Communications Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 3

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

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

508-647-7000 Phone

508-647-7001 Fax

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

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

Communications Toolbox User's Guide

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)

Contents

1 Getting Started

2 Signal Sources

3 Performance Evaluation

4 BERTool: A Bit Error Rate Analysis GUI

Source Coding

5

6 Error-Control Coding

7 Interleaving

8 Modulation

9 Special Filters

10 Channels

11 Equalizers

12 Galois Field Computations

13 Galois Fields of Odd Characteristic

14 Functions — Categorical List

15 Functions — Alphabetical List

A Examples

Index

1

Getting Started

This chapter first provides a brief overview of the Communications Toolbox and then uses several examples to help you get started using the toolbox. This chapter assumes very little about your prior knowledge of MATLAB®, although it still assumes that you have a basic knowledge about communications subject matter.

What Is the Communications Toolbox?

The Communications Toolbox extends the MATLAB technical computing environment with functions, plots, and a graphical user interface for exploring, designing, analyzing, and simulating algorithms for the physical layer of communication systems. The toolbox helps you create algorithms for commercial and defense wireless or wireline systems.

The key features of the toolbox are

- Functions for designing the physical layer of communications links, including source coding, channel coding, interleaving, modulation, channel models, and equalization
- Plots such as eye diagrams and constellations for visualizing communications signals
- Graphical user interface for comparing the bit error rate of your system with a wide variety of proven analytical results
- Galois field data type for building communications algorithms

Expected Background

This guide assumes that you already have background knowledge in the subject of communications. If you do not yet have this background, then you can acquire it using a standard communications text or the books listed in one of this guide's sections titled "Selected Bibliography for... ."

For New Users

The discussion and examples in this chapter are aimed at new users. Continue reading this chapter and try out the examples. Then read those subsequent chapters that address the specific areas that concern you. When you find out which functions you want to use, refer to the online reference pages that describe those functions.

For Experienced Users

The online reference descriptions are probably the most relevant parts of this guide for you. Each reference description includes the function's syntax as well as a complete explanation of its options and operation. Many reference

descriptions also include examples, a description of the function's algorithm, and references to additional reading material.

You might also want to browse through nonreference parts of this documentation set, depending on your interests or needs.

Studying Components of a Communication System

The Communications Toolbox implements a variety of communications-related tasks. Many of the functions in the toolbox perform computations associated with a particular component of a communication system, such as a demodulator or equalizer. Other functions are designed for visualization or analysis.

While the later chapters of this document discuss various toolbox features in more depth, this section builds an example step by step to give you a first look at the toolbox. This section also shows how tools in the Communications Toolbox build upon the computational and visualization tools in the underlying MATLAB environment. The topics are as follows:

Modulating a Random Signal

This first example addresses the following problem:

Problem Process a binary data stream using a communication system that consists of a baseband modulator, channel, and demodulator. Compute the system's bit error rate (BER). Also, display the transmitted and received signals in a scatter plot.

The table below indicates the key tasks in solving the problem, along with relevant functions from the Communications Toolbox. The solution arbitrarily chooses baseband 16-QAM (quadrature amplitude modulation) as the modulation scheme and AWGN (additive white Gaussian noise) as the channel model.

Task	Function
Generate a random binary data stream	`randint`
Modulate using 16-QAM	`qammod`
Add white Gaussian noise	`awgn`
Create a scatter plot	`scatterplot`
Demodulate using 16-QAM	`qamdemod`
Compute the system's BER	`biterr`

Solution of Problem

The discussion below describes each step in more detail, introducing M-code along the way. To view all the code in one editor window, enter the following in the MATLAB Command Window.

```
edit commdoc_mod
```

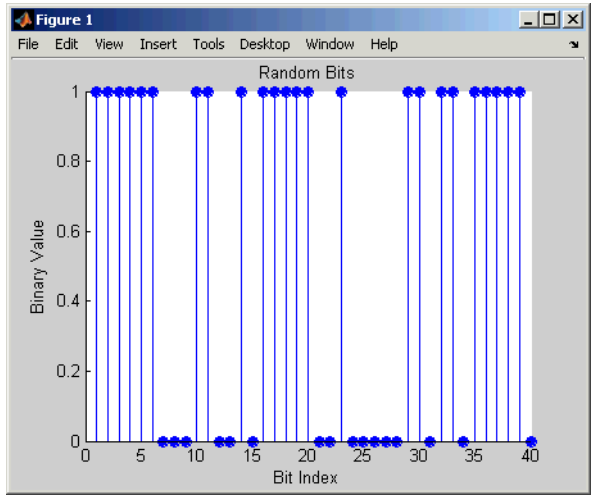
1. Generate a Random Binary Data Stream. The conventional format for representing a signal in MATLAB is a vector or matrix. This example uses the `randint` function to create a column vector that lists the successive values of a binary data stream. The length of the binary data stream (that is, the number of rows in the column vector) is arbitrarily set to 30,000.

