Pulse amplitude modulation

Syntax

```
y = pammod(x,M)
y = pammod(x,M,ini_phase)
y = pammod(x,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.pammod` instead.

`y = pammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using pulse amplitude modulation. `M` is the alphabet size. The message signal must consist of integers between 0 and `M-1`. The modulated signal has a minimum Euclidean distance of 2. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = pammod(x,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = pammod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray constellation ordering.

Examples The example in “Comparing Theoretical and Empirical Error Rates” uses this function.

See Also `pandemod`, `qammod`, `qamdemod`, `pskmod`, `pskdemod`, “Modulation”

plot (channel)

Purpose	Plot channel characteristics with channel visualization tool
Syntax	<code>plot(h)</code>
Description	<code>plot(h)</code> , where <code>h</code> is a channel object, launches the channel visualization tool. This GUI tool allows you to plot channel characteristics in various ways. See “Using the Channel Visualization Tool” for details.
Examples	Examples using this plotting tool are found in “Examples of Using the Channel Visualization Tool”.
See Also	<code>filter</code> , <code>rayleighchan</code> , <code>ricianchan</code>

Purpose

Phase demodulation

Syntax

```
z = pmmod(y,Fc,Fs,phasedev)
z = pmmod(y,Fc,Fs,phasedev,ini_phase)
```

Description

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

Examples

The example in “Analog Modulation Example” uses `pmdemod`.

See Also

`pmmod`, `fmmod`, `fmdemod`, “Modulation”

pmmmod

Purpose Phase modulation

Syntax `y = pmmmod(x,Fc,Fs,phasedev)`
`y = pmmmod(x,Fc,Fs,phasedev,ini_phase)`

Description `y = pmmmod(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 $2 \cdot Fc$. The `phasedev` argument is the phase deviation of the modulated signal in radians.

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

Examples The example in “Analog Modulation Example” uses `pmmmod`.

See Also `pmdemod`, `fmmmod`, `fndemod`, “Modulation”

Purpose Convert convolutional code polynomials to trellis description

Syntax

```
trellis = poly2trellis(ConstraintLength,CodeGenerator)
trellis = poly2trellis(ConstraintLength,CodeGenerator,...
    FeedbackConnection)
```

Description The poly2trellis function accepts a polynomial description of a convolutional encoder and returns the corresponding trellis structure description. The output of poly2trellis is suitable as an input to the convenc and vitdec functions, and as a mask parameter for the Convolutional Encoder, Viterbi Decoder, and APP Decoder blocks in Communications Blockset™ software.

trellis = poly2trellis(ConstraintLength,CodeGenerator) performs the conversion for a rate k/n feedforward encoder. ConstraintLength is a 1-by-k vector that specifies the delay for the encoder's k input bit streams. CodeGenerator is a k-by-n matrix of octal numbers that specifies the n output connections for each of the encoder's k input bit streams.

trellis = poly2trellis(ConstraintLength,CodeGenerator,... FeedbackConnection) is the same as the syntax above, except that it applies to a feedback, not feedforward, encoder. FeedbackConnection is a 1-by-k vector of octal numbers that specifies the feedback connections for the encoder's k input bit streams.

For both syntaxes, the output is a MATLAB structure whose fields are as in the table below.

Fields of the Output Structure trellis for a Rate k/n Code

Field in trellis Structure	Dimensions	Meaning
numInputSymbols	Scalar	Number of input symbols to the encoder: 2^k

Fields of the Output Structure `trellis` for a Rate k/n Code (Continued)

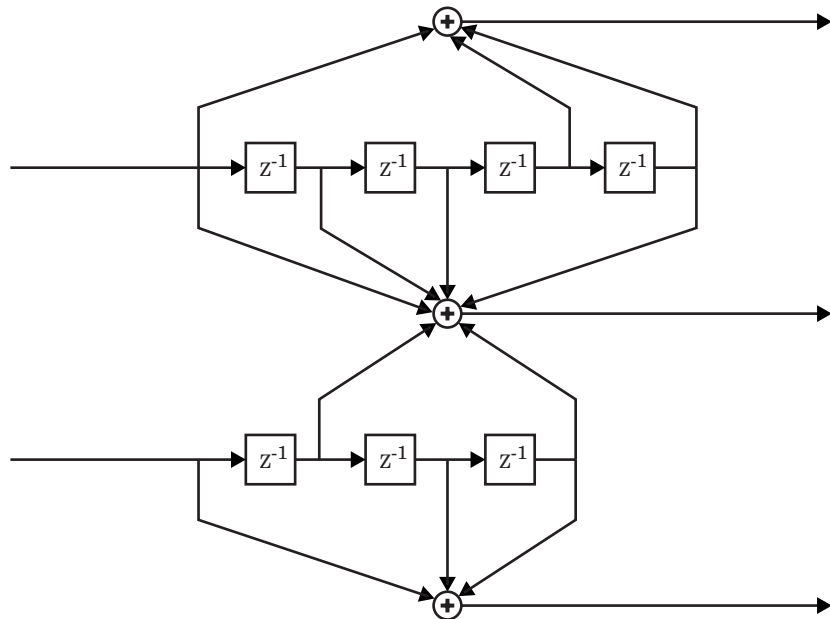
Field in <code>trellis</code> Structure	Dimensions	Meaning
<code>numOutputSymbols</code>	Scalar	Number of output symbols from the encoder: 2^n
<code>numStates</code>	Scalar	Number of states in the encoder
<code>nextStates</code>	<code>numStates-by-2^k</code> matrix	Next states for all combinations of current state and current input
<code>outputs</code>	<code>numStates-by-2^k</code> matrix	Outputs (in octal) for all combinations of current state and current input

For more about this structure, see the reference page for the `istrellis` function.

Examples

An example of a rate $1/2$ encoder is in “Polynomial Description of a Convolutional Encoder”.

As another example, consider the rate $2/3$ feedforward convolutional encoder depicted in the figure below. The reference page for the `convenc` function includes an example that uses this encoder.



For this encoder, the ConstraintLength vector is $[5,4]$ and the CodeGenerator matrix is $[23,35,0; 0,5,13]$. The output below reveals part of the corresponding trellis structure description of this encoder.

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])
```

```
trellis =
```

```
    numInputSymbols: 4
    numOutputSymbols: 8
        numStates: 128
    nextStates: [128x4 double]
        outputs: [128x4 double]
```

The scalar field `trellis.numInputSymbols` has the value 4 because the combination of two input bit streams can produce four different input

symbols. Similarly, `trellis.numOutputSymbols` is 8 because the three output bit streams can produce eight different output symbols.

The scalar field `trellis.numStates` is 128 (that is, 2^7) because each of the encoder's seven memory registers can have one of two binary values.

To get details about the matrix fields `trellis.nextStates` and `trellis.outputs`, inquire specifically about them. As an example, the command below displays the first five rows of the 128-by-4 matrix `trellis.nextStates`.

```
trellis.nextStates(1:5,:)
```

```
ans =
```

```
    0    64     8    72
    0    64     8    72
    1    65     9    73
    1    65     9    73
    2    66    10    74
```

This first row indicates that if the encoder starts in the zeroth state and receives input bits of 00, 01, 10, or 11, respectively, the next state will be the 0th, 64th, 8th, or 72nd state, respectively. The 64th state means that the bottom-left memory register in the diagram contains the value 1, while the other six memory registers contain zeros.

See Also

`istrellis`, `convenc`, `vitdec`, “Convolutional Coding”

Purpose Find primitive polynomials for Galois field

Syntax

```
pr = primpoly(m)
pr = primpoly(m,opt)
pr = primpoly(m..., 'nodisplay')
```

Description `pr = primpoly(m)` returns the primitive polynomial for $GF(2^m)$, where m is an integer between 2 and 16. The Command Window displays the polynomial using "D" as an indeterminate quantity. The output argument `pr` is an integer whose binary representation indicates the coefficients of the polynomial.

`pr = primpoly(m,opt)` returns one or more primitive polynomials for $GF(2^m)$. The output `pol` depends on the argument `opt` as shown in the table below. Each element of the output argument `pr` is an integer whose binary representation indicates the coefficients of the corresponding polynomial. If no primitive polynomial satisfies the constraints, `pr` is empty.

opt	Meaning of pr
'min'	One primitive polynomial for $GF(2^m)$ having the smallest possible number of nonzero terms
'max'	One primitive polynomial for $GF(2^m)$ having the greatest possible number of nonzero terms
'all'	All primitive polynomials for $GF(2^m)$
Positive integer k	All primitive polynomials for $GF(2^m)$ that have k nonzero terms

`pr = primpoly(m..., 'nodisplay')` prevents the function from displaying the result as polynomials in "D" in the Command Window. The output argument `pr` is unaffected by the 'nodisplay' option.

Examples

The first example below illustrates the formats that `primpoly` uses in the Command Window and in the output argument `pr`. The subsequent examples illustrate the display options and the use of the *opt* argument.

```
pr = primpoly(4)

pr1 = primpoly(5, 'max', 'nodisplay')

pr2 = primpoly(5, 'min')

pr3 = primpoly(5,2)

pr4 = primpoly(5,3);
```

The output is below.

```
Primitive polynomial(s) =
D^4+D^1+1

pr =
    19

pr1 =
    61

Primitive polynomial(s) =
```


D^5+D^2+1

pr2 =

37

No primitive polynomial satisfies the given constraints.

pr3 =

[]

Primitive polynomial(s) =

D^5+D^2+1

D^5+D^3+1

See Also

isprimitive, "Galois Field Computations"

pskdemod

Purpose Phase shift keying demodulation

Syntax

```
z = pskdemod(y,M)
z = pskdemod(y,M,ini_phase)
z = pskdemod(y,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.pskdemod` instead.

`z = pskdemod(y,M)` demodulates the complex envelope `y` of a PSK modulated signal. `M` is the alphabet size and must be an integer power of 2. The initial phase of the modulation is zero. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

`z = pskdemod(y,M,ini_phase)` specifies the initial phase of the modulation in radians.

`z = pskdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples

The example below compares PSK and PAM (phase amplitude modulation) to show that PSK is more sensitive to phase noise. This is the expected result because the PSK constellation is circular, and the PAM constellation is linear.

```
len = 10000; % Number of symbols
M = 16; % Size of alphabet
msg = randint(len,1,M); % Original signal

% Modulate using both PSK and PAM,
% to compare the two methods.
txpsk = pskmod(msg,M);
```

```
txpam = pammod(msg,M);

% Perturb the phase of the modulated signals.
phasenoise = randn(len,1)*.015;
rxpsk = txpsk.*exp(j*2*pi*phasenoise);
rxpam = txpam.*exp(j*2*pi*phasenoise);

% Create a scatter plot of the received signals.
scatterplot(rxpsk); title('Noisy PSK Scatter Plot')
scatterplot(rxpam); title('Noisy PAM Scatter Plot')

% Demodulate the received signals.
recovpsk = pskdemod(rxpsk,M);
recovpam = pamdemod(rxpam,M);

% Compute number of symbol errors in each case.
numerrs_psk = symerr(msg,recovpsk)
numerrs_pam = symerr(msg,recovpam)
```

The output and scatter plots are below. Your results might vary because this example uses random numbers.

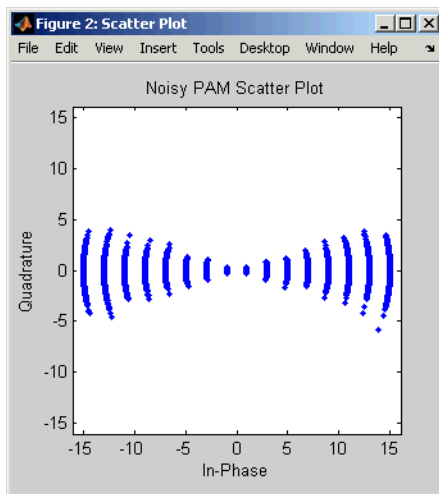
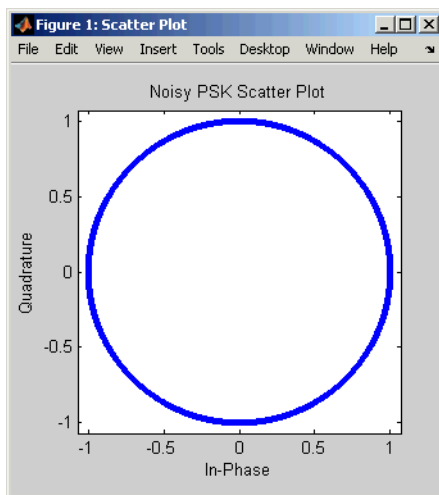
```
numerrs_psk =
```

```
374
```

```
numerrs_pam =
```

```
1
```

pskdemod



See Also

pskmod, qamdemod, qammod, dpskmod, dpskdemod, modnorm, "Modulation"

Purpose Phase shift keying modulation

Syntax

```
y = pskmod(x,M)
y = pskmod(x,M,ini_phase)
y = pskmod(x,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.pskmod` instead.

`y = pskmod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using phase shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The initial phase of the modulation is zero. If `x` is a matrix with multiple rows and columns, the function processes the columns independently.

`y = pskmod(x,M,ini_phase)` specifies the initial phase of the modulation in radians.

`y = pskmod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray constellation ordering.