Note The sampling times associated with the bits do not appear explicitly, and MATLAB has no inherent notion of time. For the purpose of this example, knowing only the values in the data stream is enough to solve the problem.

The code below also creates a stem plot of a portion of the data stream, showing the binary values. Your plot might look different because the example uses random numbers. Notice the use of the colon (`:`) operator in MATLAB to select a portion of the vector. For more information about this syntax, see "The Colon Operator" in the MATLAB documentation set.

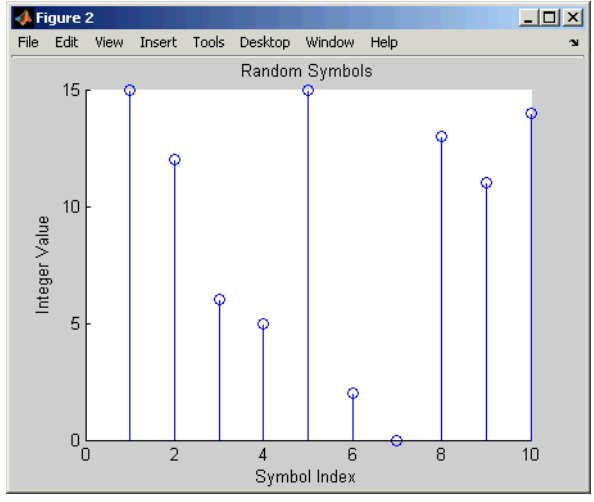
```
%% Setup
% Define parameters.
```

```
M = 16; % Size of signal constellation
k = log2(M); % Number of bits per symbol
n = 3e4; % Number of bits to process
nsamp = 1; % Oversampling rate

%% Signal Source
% Create a binary data stream as a column vector.
x = randint(n,1); % Random binary data stream

% Plot first 40 bits in a stem plot.
stem(x(1:40),'filled');
title('Random Bits');
xlabel('Bit Index'); ylabel('Binary Value');
```



2. Prepare to Modulate. The `qammod` function implements a 16-QAM modulator. However, it expects to receive integers between 0 and 15 rather than 4-tuples of bits. Therefore, you must preprocess the binary data stream `x` before invoking `qammod`. In particular, you arrange each 4-tuple of values from `x` across a row of a matrix, using the `reshape` function in MATLAB, and then apply the `bi2de` function to convert each 4-tuple to a corresponding integer. (The `.'` characters after the `reshape` command form the unconjugated array transpose operator in MATLAB. For more information about this and the similar `'` operator, see "Reshaping a Matrix" in the MATLAB documentation set.)

```
%% Bit-to-Symbol Mapping
% Convert the bits in x into k-bit symbols.
xsym = bi2de(reshape(x,k,length(x)/k).','left-msb');

%% Stem Plot of Symbols
% Plot first 10 symbols in a stem plot.
figure; % Create new figure window.
stem(xsym(1:10));
title('Random Symbols');
xlabel('Symbol Index'); ylabel('Integer Value');
```



3. Modulate Using 16-QAM. Having defined `xsym` as a column vector containing integers between 0 and 15, you can use `qammod` to modulate `xsym` using the baseband representation. Recall that `M` is 16, the alphabet size.

```
%% Modulation
% Modulate using 16-QAM.
y = qammod(xsym,M);
```

The result is a complex column vector whose values are in the 16-point QAM signal constellation. A later step in this example will show what the constellation looks like.

To learn more about modulation functions, see Chapter 8, "Modulation". Also, note that the `qammod` function does not apply any pulse shaping. To extend this example to use pulse shaping, see "Pulse Shaping Using a Raised Cosine Filter" on page 1-14. For an example that uses rectangular pulse shaping with PSK modulation, see `basicsimdemo`.

4. Add White Gaussian Noise. Applying the `awgn` function to the modulated signal adds white Gaussian noise to it. The ratio of bit energy to noise power spectral density, E_b/N_0, is arbitrarily set at 10 dB.