Examples The examples in “Constellation for 16-PSK” and on the reference page for `pskdemod` use this function.

See Also `dpskmod`, `dpskdemod`, `pskdemod`, `pammod`, `pamdmod`, `qammod`, `qamdmod`, `modnorm`, “Modulation”

qamdemod

Purpose Quadrature amplitude demodulation

Syntax

```
z = qamdemod(y,M)
z = qamdemod(y,M,ini_phase)
z = qamdemod(y,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.qamdemod` instead.

`z = qamdemod(y,M)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. `M` is the alphabet size and must be an integer power of 2. The constellation is the same as in `qammod`. If `y` is a matrix with multiple rows, the function processes the columns independently.

`z = qamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

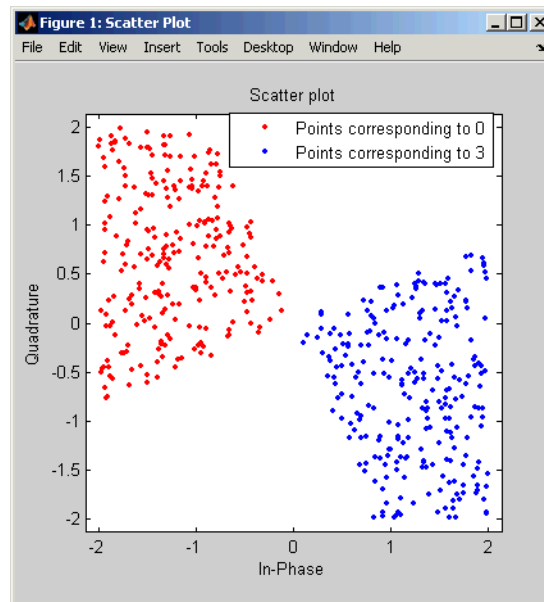
`z = qamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray-coded ordering.

Examples

The code below suggests which regions in the complex plane are associated with different digits that can form the output of the demodulator. The code demodulates random points, looks for points that were demapped to the digits 0 and 3, and plots those points in red and blue, respectively. Notice that the regions reflect a rotation of the signal constellation by $\pi/8$.

```
% Construct [in-phase, quadrature] for random points.
y = 4*(rand(1000,1)-1/2)+j*4*(rand(1000,1)-1/2);
% Demodulate using an initial phase of pi/8.
z = qamdemod(y,4,pi/8);
% Find indices of points that mapped to the digits 0 and 3.
```

```
red = find(z==0);  
blue = find(z==3);  
% Plot points corresponding to 0 and 3.  
h = scatterplot(y(red,:),1,0,'r. '); hold on  
scatterplot(y(blue,:),1,0,'b. ',h);  
legend('Points corresponding to 0','Points corresponding to 3');  
hold off
```



Another example using this function is in “Computing the Symbol Error Rate”.

See Also

qammod, genqamdemod, genqammod, pamdemod, modnorm, “Modulation”

qammod

Purpose Quadrature amplitude modulation

Syntax

```
y = qammod(x,M)
y = qammod(x,M,ini_phase)
y = qammod(x,M,ini_phase,symbol_order)
```

Description **Warning**

This function is obsolete and may be removed in the future. We strongly recommend that you use `modem.qammod` instead.

`y = qammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The signal constellation is rectangular or cross-shaped, and the nearest pair of points in the constellation is separated by 2. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = qammod(x,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = qammod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to 'bin' (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to 'gray', it uses a Gray constellation ordering.

Examples Examples using this function are in “Computing the Symbol Error Rate” and “Examples of Signal Constellation Plots”.

See Also `qamdmod`, `genqammod`, `genqamdmod`, `pammod`, `pamdmod`, `modnorm`, “Modulation”

Purpose Q function

Syntax `y = qfunc(x)`

Description `y = qfunc(x)` is one minus the cumulative distribution function of the standardized normal random variable, evaluated at each element of the real array `x`. For a scalar `x`, the formula is

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp(-t^2/2) dt$$

The Q function is related to the complementary error function, `erfc`, according to

$$Q(x) = \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Examples

The example below computes the Q function on a matrix, element by element.

```
x = [0 1 2; 3 4 5];
format short e % Switch to floating point format for displays.
y = qfunc(x)
format % Return to default format for displays.
```

The output is below.

```
y =
    5.0000e-001    1.5866e-001    2.2750e-002
    1.3499e-003    3.1671e-005    2.8665e-007
```

See Also `qfuncinv`, `erf`, `erfc`, `erfcx`, `erfinv`, `erfcinv`

qfuncinv

Purpose Inverse Q function

Syntax `y = qfuncinv(x)`

Description `y = qfuncinv(x)` returns the argument of the Q function at which the Q function's value is `x`. The input `x` must be a real array with elements between 0 and 1, inclusive.

For a scalar `x`, the Q function is one minus the cumulative distribution function of the standardized normal random variable, evaluated at `x`. The Q function is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp(-t^2/2) dt$$

The Q function is related to the complementary error function, `erfc`, according to

$$Q(x) = \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Examples

The example below illustrates the inverse relationship between `qfunc` and `qfuncinv`.

```
x1 = [0 1 2; 3 4 5];  
y1 = qfuncinv(qfunc(x1)) % Invert qfunc to recover x1.  
x2 = 0:.2:1;  
y2 = qfunc(qfuncinv(x2)) % Invert qfuncinv to recover x2.
```

The output is below.

y1 =

0	1	2
3	4	5

y2 =

0	0.2000	0.4000	0.6000	0.8000	1.0000
---	--------	--------	--------	--------	--------

See Also

qfunc, erf, erfc, erfcx, erfinv, erfcinv

quantiz

Purpose Produce quantization index and quantized output value

Syntax

```
index = quantiz(sig,partition)
[index,quants] = quantiz(sig,partition,codebook)
[index,quants,distor] = quantiz(sig,partition,codebook)
```

Description `index = quantiz(sig,partition)` returns the quantization levels in the real vector signal `sig` using the parameter `partition`. `partition` is a real vector whose entries are in strictly ascending order. If `partition` has length `n`, `index` is a vector whose `k`th entry is

- 0 if $\text{sig}(k) \leq \text{partition}(1)$
- `m` if $\text{partition}(m) < \text{sig}(k) \leq \text{partition}(m+1)$
- `n` if $\text{partition}(n) < \text{sig}(k)$

`[index,quants] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `codebook` prescribes a value for each partition in the quantization and `quants` contains the quantization of `sig` based on the quantization levels and prescribed values. `codebook` is a vector whose length exceeds the length of `partition` by one. `quants` is a row vector whose length is the same as the length of `sig`. `quants` is related to `codebook` and `index` by

```
quants(ii) = codebook(index(ii)+1);
```

where `ii` is an integer between 1 and `length(sig)`.

`[index,quants,distor] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `distor` estimates the mean square distortion of this quantization data set.

Examples The command below rounds several numbers between 1 and 100 up to the nearest multiple of 10. `quants` contains the rounded numbers, and `index` tells which quantization level each number is in.

```
[index,quants] = quantiz([3 34 84 40 23],10:10:90,10:10:100)
```

The output is below.

```
index =
```

```
    0    3    8    3    2
```

```
quants =
```

```
    10    40    90    40    30
```

See Also

`lloyds`, `dpcmenco`, `dpcmdeco`, “Quantizing a Signal”

randdeintrlv

Purpose Restore ordering of symbols using random permutation

Syntax `deintrlvd = randdeintrlv(data,state)`

Description `deintrlvd = randdeintrlv(data,state)` restores the original ordering of the elements in `data` by inverting a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

To use this function as an inverse of the `randintrlv` function, use the same `state` input in both functions. In that case, the two functions are inverses in the sense that applying `randintrlv` followed by `randdeintrlv` leaves data unchanged.

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

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

See `rand` for details on the generator algorithm.

Examples For an example using random interleaving and deinterleaving, see “Example: Block Interleavers”.

See Also `rand`, `randintrlv`, “Interleaving”

Purpose Generate bit error patterns

Syntax

```
out = randerr(m)
out = randerr(m,n)
out = randerr(m,n,errors)
out = randerr(m,n,prob,state)
```

Description For all syntaxes, randerr treats each row of out independently.

out = randerr(m) generates an m-by-m binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

out = randerr(m,n) generates an m-by-n binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

out = randerr(m,n,errors) generates an m-by-n binary matrix, where errors determines how many nonzero entries are in each row:

- If errors is a scalar, it is the number of nonzero entries in each row.
- If errors is a row vector, it lists the possible number of nonzero entries in each row.
- If errors is a matrix having two rows, the first row lists the possible number of nonzero entries in each row and the second row lists the probabilities that correspond to the possible error counts.

Once randerr determines the *number* of nonzero entries in a given row, each configuration of that number of nonzero entries has equal probability.

out = randerr(m,n,prob,state) is the same as the syntax above, except that it first resets the state of the uniform random number generator rand to the integer state.

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

randerr

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

The examples below generate an 8-by-7 binary matrix, each row of which is equally likely to have either zero or two nonzero entries, and then alter the scenario by making it three times as likely that a row has two nonzero entries. Notice in the latter example that the second row of the error parameter sums to one.

```
out = randerr(8,7,[0 2])
```

```
out2 = randerr(8,7,[0 2; .25 .75])
```

Sample output is below.

```
out =
```

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

```
out2 =
```

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



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

See Also [rand](#), [randsrc](#), [randint](#), “Signal Sources”

randint

Purpose Generate matrix of uniformly distributed random integers

Syntax

```
out = randint
out = randint(m)
out = randint(m,n)
out = randint(m,n,rg)
out = randint(m,n,rg,state)
```

Description

`out = randint` generates a random scalar that is either 0 or 1, with equal probability.

`out = randint(m)` generates an m-by-m binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n)` generates an m-by-n binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n,rg)` generates an m-by-n integer matrix. If `rg` is zero, `out` is a zero matrix. Otherwise, the entries are uniformly distributed and independently chosen from the range

- `[0, rg-1]` if `rg` is a positive integer
- `[rg+1, 0]` if `rg` is a negative integer
- Between `min` and `max`, inclusive, if `rg = [min,max]` or `[max,min]`

`out = randint(m,n,rg,state)` is the same as the syntax above, except that it first resets the state of the uniform random number generator `rand` to the integer state.

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

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

See `rand` for details on the generator algorithm.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed in the range from 0 to 7, use either of the following commands.

```
out = randint(10,10,[0,7]);
```

```
out = randint(10,10,8);
```

See Also

rand, randsrc, randerr, “Signal Sources”

randintrlv

Purpose Reorder symbols using random permutation

Syntax `intrlvd = randintrlv(data, state)`

Description `intrlvd = randintrlv(data, state)` rearranges the elements in `data` using a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable and invertible for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

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

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

See `rand` for details on the generator algorithm.

Examples For an example using random interleaving and deinterleaving, see “Example: Block Interleavers”.

See Also `rand`, `randdeintrlv`, “Interleaving”

Purpose

Generate random matrix using prescribed alphabet

Syntax

```
out = randsrc
out = randsrc(m)
out = randsrc(m,n)
out = randsrc(m,n,alphabet)
out = randsrc(m,n,[alphabet; prob])
out = randsrc(m,n,...,state);
```

Description

`out = randsrc` generates a random scalar that is either -1 or 1, with equal probability.

`out = randsrc(m)` generates an m -by- m matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n)` generates an m -by- n matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n,alphabet)` generates an m -by- n matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Each entry in `alphabet` occurs in `out` with equal probability. Duplicate values in `alphabet` are ignored.

`out = randsrc(m,n,[alphabet; prob])` generates an m -by- n matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Duplicate values in `alphabet` are ignored. The row vector `prob` lists corresponding probabilities, so that the symbol `alphabet(k)` occurs with probability `prob(k)`, where k is any integer between one and the number of columns of `alphabet`. The elements of `prob` must add up to 1.

`out = randsrc(m,n,...,state);` is the same as the two preceding syntaxes, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

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

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

See rand for details on the generator algorithm.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed among members of the set {-3,-1,1,3}, you can use either of these commands.

```
out = randsrc(10,10,[-3 -1 1 3]);
```

```
out = randsrc(10,10,[-3 -1 1 3; .25 .25 .25 .25]);
```

To skew the probability distribution so that -1 and 1 each occur with probability .3, while -3 and 3 each occur with probability .2, use this command.