The expression to convert this value to the corresponding signal-to-noise ratio (SNR) involves `k`, the number of bits per symbol (which is 4 for 16-QAM), and `nsamp`, the oversampling factor (which is 1 in this example). The factor `k` is used to convert E_b/N_0 to an equivalent E_s/N_0, which is the ratio of *symbol* energy to noise power spectral density. The factor `nsamp` is used to convert E_s/N_0 in the symbol rate bandwidth to an SNR in the sampling bandwidth.

Note The definitions of `ytx` and `yrx` and the `nsamp` term in the definition of `snr` are not significant in this example so far, but will make it easier to extend the example later to use pulse shaping.

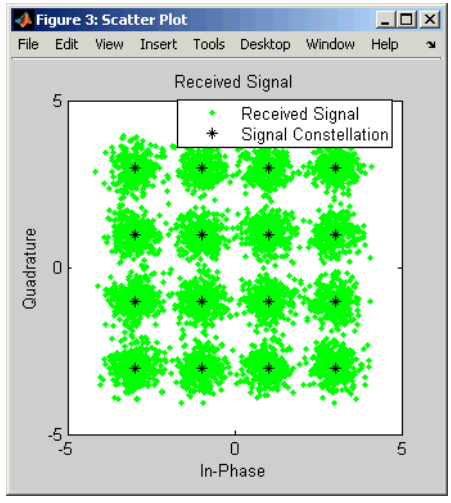
```
%% Transmitted Signal
ytx = y;

%% Channel
% Send signal over an AWGN channel.
```

```
EbNo = 10; % In dB
snr = EbNo + 10*log10(k) - 10*log10(nsamp);
ynoisy = awgn(ytx,snr,'measured');

%% Received Signal
yrx = ynoisy;
```

To learn more about `awgn` and other channel functions, see Chapter 10, "Channels".

5. Create a Scatter Plot. Applying the `scatterplot` function to the transmitted and received signals shows what the signal constellation looks like and how the noise distorts the signal. In the plot, the horizontal axis is the in-phase component of the signal and the vertical axis is the quadrature component. The code below also uses the `title`, `legend`, and `axis` functions in MATLAB to customize the plot.

```
%% Scatter Plot
% Create scatter plot of noisy signal and transmitted
% signal on the same axes.
h = scatterplot(yrx(1:nsamp*5e3),nsamp,0,'g.');
hold on;
scatterplot(ytx(1:5e3),1,0,'k*',h);
title('Received Signal');
legend('Received Signal','Signal Constellation');
axis([-5 5 -5 5]); % Set axis ranges.
hold off;
```



To learn more about `scatterplot`, see "Scatter Plots" on page 3-22.

6. Demodulate Using 16-QAM. Applying the `qamdemod` function to the received signal demodulates it. The result is a column vector containing integers between 0 and 15.

```
%% Demodulation
% Demodulate signal using 16-QAM.
zsym = qamdemod(yrx,M);
```

7. Convert the Integer-Valued Signal to a Binary Signal. The previous step produced `zsym`, a vector of integers. To obtain an equivalent binary signal, use the `de2bi` function to convert each integer to a corresponding binary 4-tuple along a row of a matrix. Then use the `reshape` function to arrange all the bits in a single column vector rather than a four-column matrix.

```
%% Symbol-to-Bit Mapping
% Undo the bit-to-symbol mapping performed earlier.
z = de2bi(zsym,'left-msb'); % Convert integers to bits.
% Convert z from a matrix to a vector.
z = reshape(z.',prod(size(z)),1);
```

8. Compute the System's BER. Applying the `biterr` function to the original binary vector and to the binary vector from the demodulation step above yields the number of bit errors and the bit error rate.

```
%% BER Computation
% Compare x and z to obtain the number of errors and
% the bit error rate.
[number_of_errors,bit_error_rate] = biterr(x,z)
```

The statistics appear in the MATLAB Command Window. Your results might vary because the example uses random numbers.

```
number_of_errors =

    71


bit_error_rate =

    0.0024
```

To learn more about `biterr`, see "Performance Results via Simulation" on page 3-2.

Plotting Signal Constellations

The example in the previous section created a scatter plot from the modulated signal. Although the plot showed the points in the QAM constellation, the plot did not indicate which integers between 0 and 15 the modulator mapped to a given constellation point. This section addresses the following problem:

Problem Plot a 16-QAM signal constellation with annotations that indicate the mapping from integers to constellation points.