```
out = randsrc(10,10,[-3 -1 1 3; .2 .3 .3 .2]);
```

See Also

rand, randint, randerr, “Signal Sources”

Purpose Construct Rayleigh fading channel object

Syntax

```
chan = rayleighchan(ts,fd)
chan = rayleighchan(ts,fd,tau,pdb)
chan = rayleighchan
```

Description `chan = rayleighchan(ts,fd)` constructs a frequency-flat (“single path”) Rayleigh fading channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. You can model the effect of the channel on a signal `x` by using the syntax `y = filter(chan,x)`.

`chan = rayleighchan(ts,fd,tau,pdb)` constructs a frequency-selective (“multiple path”) fading channel object that models each discrete path as an independent Rayleigh fading process. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

With the above two syntaxes, a smaller `fd` (a few hertz to a fraction of a hertz) leads to slower variations, and a larger `fd` (a couple hundred hertz) to faster variations.

`chan = rayleighchan` constructs a frequency-flat Rayleigh channel object with no Doppler shift. This is a static channel. The sample time of the input signal is irrelevant for frequency-flat static channels.

Properties

The tables below describe the properties of the channel object, `chan`, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see “Viewing Object Properties” or “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.

Writeable Properties (Continued)

Property	Description
StorePathGains	If set to 1, the complex path gain vector is stored as the channel filter function processes the signal. The default value is 0.
ResetBeforeFiltering	If 1, each call to filter resets the state of chan before filtering. If 0, the fading process maintains continuity from one call to the next.

Read-Only Properties

Property	Description	When MATLAB Sets or Updates Value
ChannelType	Fixed value, 'Rayleigh'	When you create object
PathGains	Complex vector listing the current gains of the discrete paths. When you create or reset chan, PathGains is a random vector influenced by AvgPathGaindB and NormalizePathGains.	When you create object, reset object, or use it to filter a signal

Read-Only Properties (Continued)

Property	Description	When MATLAB Sets or Updates Value
ChannelFilterDelay	Delay of the channel filter, measured in samples	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. See “Using the Channel Visualization Tool” for details.

Examples

Several examples using this function are in “Fading Channels”.

The example below illustrates that when you change the value of `PathDelays`, MATLAB automatically changes the values of other properties to make their vector lengths consistent with that of the new value of `PathDelays`.

```
c1 = rayleighchan(1e-5,130) % Create object.
c1.PathDelays = [0 1e-6] % Change the number of delays.
% MATLAB automatically changes the size of c1.AvgPathGaindB,
% c1.PathGains, and c1.ChannelFilterDelay.
```

The output below displays all the properties of the channel object before and after the change in the value of the `PathDelays` property. In the second listing of properties, the `AvgPathGaindB`, `PathGains`, and `ChannelFilterDelay` properties all have different values compared to the first listing of properties.

```
c1 =

    ChannelType: 'Rayleigh'
InputSamplePeriod: 1.0000e-005
DopplerSpectrum: [1x1 doppler.jakes]
MaxDopplerShift: 130
    PathDelays: 0
    AvgPathGaindB: 0
```

```
NormalizePathGains: 1
  StoreHistory: 0
    PathGains: 0.2104- 0.6197i
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: [-0.3088+ 0.1842i 0.3008- 0.0338i]
ChannelFilterDelay: 4
ResetBeforeFiltering: 1
NumSamplesProcessed: 0
```

Algorithm

The methodology used to simulate fading channels is described in “Simulation of Multipath Fading Channels: Methodology”. 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 $1 \leq k \leq K$.
- The `PathGains` read-only property contains the values of $\{a_k\}$, where $1 \leq k \leq K$.

- The AvgPathGaindB vector property contains the values of

$10\log_{10}\left\{E\left[|a_k|^2\right]\right\}$, where $1 \leq k \leq K$, and $E[\cdot]$ denotes statistical expectation.

- The ChannelFilterDelay read-only property contains the value of N_1 .

See Also

ricianchan, filter, plot (channel), reset, “Fading Channels”

References

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

rcosfir

Purpose Design raised cosine finite impulse response (FIR) filter

Syntax

```
b = rcosfir(R,n_T,rate,T)
b = rcosfir(R,n_T,rate,T,filter_type)
rcosfir(...)
rcosfir(...,colr)
[b,sample_time] = rcosfir(...)
```

Optional Inputs

Input	Default Value
n_T	3
rate	5
T	1

Description The `rcosfir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes 'fir'. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t / T)}{(\pi t / T)} \cdot \frac{\cos(\pi R t / T)}{(1 - 4R^2 t^2 / T^2)}$$

`b = rcosfir(R,n_T,rate,T)` designs a raised cosine filter and returns a vector `b` of length $(n_T(2) - n_T(1)) * \text{rate} + 1$. The filter's rolloff factor is `R`, a real number between 0 and 1, inclusive. `T` is the duration of each bit in seconds. `n_T` is a scalar or a vector of length 2. If `n_T` is specified as a scalar, the filter length is $2 * n_T + 1$ input samples. If `n_T` is a vector, it specifies the extent of the filter. In this case, the filter length is $n_T(2) - n_T(1) + 1$ input samples (or $(n_T(2) - n_T(1)) * \text{rate} + 1$ output samples).

`rate` is the number of points in each input symbol period of length `T`. `rate` must be greater than 1. The input sample rate is `T` samples per second, while the output sample rate is `T*rate` samples per second.

The order of the FIR filter is

$$(n_T(2) - n_T(1)) * \text{rate}$$

The arguments `n_T`, `rate`, and `T` are optional inputs whose default values are 3, 5, and 1, respectively.

`b = rcosfir(R,n_T,rate,T,filter_type)` designs a square-root raised cosine filter if `filter_type` is 'sqrt'. If `filter_type` is 'normal', this syntax is the same as the previous one.

The impulse response of a square root raised cosine filter is

$$h(t) = 4R \frac{\cos((1+R)\pi t/T) + \frac{\sin((1-R)\pi t/T)}{4R \frac{t}{T}}}{\pi \sqrt{T} (1 - (4Rt/T)^2)}$$

`rcosfir(...)` produces plots of the time and frequency responses of the raised cosine filter.

`rcosfir(...,color)` uses the string `color` to determine the plotting color. The choices for `color` are the same as those listed for the `plot` function.

`[b,sample_time] = rcosfir(...)` returns the FIR filter and its sample time.

Examples

The commands below compare different rolloff factors.

```
rcosfir(0);
subplot(211); hold on;
subplot(212); hold on;
rcosfir(.5,[],[],[],[],[],'r-');
rcosfir(1,[],[],[],[],[],'g-');
```

rcosfir

See Also

rcosiir, rcosflt, rcosine, firrcos, rcosdemo, “Special Filters”

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose Filter input signal using raised cosine filter

Syntax

```

y = rcosflt(x,Fd,Fs)
y = rcosflt(x,Fd,Fs,'type_flag',r,delay,tol)
y = rcosflt(x,Fd,Fs,'filter_type/Fs',r,delay,tol)
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den)
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den,delay)
y = rcosflt(x,Fd,Fs,'filter_type/filter/Fs',num,den...)
[y,t] = rcosflt(...)

```

Optional Inputs

Input	Default Value
<i>filter_type</i>	fir/normal
r	0.5
delay	3
tol	0.01
den	1

Description The function `rcosflt` passes an input signal through a raised cosine filter. You can either let `rcosflt` design a raised cosine filter automatically or you can specify the raised cosine filter yourself using input arguments.

Designing the Filter Automatically

`y = rcosflt(x,Fd,Fs)` designs a raised cosine FIR filter and then filters the input signal `x` using it. The sample frequency for the digital input signal `x` is `Fd`, and the sample frequency for the output signal `y` is `Fs`. The ratio `Fs/Fd` must be an integer. In the course of filtering, `rcosflt` upsamples the data by a factor of `Fs/Fd`, by inserting zeros between samples. The order of the filter is $1+2*\text{delay}*Fs/Fd$, where `delay` is 3 by default. If `x` is a vector, then the sizes of `x` and `y` are related by this equation.

$$\text{length}(y) = (\text{length}(x) + 2 * \text{delay}) * F_s / F_d$$

Otherwise, y is a matrix, each of whose columns is the result of filtering the corresponding column of x .

$y = \text{rcosflt}(x, F_d, F_s, 'type_flag', r, \text{delay}, \text{tol})$ designs a raised cosine FIR or IIR filter and then filters the input signal x using it. The ratio F_s / F_d must be an integer. r is the rolloff factor for the filter, a real number in the range $[0, 1]$. delay is the filter's group delay, measured in input samples. The actual group delay in the filter design is delay / F_d seconds. The input tol is the tolerance in the IIR filter design. FIR filter design does not use tol .

The characteristics of x , F_d , F_s , and y are as in the first syntax.

The fourth input argument, ' $type_flag$ ', determines the type of filter that rcosflt should design and can have up to three components: filter type, sample frequency, and filter.

Values of filter_type to Determine the Type of Filter

Type of Filter	Value of filter_type
FIR raised cosine filter	fir or fir/normal
IIR raised cosine filter	iir or iir/normal
Square-root FIR raised cosine filter	fir/sqrt
Square-root IIR raised cosine filter	iir/sqrt

$y = \text{rcosflt}(x, F_d, F_s, 'filter_type / F_s', r, \text{delay}, \text{tol})$ is the same as the previous syntax, except that it assumes that x has sample frequency F_s . This syntax does not upsample x any further. If x is a vector, then the relative sizes of x and y are related by this equation.

$$\text{length}(y) = \text{length}(x) + (2 * \text{delay} * F_s / F_d)$$

As before, if x is a nonvector matrix, y is a matrix, each of whose columns is the result of filtering the corresponding column of x .

Specifying the Filter Using Input Arguments

`y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den)` filters the input signal x using a filter whose transfer function numerator and denominator are given in `num` and `den`, respectively. If `type_filter` includes `fir`, then omit `den`. This syntax uses the same arguments x , Fd , Fs , and `type_filter` as explained in the first and second syntaxes above.

`y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den,delay)` uses `delay` in the same way that the `rcosine` function uses it. This syntax assumes that the filter described by `num`, `den`, and `delay` was designed using `rcosine`.

As before, if x is a nonvector matrix, y is a matrix each of whose columns is the result of filtering the corresponding column of x .

`y = rcosflt(x,Fd,Fs,'filter_type/filter/Fs',num,den,...)` is the same as the earlier syntaxes, except that it assumes that x has sample frequency Fs instead of Fd . This syntax does not upsample x any further. If x is a vector, the relative sizes of x and y are related by this equation.

$$\text{length}(y) = \text{length}(x) + (2 * \text{delay} * Fs/Fd)$$

Additional Output

`[y,t] = rcosflt(...)` outputs `t`, a vector that contains the sampling time points of y .

See Also

`rcosine`, `rcosfir`, `rcosiir`, `rcosdemo`, “Special Filters”

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose

Design raised cosine infinite impulse response (IIR) filter

Syntax

```
[num,den] = rcosiir(R,T_delay,rate,T,tol)
[num,den] = rcosiir(R,T_delay,rate,T,tol,type_filter)
rcosiir(...)
rcosiir(...,colr)
[num,den,sample_time] = rcosiir(...)
```

Optional Inputs

Input	Default Value
T_delay	3
rate	5
T	1
tol	0.01

Description

The `rcosiir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes 'iir'. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t / T)}{(\pi t / T)} \cdot \frac{\cos(\pi R t / T)}{(1 - 4R^2 t^2 / T^2)}$$

`[num,den] = rcosiir(R,T_delay,rate,T,tol)` designs an IIR approximation of an FIR raised cosine filter, and returns the numerator and denominator of the IIR filter. The filter's rolloff factor is `R`, a real number between 0 and 1, inclusive. `T` is the symbol period in seconds. The filter's group delay is `T_delay` symbol periods. `rate` is the number of sample points in each interval of duration `T`. `rate` must be greater than 1. The input sample rate is `T` samples per second, while the output sample rate is `T*rate` samples per second. If `tol` is an integer greater than 1, it becomes the order of the IIR filter; if `tol` is less than 1, it

indicates the relative tolerance for `rcosiir` to use when selecting the order based on the singular values.

The arguments `T_delay`, `rate`, `T`, and `tol` are optional inputs whose default values are 3, 5, 1, and 0.01, respectively.

`[num,den] = rcosiir(R,T_delay,rate,T,tol,type_filter)` designs a square-root raised cosine filter if `type_filter` is 'sqrt'. If `type_filter` is 'normal', this syntax is the same as the previous one.

`rcosiir(...)` plots the time and frequency responses of the raised cosine filter.

`rcosiir(...,colr)` uses the string `colr` to determine the plotting color. The choices for `colr` are the same as those listed for the `plot` function.

`[num,den,sample_time] = rcosiir(...)` returns the transfer function and the sample time of the IIR filter.

Examples

The script below compares different values of `T_delay`.