The solution uses the `scatterplot` function to create the plot and the `text` function in MATLAB to create the annotations.

Solution of Problem

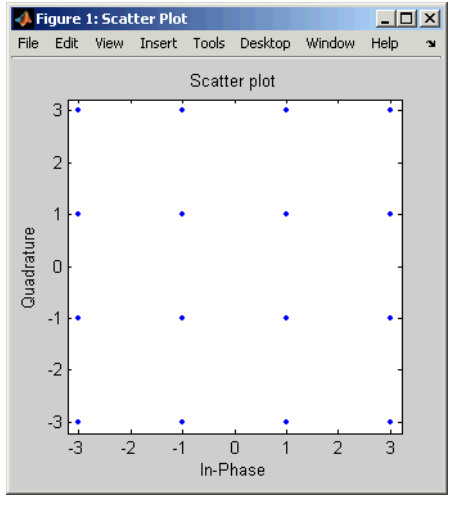
To view a completed M-file for this example, enter `edit commdoc_const` in the MATLAB Command Window.

1. Find All Points in the 16-QAM Signal Constellation. Applying the `qammod` function to a vector of integers between 0 and 15 results in an output vector containing all points in the 16-QAM signal constellation.

```
M = 16; % Number of points in constellation
intg = [0:M-1].'; % Vector of integers between 0 and M-1
pt = qammod(intg,M); % Vector of all points in constellation
```

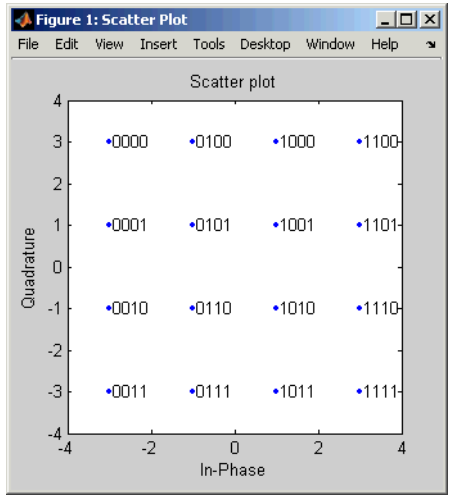
2. Plot the Signal Constellation. The `scatterplot` function plots the points in `pt`.

```
% Plot the constellation.
scatterplot(pt);
```



3. Annotate the Plot to Indicate the Mapping. To annotate the plot to show the relationship between `intg` and `pt`, use the `text` function to place a number in the plot beside each constellation point. The coordinates of the annotation are near the real and imaginary parts of the constellation point, but slightly offset to avoid overlap. The text of the annotation comes from the binary representation of `intg`. (The `dec2bin` function in MATLAB produces a string of digit characters, while the `de2bi` function used in the last section produces a vector of numbers.)

```
% Include text annotations that number the points.
text(real(pt)+0.1,imag(pt),dec2bin(intg));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.
```

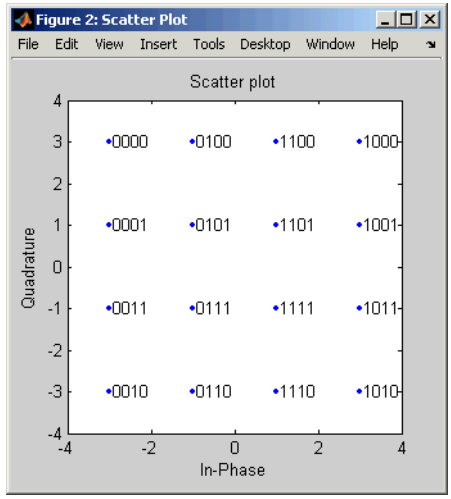


Binary-Coded 16-QAM Signal Constellation

Examining the Plot

In the plot above, notice that 0001 and 0010 correspond to adjacent constellation points on the left side of the diagram. Because these binary representations differ by two bits, the adjacency indicates that `qammod` did *not* use a Gray-coded signal constellation. (That is, if it were a Gray-coded signal constellation, then the annotations for each pair of adjacent points would differ by one bit.)

By contrast, the constellation below is one example of a Gray-coded 16-QAM signal constellation.



Gray-Coded 16-QAM Signal Constellation

The only difference, compared to the previous example, is that you pass in `'grey'` as the symbol order argument to the `qammod` function.

```
%% Modified Plot, With Gray Coding
M = 16; % Number of points in constellation
intg = [0:M-1].';
pt = qammod(intg,M,[],'gray'); % Vector of all points in constellation

scatterplot(pt); % Plot the constellation.

% Include text annotations that number the points.
text(real(pt)+0.1,imag(pt),dec2bin(intg));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.
```

Pulse Shaping Using a Raised Cosine Filter

This section further extends the example by addressing the following problem:

Problem Modify the Gray-coded modulation example so that it uses a pair of square root raised cosine filters to perform pulse shaping and matched filtering at the transmitter and receiver, respectively.

The solution uses the `rcosine` function to design the square root raised cosine filter and the `rcosflt` function to filter the signals. Alternatively, you can use the `rcosflt` function to perform both tasks in one command; see "Filtering with Raised Cosine Filters" on page 9-7 or the `rcosdemo` demonstration for more details.

Solution of Problem

This solution modifies the code from . To view the original code in an editor window, enter the following command in the MATLAB Command Window.

```
edit commdoc_gray
```

To view a completed M-file for this example, enter `edit commdoc_rrc` in the MATLAB Command Window.

1. Define Filter-Related Parameters. In the `Setup` section of the example from , replace the definition of the oversampling rate, `nsamp`, with the following.

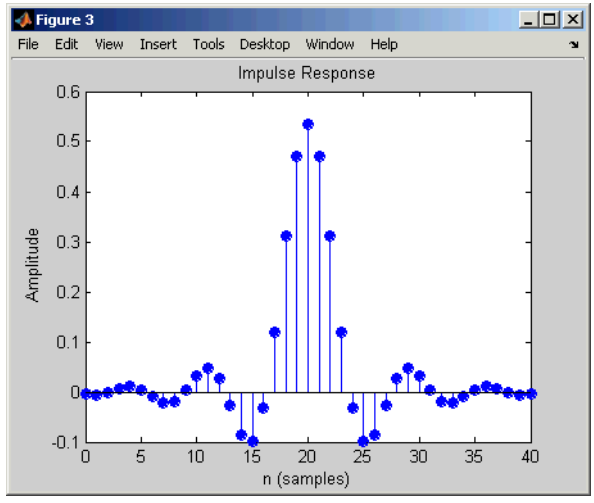
```
nsamp = 4; % Oversampling rate
```

Also, define other key parameters related to the filter by inserting the following after the `Modulation` section of the example and before the `Transmitted signal` section.

```
%% Filter Definition
% Define filter-related parameters.
filtorder = 40; % Filter order
delay = filtorder/(nsamp*2); % Group delay (# of input samples)
rolloff = 0.25; % Rolloff factor of filter
```

2. Create a Square Root Raised Cosine Filter. To design the filter and plot its impulse response, insert the following commands after the commands you added in the previous step.

```
% Create a square root raised cosine filter.
rrcfilter = rcosine(1,nsamp,'fir/sqrt',rolloff,delay);

% Plot impulse response.
figure; impz(rrcfilter,1);
```

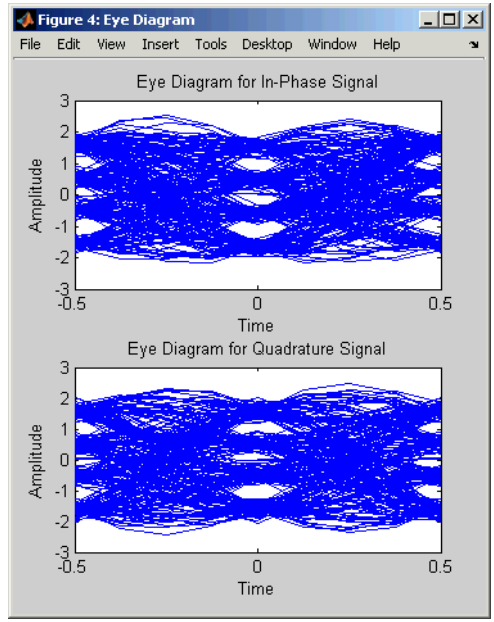


3. Filter the Modulated Signal. To filter the modulated signal, replace the `Transmitted Signal` section with following.

```
%% Transmitted Signal
% Upsample and apply square root raised cosine filter.
ytx = rcosflt(y,1,nsamp,'filter',rrcfilter);

% Create eye diagram for part of filtered signal.
eyediagram(ytx(1:2000),nsamp*2);
```

The `rcosflt` command internally upsamples the modulated signal, `y`, by a factor of `nsamp`, pads the upsampled signal with zeros at the end to flush the filter at the end of the filtering operation, and then applies the filter.