```
rcosiir(0,10);
subplot(211); hold on;
subplot(212); hold on;
col = ['r-';'g-';'b-';'m-';'c-';'w-'];
R = [8,6,4,3,2,1];
for ii = R
    rcosiir(0,ii,[],[],[],[],col(find(R==ii),:));
end;
```

This example shows how the filter's frequency response more closely approximates that of the ideal raised cosine filter as `T_delay` increases.

See Also

`rcosfir`, `rcosflt`, `rcosine`, `rcosdemo`, "Special Filters"

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, N.J., Prentice-Hall, 1980.

[2] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose Design raised cosine filter

Syntax

```
num = rcosine(Fd,Fs)
[num,den] = rcosine(Fd,Fs,type_flag)
[num,den] = rcosine(Fd,Fs,type_flag,r)
[num,den] = rcosine(Fd,Fs,type_flag,r,delay)
[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol)
```

Description `num = rcosine(Fd,Fs)` designs a finite impulse response (FIR) raised cosine filter and returns its transfer function. The digital input signal has sampling frequency F_d . The sampling frequency for the filter is F_s . The ratio F_s/F_d must be a positive integer greater than 1. The default rolloff factor is .5. The filter's group delay, which is the time between the input to the filter and the filter's peak response, is three input samples. Equivalently, the group delay is $3/F_d$ seconds.

`[num,den] = rcosine(Fd,Fs,type_flag)` designs a raised cosine filter using directions in the string variable `type_flag`. Filter types are listed in the table below, along with the corresponding values of `type_flag`.

Types of Filter and Corresponding Values of type_flag

Type of Filter	Value of type_flag
Finite impulse response (FIR)	'default' or 'fir/normal'
Infinite impulse response (IIR)	'iir' or 'iir/normal'
Square-root raised cosine FIR	'sqrt' or 'fir/sqrt'
Square-root raised cosine IIR	'iir/sqrt'

The default tolerance value in IIR filter design is 0.01.

`[num,den] = rcosine(Fd,Fs,type_flag,r)` specifies the rolloff factor, r . The rolloff factor is a real number in the range [0, 1].

rcosine

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay)` specifies the filter's group delay, measured in input samples. `delay` is a positive integer. The actual group delay in the filter design is `delay/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol)` specifies the tolerance in the IIR filter design. FIR filter design does not use `tol`.

See Also

`rcosflt`, `rcosiir`, `rcosfir`, `rcosdemo`, "Special Filters"

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose Rectangular pulse shaping

Syntax `y = rectpulse(x,nsamp)`

Description `y = rectpulse(x,nsamp)` applies rectangular pulse shaping to `x` to produce an output signal having `nsamp` samples per symbol. Rectangular pulse shaping means that each symbol from `x` is repeated `nsamp` times to form the output `y`. If `x` is a matrix with multiple rows, the function treats each column as a channel and processes the columns independently.

Note To insert zeros between successive samples of `x` instead of repeating the samples of `x`, use the `upsample` function instead.

Examples

An example in “Combining 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
ch1 = randint(nsymb,1,2,68521); % 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.

rectpulse

x =

1	1
1	2
0	3

y =

1	1
1	1
1	1
1	1
1	2
1	2
1	2
1	2
0	3
0	3
0	3
0	3

See Also

`intdump`, `upsample`, `rcosflt`

Purpose	Reset channel object
Syntax	<code>reset(chan)</code> <code>reset(chan,randstate)</code>
Description	<p><code>reset(chan)</code> resets the channel object <code>chan</code>, initializing the <code>PathGains</code> and <code>NumSamplesProcessed</code> properties as well as internal filter states. This syntax is useful when you want the effect of creating a new channel.</p> <p><code>reset(chan,randstate)</code> resets the channel object <code>chan</code> and initializes the state of the random number generator that the channel uses. <code>randstate</code> is a two-element column vector. This syntax is useful when you want to repeat previous numerical results that started from a particular state.</p>
Examples	<p>The example below shows how to get 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.</p> <pre>% 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); reset(c,[11; 13]); % Choose arbitrary state. c.ResetBeforeFiltering = 0; % Filter some data. sig = randint(100,1); y1 = [filter(c,sig(1:50)) filter(c,sig(51:end))]; % Try to repeat the results. reset(c,[11; 13]); % Use same state as before. y2 = [filter(c,sig(1:50)) filter(c,sig(51:end))];</pre>

reset (channel)

```
isequal(y1,y2) % y1 and y2 should be the same.
```

The output is below.

```
ans =
```

```
1
```

See Also

rayleighchan, ricianchan, filter, “Fading Channels”

Purpose	Reset equalizer object
Syntax	<code>reset(eqobj)</code>
Description	<code>reset(eqobj)</code> resets the equalizer object <code>eqobj</code> , initializing the <code>Weights</code> , <code>WeightInputs</code> , and <code>NumSamplesProcessed</code> properties and the adaptive algorithm states. If <code>eqobj</code> is a CMA equalizer, <code>reset</code> does not change the <code>Weights</code> property.
See Also	<code>dfe</code> , <code>equalize</code> , <code>lineareq</code> , “Equalizers”

ricianchan

Purpose Construct Rician fading channel object

Syntax

```
chan = ricianchan(ts,fd,k)
chan = ricianchan(ts,fd,k,tau,pdb)
chan = ricianchan(ts,fd,k,tau,pdb,fdLOS)
chan = ricianchan
```

Description `chan = ricianchan(ts,fd,k)` constructs a frequency-flat (single path) Rician fading-channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. `k` is the Rician K-factor in linear scale. You can model the effect of the channel `chan` on a signal `x` by using the syntax `y = filter(chan,x)`. See `filter(channel)` for more information.

`chan = ricianchan(ts,fd,k,tau,pdb)` constructs a frequency-selective (multiple paths) fading-channel object. If `k` is a scalar, then the first discrete path is a Rician fading process (it contains a line-of-sight component) with a K-factor of `k`, while the remaining discrete paths are independent Rayleigh fading processes (no line-of-sight component). If `k` is a vector of the same size as `tau`, then each discrete path is a Rician fading process with a K-factor given by the corresponding element of the vector `k`. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

`chan = ricianchan(ts,fd,k,tau,pdb,fdLOS)` specifies `fdlos` as the Doppler shift(s) of the line-of-sight component(s) of the discrete path(s), in hertz. `fdlos` must be the same size as `k`. If `k` and `fdlos` are scalars, the line-of-sight component of the first discrete path has a Doppler shift of `fdlos`, while the remaining discrete paths are independent Rayleigh fading processes. If `fdlos` is a vector of the same size as `k`, the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector `fdlos`. By default, `fdlos` is 0. The initial phase(s) of the line-of-sight component(s) can be set through the property `DirectPathInitPhase`.

`chan = ricianchan` sets the maximum Doppler shift to 0, the Rician K-factor to 1, and the Doppler shift and initial phase of the line-of-sight

component to 0. This syntax models a static frequency-flat channel, and, in this trivial case, the sample time of the signal is unimportant.

Properties

The following tables describe the properties of the channel object, chan, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see “Viewing Object Properties” or “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.

Writeable Properties (Continued)

Property	Description
DirectPathInitPhase	Initial phase(s) of line-of-sight component(s) in radians. The default value is 0.
NormalizePathGains	If this value is 1, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
StoreHistory	If this value is 1, channel state information needed by the channel visualization tool is stored as the channel filter function processes the signal. The default value is 0.
StorePathGains	If this value is 1, the complex path gain vector is stored as the channel filter function processes the signal. The default value is 0.
ResetBeforeFiltering	If this value is 1, each call to <code>filter</code> resets the state of <code>chan</code> 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.	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 “Simulation of Multipath Fading Channels: Methodology”, where the properties specific to the Rician channel object are related to the quantities of this section as follows (see the `rayleighchan` reference

page for properties common to both Rayleigh and Rician channel objects):

- The `Kfactor` property contains the value of K_r (if it's a scalar) or $\{K_{r,k}\}$, $1 \leq k \leq K$ (if it's a vector).
- The `DirectPathDopplerShift` property contains the value of $f_{d,LOS}$ (if it's a scalar) or $\{f_{d,LOS,k}\}$, $1 \leq k \leq K$ (if it's a vector).
- The `DirectPathInitPhase` property contains the value of θ_{LOS} (if it's a scalar) or $\{\theta_{LOS,k}\}$, $1 \leq k \leq K$ (if it's a vector).

Channel Visualization

The characteristics of a channel can be plotted using the channel visualization tool. See “Using the Channel Visualization Tool” for details.

Examples

The example in “Quasi-Static Channel Modeling” uses this function.

See Also

`rayleighchan`, `filter`, `plot (channel)`, `reset`, “Fading Channels”

References

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

Purpose Construct recursive least squares (RLS) adaptive algorithm object

Syntax

```
alg = rls(forgetfactor)
alg = rls(forgetfactor, invcorr0)
```

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

`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 “Accessing Properties of an Adaptive Algorithm”.

Property	Description
AlgType	Fixed value, 'RLS'
ForgetFactor	Forgetting factor
InvCorrInit	Scalar value used to initialize or reset the diagonal elements of the inverse correlation matrix

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` function or `dfe` function), the equalizer object has an `InvCorrMatrix` property that represents the inverse correlation

matrix for the RLS algorithm. The initial value of `InvCorrMatrix` is `InvCorrInit*eye(N)`, where `N` is the total number of equalizer weights.

Examples

For examples that use this function, see “Defining an Equalizer Object” and “Example: Adaptive Equalization Within a Loop”.

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

$$K = \frac{Pu}{(\text{ForgetFactor}) + u^H Pu}$$

where H denotes the Hermitian transpose.

Then the new inverse correlation matrix is given by

$$(\text{ForgetFactor})^{-1}(P - Ku^H P)$$

and the new set of weights is given by

$$w + K^*e$$

where the $*$ operator denotes the complex conjugate.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `lineareq`, `dfe`, `equalize`, “Equalizers”

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.

Purpose

Reed-Solomon decoder

Note `rsdec` will be removed in a future release. Use `fec.rsdec` instead.

Syntax

```

decoded = rsdec(code,n,k)
decoded = rsdec(code,n,k,genpoly)
decoded = rsdec(...,paritypos)
[decoded,cnumerr] = rsdec(...)
[decoded,cnumerr,ccode] = rsdec(...)

```

Description

`decoded = rsdec(code,n,k)` attempts to decode the received signal in `code` using an $[n,k]$ Reed-Solomon decoding process with the narrow-sense generator polynomial. `code` is a Galois array of symbols having m bits each. Each n -element row of `code` represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol. n is at most 2^m-1 . If n is not exactly 2^m-1 , `rsdec` assumes that `code` is a corrupted version of a shortened code.

In the Galois array `decoded`, each row represents the attempt at decoding the corresponding row in `code`. A *decoding failure* occurs if `rsdec` detects more than $(n-k)/2$ errors in a row of `code`. In this case, `rsdec` forms the corresponding row of `decoded` by merely removing $n-k$ symbols from the end of the row of `code`.

`decoded = rsdec(code,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree $n-k$. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

`decoded = rsdec(...,paritypos)` specifies whether the parity symbols in `code` were appended or prepended to the message in the coding operation. The string `paritypos` can be either `'end'` or

'beginning'. The default is 'end'. If *paritypos* is 'beginning', a decoding failure causes `rsdec` to remove $n-k$ symbols from the beginning rather than the end of the row.

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

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

Examples

The example below encodes three message words using a (7,3) Reed-Solomon encoder. It then corrupts the code by introducing one error in the first codeword, two errors in the second codeword, and three errors in the third codeword. Then `rsdec` tries to decode the corrupted code.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6; 5 1 1],m); % Three rows of m-bit symbols
code = rsenc(msg,n,k);
errors = gf([2 0 0 0 0 0 0; 3 4 0 0 0 0 0; 5 6 7 0 0 0 0],m);
noisycode = code + errors;
[dec,cnumerr] = rsdec(noisycode,n,k)
```

The output is below.

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

```
Array elements =
```

```
    2     7     3
    4     0     6
    0     7     6
```



```
cnumerr =  
  
    1  
    2  
   -1
```

The output shows that `rsdec` successfully corrects the errors in the first two codewords and recovers the first two original message words. However, a $(7,3)$ Reed-Solomon code can correct at most two errors in each word, so `rsdec` cannot recover the third message word. The elements of the vector `cnumerr` indicate the number of corrected errors in the first two words and also indicate the decoding failure in the third word.

For additional examples, see “Creating and Decoding Reed-Solomon Codes”.

Limitations

n and k must differ by an even integer. n must be between 3 and 65535.

Algorithm

`rsdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” on page 2-539 below.

See Also

`rsenc`, `gf`, `rsgenpoly`, “Block Coding”

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.

rsdecof

Purpose	Decode ASCII file encoded using Reed-Solomon code
Syntax	<code>rsdecof(file_in,file_out); rsdecof(file_in,file_out,err_cor);</code>
Description	<p>This function is the inverse process of the function <code>rsencof</code> in that it decodes a file that <code>rsencof</code> encoded.</p> <p><code>rsdecof(file_in,file_out)</code> decodes the ASCII file <code>file_in</code> that was previously created by the function <code>rsencof</code> using an error-correction capability of 5. The decoded message is written to <code>file_out</code>. Both <code>file_in</code> and <code>file_out</code> are string variables.</p> <hr/> <p>Note If the number of characters in <code>file_in</code> is not an integer multiple of 127, the function appends <code>char(4)</code> symbols to the data it must decode. If you encode and then decode a file using <code>rsencof</code> and <code>rsdecof</code>, respectively, the decoded file might have <code>char(4)</code> symbols at the end that the original file does not have.</p> <hr/> <p><code>rsdecof(file_in,file_out,err_cor)</code> is the same as the first syntax, except that <code>err_cor</code> specifies the error-correction capability for each block of 127 codeword characters. The message length is $127 - 2 * err_cor$. The value in <code>err_cor</code> must match the value used in <code>rsencof</code> when <code>file_in</code> was created.</p>
Examples	An example is on the reference page for <code>rsencof</code> .
See Also	<code>rsencof</code> , “Block Coding”

Purpose Reed-Solomon encoder

Note `rsenc` will be removed in a future version. Use `fec.rsenc` instead.