The `eyediagram` command creates an eye diagram for part of the filtered noiseless signal. This diagram illustrates the effect of the pulse shaping. Note that the signal shows significant intersymbol interference (ISI) because the filter is a square root raised cosine filter, not a full raised cosine filter.



To learn more about `eyediagram`, see “Eye Diagrams” on page 3-19.

4. Filter the Received Signal. To filter the received signal, replace the `Received Signal` section with the following.

```
%% Received Signal
% Filter received signal using square root raised cosine filter.
yrx = rcosflt(ynoisy,1,nsamp,'Fs/filter',rrcfilter);
yrx = downsample(yrx,nsamp); % Downsample.
yrx = yrx(2*delay+1:end-2*delay); % Account for delay.
```

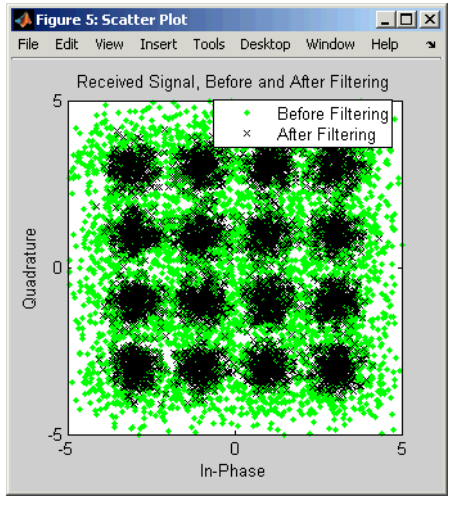
These commands apply the same square root raised cosine filter that the transmitter used earlier, and then downsample the result by a factor of `nsamp`.

The last command removes the first `2*delay` symbols and the last `2*delay` symbols in the downsampled signal because they represent the cumulative delay of the two filtering operations. Now `yrx`, which is the input to the demodulator, and `y`, which is the output from the modulator, have the same vector size. In the part of the example that computes the bit error rate, it is important to compare two vectors that have the same size.

5. Adjust the Scatter Plot. For variety in this example, make the scatter plot show the received signal before and after the filtering operation. To do this, replace the `Scatter Plot` section of the example with the following.

```
%% Scatter Plot
% Create scatter plot of received signal before and
% after filtering.
h = scatterplot(sqrt(nsamp)*ynoisy(1:nsamp*5e3),nsamp,0,'g.');
hold on;
scatterplot(yrx(1:5e3),1,0,'kx',h);
title('Received Signal, Before and After Filtering');
legend('Before Filtering','After Filtering');
axis([-5 5 -5 5]); % Set axis ranges.
```

Notice that the first `scatterplot` command scales `ynoisy` by `sqrt(nsamp)` when plotting. This is because the filtering operation changes the signal's power.



Using a Convolutional Code

This section further extends the example by addressing the following problem:

Problem Modify the previous example so that it includes convolutional coding and decoding, given the constraint lengths and generator polynomials of the convolutional code.

The solution uses the `convenc` and `vitdec` functions to perform encoding and decoding, respectively. It also uses the `poly2trellis` function to define a trellis that represents a convolutional encoder. To learn more about these functions, see "Convolutional Coding" on page 6-30.

See also `vitsimdemo` for an example of convolutional coding and decoding.

Solution of Problem

This solution modifies the code from "Pulse Shaping Using a Raised Cosine Filter" on page 1-14. To view the original code in an editor window, enter the following command in the MATLAB Command Window.

```
edit commdoc_rrc
```

To view a completed M-file for this example, enter `edit commdoc_code` in the MATLAB Command Window.

1. Increase the Number of Symbols. Convolutional coding at this value of `EbNo` reduces the BER markedly. As a result, accumulating enough errors to compute a reliable BER requires you to process more symbols. In the `Setup` section, replace the definition of the number of bits, `n`, with the following.

```
n = 5e5; % Number of bits to process
```

Note The larger number of bits in this example causes it to take a noticeably longer time to run compared to the examples in previous sections.