Syntax

```
code = rsenc(msg,n,k)
code = rsenc(msg,n,k,genpoly)
code = rsenc(...,paritypos)
```

Description `code = rsenc(msg,n,k)` encodes the message in `msg` using an $[n,k]$ Reed-Solomon code with the narrow-sense generator polynomial. `msg` is a Galois array of symbols having m bits each. Each k -element row of `msg` represents a message word, where the leftmost symbol is the most significant symbol. n is at most 2^m-1 . If n is not exactly 2^m-1 , `rsenc` uses a shortened Reed-Solomon code. Parity symbols are at the end of each word in the output Galois array `code`.

`code = rsenc(msg,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree $n-k$. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

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

Examples The example below encodes two message words using a (7,3) Reed-Solomon encoder.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6],m); % Two rows of m-bit symbols
code = rsenc(msg,n,k)
```

The output is below.

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

```
Array elements =
```

```
    2    7    3    3    6    7    6  
    4    0    6    4    2    2    0
```

For additional examples, see “Representing Words for Reed-Solomon Codes” and “Creating and Decoding Reed-Solomon Codes”.

Limitations

n and k must differ by an even integer. n must be between 3 and 65535.

See Also

rsdec, gf, rsgenpoly, “Block Coding”

Purpose	Encode ASCII file using Reed-Solomon code
Syntax	<code>rsencof(file_in,file_out); rsencof(file_in,file_out,err_cor);</code>
Description	<p><code>rsencof(file_in,file_out)</code> encodes the ASCII file <code>file_in</code> using (127, 117) Reed-Solomon code. The error-correction capability of this code is 5 for each block of 127 codeword characters. This function writes the encoded text to the file <code>file_out</code>. Both <code>file_in</code> and <code>file_out</code> are string variables.</p> <p><code>rsencof(file_in,file_out,err_cor)</code> is the same as the first syntax, except that <code>err_cor</code> specifies the error-correction capability for each block of 127 codeword characters. The message length is $127 - 2 * \text{err_cor}$.</p>

Note If the number of characters in `file_in` is not an integer multiple of $127 - 2 * \text{err_cor}$, the function appends `char(4)` symbols to `file_out`.

Examples

The file `matlabroot/toolbox/comm/comm/oct2dec.m` contains text help for the `oct2dec` function in this toolbox. The commands below encode the file using `rsencof` and then decode it using `rsdecof`.

```
file_in = [matlabroot '/toolbox/comm/comm/oct2dec.m'];
file_out = 'encodedfile'; % Or use another filename
rsencof(file_in,file_out) % Encode the file.

file_in = file_out;
file_out = 'decodedfile'; % Or use another filename
rsdecof(file_in,file_out) % Decode the file.
```

To see the original file and the decoded file in the MATLAB workspace, use the commands below (or similar ones if you modified the filenames above).

```
type oct2dec.m
```

rsencof

type decodedfile

See Also

rsdecof, “Block Coding”

Purpose Generator polynomial of Reed-Solomon code

Syntax

```
genpoly = rsgenpoly(n,k)
genpoly = rsgenpoly(n,k,prim_poly)
genpoly = rsgenpoly(n,k,prim_poly,b)
[genpoly,t] = rsgenpoly(...)
```

Description `genpoly = rsgenpoly(n,k)` returns the narrow-sense generator polynomial of a Reed-Solomon code with codeword length n and message length k . The codeword length n must have the form $2^m - 1$ for some integer m , 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 $(X - A^1)(X - A^2)\dots(X - A^{2^t})$ where A is a root of the default primitive polynomial for the field $GF(n+1)$ and t is 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 A 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 $(X - A^b)(X - A^{b+1})\dots(X - A^{b+2^t-1})$, where b is an integer, A is a root of `prim_poly`, and t is the code's error-correction capability, $(n - k) / 2$.

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

Examples The examples below create Galois row vectors that represent generator polynomials for a $[7,3]$ Reed-Solomon code. The vectors `g` and `g2` both represent the narrow-sense generator polynomial, but with respect to different primitive elements A . More specifically, `g2` is defined such that A is a root of the primitive polynomial $D^3 + D^2 + 1$ for $GF(8)$, not of the default primitive polynomial $D^3 + D + 1$. The vector `g3` represents

the generator polynomial $(X - A^3)(X - A^4)(X - A^5)(X - A^6)$, where A is a root of $D^3 + D^2 + 1$ in $GF(8)$.

```
g = rsgenpoly(7,3)
g2 = rsgenpoly(7,3,13) % Use nondefault primitive polynomial.
g3 = rsgenpoly(7,3,13,3) % Use b = 3.
```

The output is below.

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

```
Array elements =
```

```
      1      3      1      2      3
```

```
g2 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)
```

```
Array elements =
```

```
      1      4      5      1      5
```

```
g3 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)
```

```
Array elements =
```

```
      1      7      1      6      7
```

As another example, the command below shows that the default narrow-sense generator polynomial for a $[15,11]$ Reed-Solomon code is $X^4 + (A^3 + A^2 + 1)X^3 + (A^3 + A^2)X^2 + A^3X + (A^2 + A + 1)$, where A is a root of the default primitive polynomial for $GF(16)$.

```
gp = rsgenpoly(15,11)
```


gp = GF(2⁴) array. Primitive polynomial = D⁴+D+1 (19 decimal)

Array elements =

1 13 12 8 7

For additional examples, see “Parameters for Reed-Solomon Codes”.

Limitations

n and k must differ by an even integer. The maximum allowable value of n is 65535.

See Also

gf, rsenc, rsdec, “Block Coding”

scatterplot

Purpose Generate scatter plot

Syntax

```
scatterplot(x)
scatterplot(x,n)
scatterplot(x,n,offset)
scatterplot(x,n,offset,plotstring)
scatterplot(x,n,offset,plotstring,h)
h = scatterplot(...)
```

Description `scatterplot(x)` produces a scatter plot for the signal `x`. The interpretation of `x` depends on its shape and complexity:

- If `x` is a real two-column matrix, `scatterplot` interprets the first column as in-phase components and the second column as quadrature components.
- If `x` is a complex vector, `scatterplot` interprets the real part as in-phase components and the imaginary part as quadrature components.
- If `x` is a real vector, `scatterplot` interprets it as a real signal.

`scatterplot(x,n)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the first value. That is, the function decimates `x` by a factor of `n` before plotting.

`scatterplot(x,n,offset)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the `(offset+1)`st value in `x`.

`scatterplot(x,n,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function.

`scatterplot(x,n,offset,plotstring,h)` is the same as the syntax above, except that the scatter plot is in the figure whose handle is `h`, rather than a new figure. `h` must be a handle to a figure that

`scatterplot` previously generated. To plot multiple signals in the same figure, use `hold on`.

`h = scatterplot(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the scatter plot.

Examples

See “Example: Scatter Plots” or the example on the reference page for `qandemod`. Both examples illustrate how to plot multiple signals in a single scatter plot.

For an online demonstration, type `showdemo scattereyedemo`.

See Also

`eyediagram`, `plot`, `scattereyedemo`, `scatter`, “Scatter Plots”

semianalytic

Purpose

Calculate bit error rate (BER) using semianalytic technique

Syntax

```
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,EbNo)
ber =
semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den,EbNo)
[ber,avgamp,avgpower] = semianalytic(...)
```

Graphical Interface

As an alternative to the semianalytic function, invoke the BERTool GUI (bertool) and use the **Semianalytic** tab.

Description

`ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)` returns the bit error rate (BER) of a system that transmits the complex baseband vector signal `txsig` and receives the noiseless complex baseband vector signal `rxsig`. Each of these signals has `Nsamp` samples per symbol. `Nsamp` is also the sampling rate of `txsig` and `rxsig`, in Hz. The function assumes that `rxsig` is the input to the receiver filter, and the function filters `rxsig` with an ideal integrator. `modtype` is the modulation type of the signal and `M` is the alphabet size. The table below lists the valid values for `modtype` and `M`.

Modulation Scheme	Value of <i>modtype</i>	Valid Values of <i>M</i>
Differential phase shift keying (DPSK)	'dpsk'	2, 4
Minimum shift keying (MSK) with differential encoding	'msk/diff'	2
Minimum shift keying (MSK) with nondifferential encoding	'msk/nondiff'	2

Modulation Scheme	Value of <i>modtype</i>	Valid Values of <i>M</i>
Phase shift keying (PSK) with differential encoding, where the phase offset of the constellation is 0	'psk/diff'	2, 4
Phase shift keying (PSK) with nondifferential encoding, where the phase offset of the constellation is 0	'psk/nondiff'	2, 4, 8, 16, 32, or 64
Offset quaternary phase shift keying (OQPSK)	'oqpsk'	4
Quadrature amplitude modulation (QAM)	'qam'	4, 8, 16, 32, 64, 128, 256, 512, 1024

'msk/diff' is equivalent to conventional MSK (setting the 'Precoding' property of the MSK object to 'off'), while 'msk/nondiff' is equivalent to precoded MSK (setting the 'Precoding' property of the MSK object to 'on').

Note The output *ber* is an *upper bound* on the BER in these cases:

- DQPSK (*modtype* = 'dpsk', *M* = 4)
- Cross QAM (*modtype* = 'qam', *M* not a perfect square). In this case, note that the upper bound used here is slightly tighter than the upper bound used for cross QAM in the *berawgn* function.

When the function computes the BER, it assumes that symbols are Gray-coded. The function calculates the BER for values of E_b/N_0 in the range of [0:20] dB and returns a vector of length 21 whose elements correspond to the different E_b/N_0 levels.

Note You must use a sufficiently long vector `txsig`, or else the calculated BER will be inaccurate. If the system's impulse response is L symbols long, the length of `txsig` should be at least M^L . A common approach is to start with an augmented binary pseudonoise (PN) sequence of total length $(\log_2 M)M^L$. An *augmented* PN sequence is a PN sequence with an extra zero appended, which makes the distribution of ones and zeros equal.

`ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den)` is the same as the previous syntax, except that the function filters `rxsig` with a receiver filter instead of an ideal integrator. The transfer function of the receiver filter is given in descending powers of z by the vectors `num` and `den`.

`ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,EbNo)` is the same as the first syntax, except that `EbNo` represents E_b/N_0 , the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, then the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels.

`ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den,EbNo)` combines the functionality of the previous two syntaxes.

`[ber,avgampl,avgpower] = semianalytic(...)` returns the mean complex signal amplitude and the mean power of `rxsig` after filtering it by the receiver filter and sampling it at the symbol rate.

Examples

A typical procedure for implementing the semianalytic technique is in “Procedure for the Semianalytic Technique”. Sample code is in “Example: Using the Semianalytic Technique”.

Limitations

The function makes several important assumptions about the communication system. See “When to Use the Semianalytic Technique” to find out whether your communication system is suitable for the semianalytic technique and the `semianalytic` function.

See Also

`noisebw`, `qfunc`, “Performance Results via the Semianalytic Technique”

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.

seqgen

Purpose Sequence generator package

Syntax `h = seqgen.type(...)`

Description `h = seqgen.type(...)` returns a sequence generator object, `h`, of a particular type (e.g., `h = seqgen.pn`).

Sequence generator objects have sets of properties and methods based on their type. A method common to all `seqgen` object types is `generate`, which is used to generate the specific sequence type.

Type `help seqgen/types` to get the complete listing of types.

Each `seqgen` object is equipped with additional type-specific functions for simulation. Type `help seqgen.<type>` (e.g., `help seqgen.pn`) to get the complete help on the specific type.

Example `h = seqgen.pn; % construct PN Sequence Generator object`

See Also `seqgen.pn`

Purpose Construct default PN sequence generator object

Note The MathWorks™ will remove the `seqgen` function from a future version of the Communications Toolbox software. While the product still supports this function, you should use `commsrc.pn` instead.

Syntax

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

Description `h = seqgen.pn` constructs a default PN sequence generator object `h`, and is equivalent to the following:

```
H = SEQGEN.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 = SEQGEN.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 = seqgen.pn(property1,value1,...)` constructs a PN sequence generator object `h` with properties as specified by pairs of properties and values.

Methods PN sequence generator objects have the following methods.

Method	Result
<code>generate</code>	Generate [<code>NumBitsOut</code> x 1] PN sequence generator values.

Method	Result
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.

The following code shows how to get the 'Shift' or 'Mask' property values:

```
h = seqgen.pn('Shift', 0);  
maskBits = getmask(h)  
shiftVal = getshift(h)
```

Properties

PN sequence generator objects have the following properties.

Property	Description
GenPoly	Generator polynomial vector array of bits; must be descending order
InitialStates	Vector array (with length of the general polynomial order) of initial shift register values (in bits)
CurrentStates	Vector array (with length of the general polynomial order) of present shift register values (in bits)

Property	Description
NumBitsOut	Number of bits to output at each <code>generate</code> method invocation
Mask or Shift	<p>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.</p> <p>Alternatively, a scalar shift value may be used to specify an equivalent shift (either a delay or advance) in the output sequence.</p>

seqgen.pn objects also have either a 'Mask' (vector of mask bits) or 'Shift' (scalar shift value) property.

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 “LFSR SSRG Details” on page 2-558.

Side Effects of Setting Certain Properties

Setting the GenPoly Property

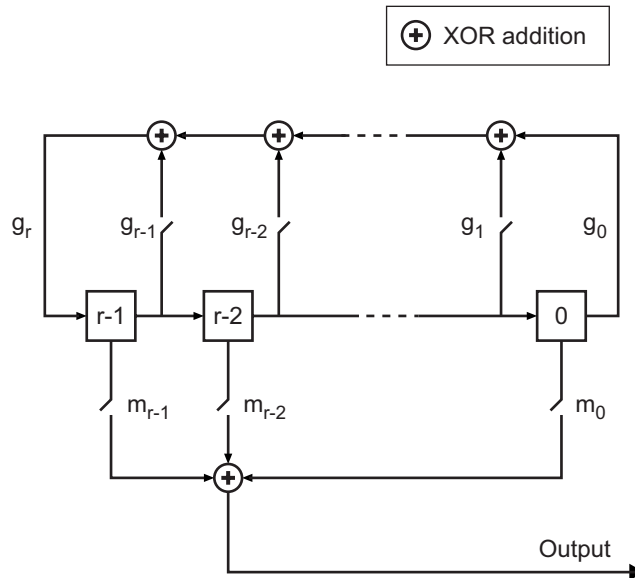
Every time this property is set, it will reset the entire object. In addition to changing the polynomial values, 'CurrentStates', 'InitialStates', and 'Mask' will be set to their default values ('NumBitsOut' will remain the same), and no warnings will be issued.

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 , $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \dots + g_0$. The coefficient g_k is 1 if there is a connection from the k th register, as labeled in the preceding diagram. 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 $\text{length}(g1) - 1$	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 , $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + \dots + m_1z + m_0$, 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 Toolbox function `shift2mask`.

Sequences of Maximum Length

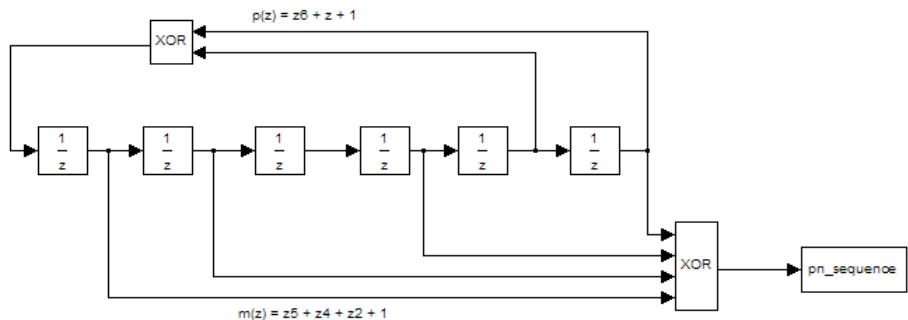
If you want to generate a sequence of the maximum possible length for a fixed degree, r , of the generator polynomial, you can set **Generator polynomial** to a value from the following table. See Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995 for more information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

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

r	Generator Polynomial	r	Generator Polynomial
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 the PN Sequence Generator



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

```
h = seqgen.pn('GenPoly', [1 0 0 0 0 1 1], 'MaskOrShift', [1
1 0 1 0 1])
```

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


```
h = seqgen.pn('GenPoly', [6 1 0], 'MaskOrShift', [1 1 0 1 0
1])
```

General Use of seqgen.pn

The following is an example of typical usage:

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

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

% Output 10 more PN bits
generate(h)

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

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

Behavior of a Copied seqgen.pn Object

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

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

h generates the sequence:

```
1
0
0
```

```
0  
0
```

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

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

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

```
0  
1  
0  
0  
0
```

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

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

then it generates the same sequence that `h` did:

```
1  
0  
0  
0  
0
```

See Also

`mask2shift`, `seqgen`, `shift2mask`

Purpose Convert shift to mask vector for shift register configuration

Syntax `mask = shift2mask(prpoly,shift)`

Description `mask = shift2mask(prpoly,shift)` returns the mask that is equivalent to the shift (or offset) specified by `shift`, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `shift` input is an integer scalar.

Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

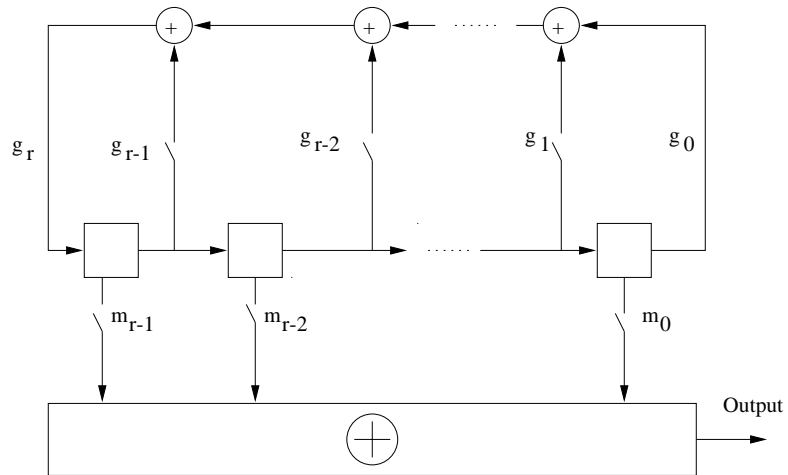
Definition of Equivalent Mask

The equivalent mask for the shift s is the remainder after dividing the polynomial x^s by the primitive polynomial. The vector `mask` represents the remainder polynomial by listing the coefficients in order of descending powers.

Shifts, Masks, and Pseudonoise Sequence Generators

Linear feedback shift registers are part of an implementation of a pseudonoise sequence generator. Below is a schematic diagram of a pseudonoise sequence generator. All adders perform addition modulo 2.

shift2mask



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 Blockset software and want to generate a pseudonoise sequence in a Simulink® model, see the reference page for the PN Sequence Generator block in the blockset's documentation set.

Examples

The command below converts a shift of 5 into the equivalent mask $x^3 + x + 1$, for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$.

```
mk = shift2mask([1 1 0 0 1],5)
```

```
mk =
```

```
1    0    1    1
```

See Also

mask2shift, deconv, isprimitive, primpoly

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.

signlms

Purpose Construct signed least mean square (LMS) adaptive algorithm object

Syntax
`alg = signlms(stepsize)`
`alg = signlms(stepsize,algtype)`

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

`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 <code>algtype</code>	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 “Accessing Properties of an Adaptive Algorithm”.

Property	Description
AlgType	Type of signed LMS algorithm, corresponding to the <i>algtype</i> 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.

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

- $(\text{LeakageFactor}) w + (\text{StepSize}) u^* \text{sgn}(\text{Re}(e))$, for sign LMS
- $(\text{LeakageFactor}) w + (\text{StepSize}) \text{sgn}(\text{Re}(u)) \text{Re}(e)$, for signed regressor LMS
- $(\text{LeakageFactor}) w + (\text{StepSize}) \text{sgn}(\text{Re}(u)) \text{sgn}(\text{Re}(e))$, for sign-sign LMS

where the $*$ operator denotes the complex conjugate and sgn denotes the signum function (sign in MATLAB technical computing software).

See Also

lms, normlms, varlms, rls, cma, lineareq, dfe, equalize, “Equalizers”

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.

Purpose Single sideband amplitude demodulation

Syntax

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

Description **For All Syntaxes**

`z = ssbdemod(y,Fc,Fs)` demodulates the single sideband amplitude modulated signal `y` from the carrier signal having frequency `Fc` (Hz). The carrier signal and `y` have sampling rate `Fs` (Hz). The modulated signal has zero initial phase, and can be an upper- or lower-sideband signal. The demodulation process uses the lowpass filter specified by `[num,den] = butter(5,Fc*2/Fs)`.

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

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

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

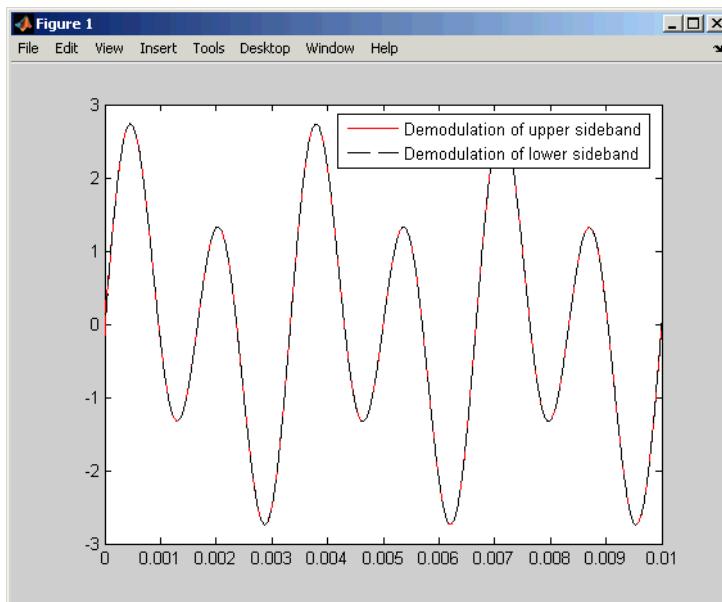
Examples

The code below shows that `ssbdemod` can demodulate an upper-sideband or lower-sideband signal.

```
Fc = 12000; Fs = 270000;
t = [0:1/Fs:0.01]';
s = sin(2*pi*300*t)+2*sin(2*pi*600*t);
y1 = ssbmod(s,Fc,Fs,0); % Lower-sideband modulated signal
y2 = ssbmod(s,Fc,Fs,0,'upper'); % Upper-sideband modulated signal
s1 = ssbdemod(y1,Fc,Fs); % Demodulate lower sideband
s2 = ssbdemod(y2,Fc,Fs); % Demodulate upper sideband
% Plot results to show that the curves overlap.
figure; plot(t,s1,'r-',t,s2,'k--');
```

ssbdemod

```
legend('Demodulation of upper sideband','Demodulation of lower sideband')
```



See Also

ssbmod, amdemod, “Modulation”

Purpose Single sideband amplitude modulation

Syntax

```
y = ssbmod(x,Fc,Fs)
y = ssbmod(x,Fc,Fs,ini_phase)
y = ssbmod(x,fc,fs,ini_phase,'upper')
```

Description `y = ssbmod(x,Fc,Fs)` uses the message signal `x` to modulate a carrier signal with frequency `Fc` (Hz) using single sideband amplitude modulation in which the lower sideband is the desired sideband. The carrier signal and `x` have sample frequency `Fs` (Hz). The modulated signal has zero initial phase.

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

`y = ssbmod(x,fc,fs,ini_phase,'upper')` uses the upper sideband as the desired sideband.

Examples An example using `ssbmod` is on the reference page for `ammod`.

See Also `ssbdemod`, `ammod`, “Modulation”

stdchan

Purpose Construct channel object from set of standardized channel models

Syntax
`chan = stdchan(ts,fd,chantype)`
`[chan, chanprofile] = stdchan(...)`

Description `chan = stdchan(ts,fd,chantype)` constructs a fading channel object `chan` according to the specified `chantype`. The input string `chantype` is chosen from the set of standardized channel profiles listed below. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in Hertz.

`[chan, chanprofile] = stdchan(...)` also returns a structure `chanprofile` containing the parameters of the channel profile specified by `chantype`.

Channel Models

COST 207 channel models (The Rician K factors for the cases `cost207RAx4` and `cost207RAx6` are chosen as in 3GPP TS 45.005 V7.9.0 (2007-2)):

Channel model	Profile
<code>cost207RAx4</code>	Rural Area (RAx), 4 taps
<code>cost207RAx6</code>	Rural Area (RAx), 6 taps
<code>cost207TUx6</code>	Typical Urban (TUx), 6 taps
<code>cost207TUx6alt</code>	Typical Urban (TUx), 6 taps, alternative
<code>cost207TUx12</code>	Typical Urban (TUx), 12 taps
<code>cost207TUx12alt</code>	Typical Urban (TUx), 12 taps, alternative
<code>cost207BUx6</code>	Bad Urban (BUx), 6 taps
<code>cost207BUx6alt</code>	Bad Urban (BUx), 6 taps, alternative
<code>cost207BUx12</code>	Bad Urban (BUx), 12 taps

Channel model	Profile
cost207BUx12alt	Bad Urban (BUx), 12 taps, alternative
cost207HTx6	Hilly Terrain (HTx), 6 taps
cost207HTx6alt	Hilly Terrain (HTx), 6 taps, alternative
cost207HTx12	Hilly Terrain (HTx), 12 taps
cost207HTx12alt	Hilly Terrain (HTx), 12 taps, alternative

GSM/EDGE channel models (3GPP TS 45.005 V7.9.0 (2007-2), 3GPP TS 05.05 V8.20.0 (2005-11)):

Channel model	Profile
gsmRAx6c1	Typical case for rural area (RAx), 6 taps, case 1
gsmRAx4c2	Typical case for rural area (RAx), 4 taps, case 2
gsmHTx12c1	Typical case for hilly terrain (HTx), 12 taps, case 1
gsmHTx12c2	Typical case for hilly terrain (HTx), 12 taps, case 2
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

Channel model	Profile
gsmTUx6c1	Typical case for urban area (TUx), 6 taps, case 1
gsmTUx6c2	Typical case for urban area (TUx), 6 taps, case 2
gsmEQx6	Profile for equalization test (EQx), 6 taps
gsmTIx2	Typical case for very small cells (TIx), 2 taps

3GPP channel models for deployment evaluation (3GPP TR 25.943 V6.0.0 (2004-12)):

Channel model	Profile
3gppTUx	Typical Urban channel model (TUx)
3gppRAx	Rural Area channel model (RAx)
3gppHTx	Hilly Terrain channel model (HTx)

ITU-R 3G channel models (ITU-R M.1225 (1997-2)):

Channel model	Profile
itur3GIAx	Indoor office, channel A
itur3GIBx	Indoor office, channel B
itur3GPAX	Outdoor to indoor and pedestrian, channel A
itur3GPBx	Outdoor to indoor and pedestrian, channel B
itur3GVAx	Vehicular - high antenna, channel A

Channel model	Profile
itur3GVBx	Vehicular - high antenna, channel B
itur3GSAXLOS	Satellite, channel A, LOS
itur3GSAXNLOS	Satellite, channel A, NLOS
itur3GSBxLOS	Satellite, channel B, LOS
itur3GSBxNLOS	Satellite, channel B, NLOS
itur3GSCxLOS	Satellite, channel C, LOS
itur3GSCxNLOS	Satellite, channel C, NLOS

ITU-R HF channel models (ITU-R F.1487 (2000)) (FD must be 1 to obtain the correct frequency spreads for these models.):

Channel model	Profile
iturHFLQ	Low latitudes, Quiet conditions
iturHFLM	Low latitudes, Moderate conditions
iturHFLD	Low latitudes, Disturbed conditions
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

Channel model	Profile
iturHFHM	High latitudes, Moderate conditions
iturHFHD	High latitudes, Disturbed conditions

JTC channel models:

Channel model	Profile
jtcInResA	Indoor residential A
jtcInResB	Indoor residential B
jtcInResC	Indoor residential C
jtcInOffA	Indoor office A
jtcInOffB	Indoor office B
jtcInOffC	Indoor office C
jtcInComA	Indoor commercial A
jtcInComB	Indoor commercial B
jtcInComC	Indoor commercial C
jtcOutUrbHRLAA	Outdoor urban high-rise areas - Low antenna A
jtcOutUrbHRLAB	Outdoor urban high-rise areas - Low antenna B
jtcOutUrbHRLAC	Outdoor urban high-rise areas - Low antenna C
jtcOutUrbLRLAA	Outdoor urban low-rise areas - Low antenna A

Channel model	Profile
jtcOutUrbLRLAB	Outdoor urban low-rise areas - Low antenna B
jtcOutUrbLRLAC	Outdoor urban low-rise areas - Low antenna C
jtcOutResLAA	Outdoor residential areas - Low antenna A
jtcOutResLAB	Outdoor residential areas - Low antenna B
jtcOutResLAC	Outdoor residential areas - Low antenna C
jtcOutUrbHRHAA	Outdoor urban high-rise areas - High antenna A
jtcOutUrbHRHAB	Outdoor urban high-rise areas - High antenna B
jtcOutUrbHRHAC	Outdoor urban high-rise areas - High antenna C
jtcOutUrbLRHAA	Outdoor urban low-rise areas - High antenna A
jtcOutUrbLRHAB	Outdoor urban low-rise areas - High antenna B
jtcOutUrbLRHAC	Outdoor urban low-rise areas - High antenna C
jtcOutResHAA	Outdoor residential areas - High antenna A
jtcOutResHAB	Outdoor residential areas - High antenna B
jtcOutResHAC	Outdoor residential areas - High antenna C

HIPERLAN/2 channel models:

Channel model	Profile
hiperlan2A	Model A
hiperlan2B	Model B
hiperlan2C	Model C
hiperlan2D	Model D
hiperlan2E	Model E

802.11a/b/g channel models:

802.11a/b/g channel models share a common multipath delay profile

Note TS should not be larger than TRMS/2, as per 802.11 specifications.

Channel model
802.11a
802.11b
802.11g

Example

```
ts = 0.1e-4; fd = 200;
chan = stdchan(ts, fd, 'cost207TUx6');
chan.NormalizePathGains = 1;
chan.StoreHistory = 1;
y = filter(chan, ones(1,5e4));
plot(chan);
```

See Also

doppler, rayleighchan, and ricianchan

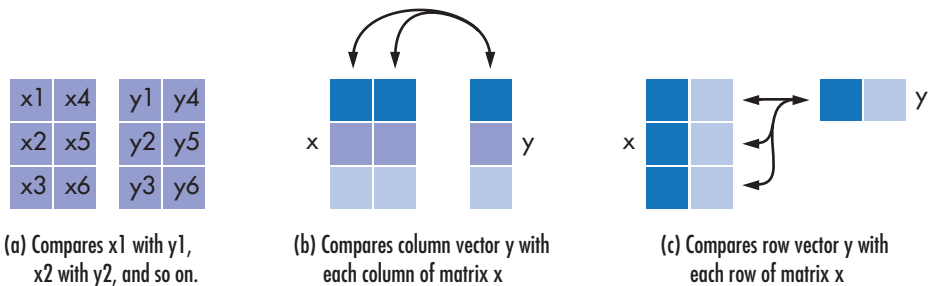
Purpose Compute number of symbol errors and symbol error rate

Syntax

```
[number,ratio] = symerr(x,y)
[number,ratio] = symerr(x,y,flag)
[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 `flag` 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 *m*th entry indicates the number of elements that differ when comparing the vector with the *m*th row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

`[number,ratio] = symerr(x,y,flag)` is similar to the previous syntax, except that *flag* can override the defaults that govern which elements `symerr` compares and how `symerr` computes the outputs. The values of *flag* 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 <i>y</i>	<i>flag</i>	Type of Comparison	<i>number</i>
Two-dim. matrix	'overall' (default)	Element by element	Total number of symbol errors
	'column-wise'	<i>m</i> th column of <i>x</i> vs. <i>m</i> th column of <i>y</i>	Row vector whose entries count symbol errors in each column
	'row-wise'	<i>m</i> th row of <i>x</i> vs. <i>m</i> th row of <i>y</i>	Column vector whose entries count symbol errors in each row

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

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

`[number, ratio, loc] = symerr(...)` returns a binary matrix `loc` that indicates which elements of x and y differ. An element of `loc` is zero if the corresponding comparison yields no discrepancy, and one otherwise.

Examples

On the reference page for `biterr`, the last example uses `symerr`.

The command below illustrates how `symerr` works when one argument is a vector and the other is a matrix. It compares the vector `[1,2,3]'` to the columns

$$\begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 3 \\ 2 \\ 8 \end{bmatrix}, \text{ and } \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

of the matrix.

```
num = symerr([1 2 3]',[1 1 3 1;3 2 2 2; 3 3 8 3])
```

```
num =
```

```
1    0    2    0
```

As another example, the command below illustrates the use of *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')
```

The output is below.

```
number =
```

```
3
```

```
ratio =
```

```
3/4
```

```
loc =
```

```
0    1  
1    1
```

See Also

biterr, “Performance Results via Simulation”

Purpose	Produce syndrome decoding table
Syntax	<code>t = syndtable(h)</code>
Description	<p><code>t = syndtable(h)</code> 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.</p> <p>When converting between binary and decimal values, the leftmost column is interpreted as the <i>most</i> significant digit. This differs from the default convention in the <code>bi2de</code> and <code>de2bi</code> commands.</p>
Examples	An example is in "Decoding Table".
See Also	<code>decode</code> , <code>hammgen</code> , <code>gfcosets</code> , "Block Coding"
References	[1] Clark, George C., Jr., and J. Bibb Cain, <i>Error-Correction Coding for Digital Communications</i> , New York, Plenum, 1981.

varlms

Purpose Construct variable-step-size least mean square (LMS) adaptive algorithm object

Syntax `alg = varlms(initstep,incstep,minstep,maxstep)`

Description The `varlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfc` 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 “Using Adaptive Equalizer Functions and Objects”.

`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 “Accessing Properties of an Adaptive Algorithm”.

Property	Description
AlgType	Fixed value, 'Variable Step Size LMS'
LeakageFactor	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.
InitStep	Initial value of step size when the algorithm starts

Property	Description
IncStep	Increment by which the step size changes from iteration to iteration
MinStep	Minimum value of step size
MaxStep	Maximum value of step size

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` or `dfe` function), the equalizer object has a `StepSize` property. The property value is a vector that lists the current step size for each weight in the equalizer.

Examples

For an example that uses this function, see “Linked Properties of an Equalizer Object”.

Algorithm

Referring to the schematics presented in “Overview of Adaptive Equalizer Classes”, 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

$$\mu_0 = \mu + (\text{IncStep}) \text{Re}(g g_{\text{prev}})$$

where $g = u e^*$, 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 $\mu_0 < \text{MinStep}$
- `MaxStep`, if $\mu_0 > \text{MaxStep}$

The new set of weights is given by

$$(\text{LeakageFactor}) w + 2 \mu g^*$$

varlms

See Also lms, signlms, normlms, rls, cma, lineareq, dfe, equalize, “Equalizers”

References [1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Purpose

Convert vector into matrix

Syntax

```
mat = vec2mat(vec,matcol)
mat = vec2mat(vec,matcol,padding)
[mat,padded] = vec2mat(...)
```

Description

`mat = vec2mat(vec,matcol)` converts the vector `vec` into a matrix with `matcol` columns, creating one row at a time. If the length of `vec` is not a multiple of `matcol`, then extra zeros are placed in the last row of `mat`. The matrix `mat` has `ceil(length(vec)/matcol)` rows.

`mat = vec2mat(vec,matcol,padding)` is the same as the first syntax, except that the extra entries placed in the last row of `mat` are not necessarily zeros. The extra entries are taken from the matrix `padding`, in order. If `padding` has fewer entries than are needed, then the last entry is used repeatedly.

`[mat,padded] = vec2mat(...)` returns an integer `padded` that indicates how many extra entries were placed in the last row of `mat`.

Note `vec2mat` is similar to the built-in MATLAB function `reshape`. However, given a vector input, `reshape` creates a matrix one *column* at a time instead of one row at a time. Also, `reshape` requires the input and output matrices to have the same number of entries, whereas `vec2mat` places extra entries in the output matrix if necessary.

Examples

```
vec = [1 2 3 4 5];
[mat,padded] = vec2mat(vec,3)
[mat2,padded2] = vec2mat(vec,4)
mat3 = vec2mat(vec,4,[10 9 8; 7 6 5; 4 3 2])
```

The output is below.

vec2mat

mat =

1	2	3
4	5	0

padded =

1

mat2 =

1	2	3	4
5	0	0	0

padded2 =

3

mat3 =

1	2	3	4
5	10	7	4

See Also

[reshape](#)

Purpose

Convolutionally decode binary data using Viterbi algorithm

Syntax

```

decoded = vitdec(code,trellis,tblen,opmode,dectype)
decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec)
decoded = ...
    vitdec(code,trellis,tblen,opmode,dectype,puncpat)
decoded = ...
    vitdec(code,trellis,tblen,opmode,dectype,puncpat,eraspat)
decoded = ...
    vitdec(...,'cont',...,initmetric,initstates,initinputs)
[decoded,finalmetric,finalstates,finalinputs] = ...
    vitdec(...,'cont',...)

```

Description

`decoded = vitdec(code,trellis,tblen,opmode,dectype)` decodes the vector `code` using the Viterbi algorithm. The MATLAB structure `trellis` specifies the convolutional encoder that produced `code`; the format of `trellis` is described in “Trellis Description of a Convolutional Encoder” and the reference page for the `istrellis` function. `code` contains one or more symbols, each of which consists of $\log_2(\text{trellis.numOutputSymbols})$ bits. Each symbol in the vector `decoded` consists of $\log_2(\text{trellis.numInputSymbols})$ bits. `tblen` is a positive integer scalar that specifies the traceback depth. If the code rate is $1/2$, a typical value for `tblen` is about five times the constraint length of the code.

The string `opmode` indicates the decoder’s operation mode and its assumptions about the corresponding encoder’s operation. Choices are in the table below.

Values of `opmode` Input

Value	Meaning
'cont'	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. A delay equal to <code>tblen</code> symbols elapses before the first decoded symbol appears in the output. This mode is appropriate when you invoke this function repeatedly and want to preserve continuity between successive invocations. See the continuous operation mode syntaxes below.
'term'	The encoder is assumed to have both started and ended at the all-zeros state, which is true for the default syntax of the <code>convenc</code> 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 <code>convenc</code>) has enough zeros at the end to fill all memory registers of the encoder. If the encoder has <code>k</code> input streams and constraint length vector <code>constr</code> (using the polynomial description of the encoder), “enough” means $k * \max(\text{constr} - 1)$.
'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.

The string *dectype* indicates the type of decision that the decoder makes, and influences the type of data the decoder expects in `code`. Choices are in the table below.

Values of dectype Input

Value	Meaning
'unquant'	code contains real input values, where 1 represents a logical zero and -1 represents a logical one.
'hard'	code contains binary input values.
'soft'	For soft-decision decoding, use the syntax below. nsdec is required for soft-decision decoding.

Syntax for Soft Decision Decoding

`decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec)` decodes the vector code using soft-decision decoding. `code` consists of integers between 0 and $2^{nsdec}-1$, where 0 represents the most confident 0 and $2^{nsdec}-1$ represents the most confident 1. The existing implementation of the functionality supports up to 13 bits of quantization, meaning `nsdec` can be set up to 13. For reference, 3 bits of quantization is about 2 db better than hard decision decoding.

Syntax for Punctures and Erasures

`decoded = ...`

`vitdec(code,trellis,tblen,opmode,dectype,puncpat)` denotes the input punctured code, where `puncpat` is the puncture pattern vector, and where 0s 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 1s 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, 1s 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.
```

Examples

The example below encodes random data and adds noise. Then it decodes the noisy code three times to illustrate the three decision types that `vitdec` supports. For unquantized and soft decisions, the output of `convenc` does not have the same data type that `vitdec` expects for

the input code, so it is necessary to manipulate `ncode` before invoking `vitdec`. That the bit error rate computations must account for the delay that the continuous operation mode incurs.

```
% Encode data bits
trell = poly2trellis(3,[6 7]); % Define trellis
msg = randi([0 1],1000,1); % Random data
code = convenc(msg,trell); % Encode
tblen = 5; % Traceback length

% Map "0" bit to 1.0 and "1" bit to -1.0. Also add AWGN.
ucode = real(awgn(1-2*code, 3, 'measured'));

% Hard decision decoding using binary inputs
hcode = ucode<0;
decoded1 = vitdec(hcode,trell,tblen,'cont','hard');

% Soft decision decoding with quantized inputs
[x,qcode] = quantiz(ucode,[-.75 -.5 -.25 0 .25 .5 .75],...
7:-1:0); % Values in qcode are between 0 and 2^3-1.
decoded2 = vitdec(qcode',trell,tblen,'cont','soft',3);

% Soft decision decoding using unquantized inputs
decoded3 = vitdec(ucode,trell,tblen,'cont','unquant');

% Compute bit error rates, using the fact that the decoder
% output is delayed by tblen symbols.
[n1,r1] = biterr(double(decoded1(tblen+1:end)),msg(1:end-tblen));
[n2,r2] = biterr(decoded2(tblen+1:end),msg(1:end-tblen));
[n3,r3] = biterr(decoded3(tblen+1:end),msg(1:end-tblen));
disp(['The bit error rates are: ',num2str([r1 r2 r3])])
```

The output is similar to the following:

```
The bit error rates are:  0.064322    0.020101    0.01608
```

The example below illustrates how to use the final state and initial state arguments when invoking `vitdec` repeatedly. [`decoded4`;`decoded5`] is the same as `decoded6`.

```

trell = poly2trellis(3,[6 7]); % Define trellis
tblen = 5; % Traceback length
bitsPerPk = 100; % number of bits per package
SNR = 3;
% Initialize the initial states of the encoder and ...
%     the decoder to default values
encState = []; decMetric = []; decState = []; decInput = [];
totalNumErr = 0; % Initialize total number of bit errors
% Main loop
for pkCount = 1:10
    msg = randi([0 1],bitsPerPk,1); % Generate random data
    % Encode starting from encState and save last state
    [code encState] = convenc(msg,trell,encState);
    % Map "0" bit to 1.0 and "1" bit to -1.0. Also add AWGN.
    ucode = real(awgn(1-2*code, SNR, 'measured'));
    % Soft decision decoding using unquantized inputs. Start with the
    % provided state and save the last state of the decoder.
    [decoded,decMetric,decState,decInput] = ...
        vitdec(ucode,trell,tblen,'cont','unquant',[], ...
zeros(bitsPerPk*2,1),...
        decMetric,decState,decInput);
    % Compute bit error rates, using the fact that the decoder
    % output is delayed by tblen symbols.
    if (pkCount == 1)
        numErr = biterr(decoded(tblen+1:end),msg(1:end-tblen));
    else
        numErr = biterr(decoded,[prevMsg; msg(1:end-tblen)]);
    end
    totalNumErr = totalNumErr + numErr;
    prevMsg = msg(end-tblen+1:end);
end
% Compute the bit error rate
BER = totalNumErr / (pkCount*bitsPerPk-tblen)

```

The output is similar to the following:

```
BER =  
  
0.0050
```

For additional examples, see “Examples of Convolutional Coding”.

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

See Also

convenc, poly2trellis, istrellis, vitsimdemo, viterbisim,
“Convolutional Coding”

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.

Purpose Generate white Gaussian noise

Syntax

```
y = wgn(m,n,p)
y = wgn(m,n,p,imp)
y = wgn(m,n,p,imp,state)
y = wgn(...,powertype)
y = wgn(...,outputtype)
```

Description `y = wgn(m,n,p)` generates an m -by- n matrix of white Gaussian noise. p specifies the power of y in decibels relative to a watt. The default load impedance is 1 ohm.

`y = wgn(m,n,p,imp)` is the same as the previous syntax, except that `imp` specifies the load impedance in ohms.

`y = wgn(m,n,p,imp,state)` is the same as the previous syntax, except that `wgn` first resets the state of the normal random number generator `randn` to the integer state.

`y = wgn(...,powertype)` is the same as the previous syntaxes, except that the string `powertype` specifies the units of p . Choices for `powertype` are 'dBW', 'dBm', and 'linear'.

`y = wgn(...,outputtype)` is the same as the previous syntaxes, except that the string `outputtype` specifies whether the noise is real or complex. Choices for `outputtype` are 'real' and 'complex'. If `outputtype` is 'complex', then the real and imaginary parts of y each have a noise power of $p/2$.

Note The unit of measure for the output of the `wgn` function is Volts. For power calculations, it is assumed that there is a load of 1 Ohm.

Examples To generate a column vector of length 100 containing real white Gaussian noise of power 0 dBW, use this command:

```
y1 = wgn(100,1,0);
```

wgn

To generate a column vector of length 100 containing complex white Gaussian noise, each component of which has a noise power of 0 dBW, use this command:

```
y2 = wgn(100,1,0,'complex');
```

See Also

randn, awgn, “Signal Sources”

A

- algdeintrlv function 2-2
- algintrlv function 2-4
- alignsignals function 2-7
- amdemod function 2-11
- ammod function 2-13
- arithdeco function 2-15
- arithenco function 2-16
- asymmetrical Jakes Doppler spectrum
 - object 2-179
- awgn function 2-17

B

- BCH coding
 - sample code
 - using various coding methods 2-214
- bchdec function 2-19
- bchenc function 2-24
- bchgenpoly function 2-31
- bchnumerr function 2-33
- berawgn function 2-34
- bercoding function 2-38
- berconfint function 2-42
- berfading function 2-44
- berfit function 2-49
- bersync function 2-58
- bertool function 2-61
- bi-Gaussian Doppler spectrum object 2-185
- bi2de function 2-62
- bin2gray function 2-64
- binary matrix format
 - sample code 2-213
- binary vector format
 - sample code 2-213
- binary-to-decimal conversion 2-62
- biterr function 2-66
- Bose-Chaudhuri-Hocquenghem (BCH) coding
 - sample code
 - using various coding methods 2-214

C

- channel visualization tool
 - plot (channel) 2-474
- cma function 2-75
- commscope package 2-77
- commscope.eyediagram 2-78
- compand function 2-132
- conjugate elements in Galois fields
 - even number of field elements 2-144
 - odd number of field elements 2-287
- constellations
 - hexagonal
 - sample code 2-277
- convdeintrlv function 2-135
- convenc function 2-137
- conversion
 - binary to decimal 2-62
 - decimal to binary 2-160
 - octal to decimal 2-469
 - vectors to matrices 2-589
- convintrlv function 2-140
- convmtx function 2-142
- cosets
 - even number of field elements 2-144
 - odd number of field elements 2-287
- cosets function 2-144
- cyclic coding
 - sample code 2-214
 - for tracking errors 2-165
 - using various coding methods 2-214
- cyclotomic cosets
 - even number of field elements 2-144
 - odd number of field elements 2-287
- cyclpoly function 2-157

D

- de2bi function 2-160
- decimal format
 - sample code 2-213

decode function 2-163
deintrlv function 2-167
dfe function 2-168
dftmtx function 2-172
distspec function 2-174
doppler package 2-178
doppler.ajakes object 2-179
doppler.bigaussian object 2-185
doppler.flat object 2-189
doppler.gaussian object 2-191
doppler.jakes object 2-194
doppler.rjakes object 2-196
doppler.rounded object 2-199
dpcmdeco function 2-202
dpcmenco function 2-203
dpcmopt function 2-204
dpskdemod function 2-206
dpskmod function 2-208
dvbs2ldpc function 2-210

E

encode function 2-211
equalize function 2-216
eyediagram function 2-218

F

fec.ldpcdec object 2-236
fec.ldpcenc object 2-244
fft function 2-260
filter function
 as a channel 2-261
 Galois fields 2-262
filters
 Galois fields
 odd number of field elements 2-294
finddelay function 2-263
flat Doppler spectrum object 2-189
fmdemod function 2-268

fmod function 2-269
fskdemod function 2-270
fskmod function 2-272

G

Gaussian Doppler spectrum object 2-191
gen2par function 2-274
genqamdemod function 2-276
genqammod function 2-277
gf function 2-279
gfadd function 2-282
gfconv function 2-284
gfcosets function 2-287
gfdeconv function 2-289
gfdiv function 2-292
gffilter function 2-294
gflineq function 2-296
gfminpol function 2-298
gfmul function 2-300
gfpretty function 2-302
gfprimck function 2-304
gfprimdf function 2-306
gfprimfd function 2-308
gfrank function 2-311
gfrepcov function 2-312
gfroots function 2-314
gfsub function 2-316
gfstable function 2-318
gftrunc function 2-319
gftuple function 2-320
gfweight function 2-324
gray2bin function 2-326

H

hammgen function 2-328
Hamming coding
 sample code
 using various coding methods 2-214

- using various formats 2-213
- Hamming weight 2-324
- hank2sys function 2-331
- heldeintrlv function 2-333
- helintrlv function 2-336
- helscandintrlv function 2-340
- helscanintrlv function 2-342
- hilbiir function 2-344
- huffmandeco function 2-348
- huffmandict function 2-350
- huffmanenco function 2-353

I

- ifft function 2-354
- intdump function 2-355
- intrlv function 2-356
- inverses in Galois fields
 - odd number of elements 2-292
- iscatastrophic function 2-357
- isprimitive function 2-358
- istrellis function 2-360

J

- Jakes Doppler spectrum object 2-194

L

- LDPC decoder object 2-236
- LDPC encoder object 2-244
- lineareq function 2-363
- lloyds function 2-367
- LLR algorithm 2-238
- lms function 2-370
- log function 2-372
- log-likelihood ratio (LLR) 2-238

M

- marcumq function 2-373

- mask2shift function 2-375
- matdeintrlv function 2-377
- matintrlv function 2-379
- minimum distance 2-324
- minpol function 2-388
- mldivide function 2-390
- mlseeq function 2-392
- modem package 2-396
- modem.dpskdemod object 2-397
- modem.dpskmod object 2-401
- modem.genqamdemod object 2-405
- modem.genqammod object 2-409
- modem.mskdemod object 2-413
- modem.mskmod object 2-417
- modem.oqpskdemod object 2-420
- modem.oqpskmod object 2-425
- modem.pamdemod object 2-429
- modem.pammod object 2-433
- modem.pskdemod object 2-437
- modem.pskmod object 2-442
- modem.qamdemod object 2-446
- modem.qammod object 2-451
- modnorm function 2-455
- mskdemod function 2-457
- mskmod function 2-460
- multiple roots over Galois fields
 - odd number of field elements 2-314
- muxdeintrlv function 2-462
- muxintrlv function 2-464

N

- noisebw function 2-465
- normlms function 2-467

O

- oct2dec function 2-469
- octal
 - conversion to decimal 2-469

oqpskdemod function 2-470
oqpskmod function 2-471

P

pamdemod function 2-472
pammod function 2-473
plot (channel) function 2-474
pmdemod function 2-475
pmmod function 2-476
PN Sequence generator object 2-555
poly2trellis function 2-477
primpoly function 2-481
pskdemod function 2-484
pskmod function 2-487

Q

qamdemod function 2-488
qammod function 2-490
qfunc function 2-491
qfuncinv function 2-492
quantiz function 2-494

R

randdeintrlv function 2-496
randerr function 2-497
randint function 2-500
randintrlv function 2-502
randsrc function 2-503
rank
 in Galois fields
 odd number of elements 2-311
rayleighchan function 2-505
rcosfir function 2-512
rcosflt function 2-515
rcosiir function 2-518
rcosine function 2-521
rectpulse function 2-523

reset function
 for channels 2-525
 for equalizers 2-527
restricted Jakes Doppler spectrum object 2-196
ricianchan function 2-528
rls function 2-534
rounded Doppler spectrum object 2-199
rsdec function 2-537
rsdecof function 2-540
rsenc function 2-541
rsencof function 2-543
rsgenpoly function 2-545

S

scatterplot function 2-548
semianalytic function 2-550
seqgen package 2-554
seqgen.pn function 2-555
shift2mask function 2-565
signal constellations
 hexagonal
 sample code 2-277
signlms function 2-568
ssbdemod function 2-571
ssbmod function 2-573
stdchan function 2-574
symerr function 2-581
syndtable function 2-585

V

varlms function 2-586
vec2mat function 2-589
vitdec function 2-591

W

weight, Hamming 2-324
wgn function 2-599