2. Encode the Binary Data. To encode the binary data before mapping it to integers for modulation, insert the following after the `Signal Source` section of the example and before the `Bit-to-Symbol Mapping` section.

```
%% Encoder
% Define a convolutional coding trellis and use it
% to encode the binary data.
t = poly2trellis([5 4],[23 35 0; 0 5 13]); % Trellis
code = convenc(x,t); % Encode.
coderate = 2/3;
```

The `poly2trellis` command defines the trellis that represents the convolutional code that `convenc` uses for encoding the binary vector, `x`. The two input arguments in the `poly2trellis` command indicate the constraint length and generator polynomials, respectively, of the code. A diagram showing this encoder is in "Example: A Rate-2/3 Feedforward Encoder" on page 6-40.

3. Apply the Bit-to-Symbol Mapping to the Encoded Signal. The bit-to-symbol mapping must apply to the encoded signal, `code`, not the original uncoded data. Replace the first definition of `xsym` (within the `Bit-to-Symbol Mapping` section) with the following.

```
% B. Do ordinary binary-to-decimal mapping.
xsym = bi2de(reshape(code,k,length(code)/k).','left-msb');
```

Recall that `k` is 4, the number of bits per symbol in 16-QAM.

4. Account for Code Rate When Defining SNR. Converting from E_b/N_0 to the signal-to-noise ratio requires you to account for the number of information bits per symbol. Previously, each symbol corresponded to `k` bits. Now, each symbol corresponds to `k*coderate` information bits. More concretely, three symbols correspond to 12 coded bits in 16-QAM, which correspond to 8 uncoded (information) bits, so the ratio of symbols to information bits is `8/3 = 4*(2/3) = k*coderate`.

Therefore, change the definition of `snr` (within the `Channel` section) to the following.

```
snr = EbNo + 10*log10(k*coderate)-10*log10(nsamp);
```

5. Decode the Convolutional Code. To decode the convolutional code before computing the error rate, insert the following after the entire `Symbol-to-Bit Mapping` section and just before the `BER Computation` section.

```
%% Decoder
% Decode the convolutional code.
tb = 16; % Traceback length for decoding
z = vitdec(z,t,tb,'cont','hard'); % Decode.
```

The syntax for the `vitdec` function instructs it to use hard decisions. The `'cont'` argument instructs it to use a mode designed for maintaining continuity when you invoke the function repeatedly (as in a loop). Although this example does not use a loop, the `'cont'` mode is used for the purpose of illustrating how to compensate for the delay in this decoding operation. The delay is discussed further in "More About Delays" on page 1-21.

6. Account for Delay When Computing BER. The continuous operation mode of the Viterbi decoder incurs a delay whose duration in bits equals the traceback length, `tb`, times the number of input streams to the *encoder*. For this rate 2/3 code, the encoder has two input streams, so the delay is `2*tb` bits.

As a result, the first `2*tb` bits in the decoded vector, `z`, are just zeros. When computing the bit error rate, you should ignore the first `2*tb` bits in `z` and the last `2*tb` bits in the original vector, `x`. If you do not compensate for the delay, then the BER computation is meaningless because it compares two vectors that do not truly correspond to each other.

Therefore, replace the `BER Computation` section with the following.

```
%% BER Computation
% Compare x and z to obtain the number of errors and
% the bit error rate. Take the decoding delay into account.
decdelay = 2*tb; % Decoder delay, in bits
[number_of_errors,bit_error_rate] = ...
   biterr(x(1:end-decdelay),z(decdelay+1:end))
```

More About Delays

The decoding operation in this example incurs a delay, which means that the output of the decoder lags the input. Timing information does not appear explicitly in the example, and the duration of the delay depends on the specific operations being performed. Delays occur in various communications-related operations, including convolutional decoding,

convolutional interleaving/deinterleaving, equalization, and filtering. To find out the duration of the delay caused by specific functions or operations, refer to the specific documentation for those functions or operations. For example:

- The `vitdec` reference page
- "Delays of Convolutional Interleavers" on page 7-9
- "Delays from Equalization" on page 11-21
- "Example: Compensating for Group Delays When Analyzing Data" on page 9-3
- "Fading Channels" on page 10-6

The "Effect of Delays on Recovery of Convolutionally Interleaved Data" on page 7-10 discussion also includes two typical ways to compensate for delays.

Simulating a Communication System

The examples so far have performed tasks associated with various components of a communication system. In some cases, you might need to create a more sophisticated simulation that uses one or more of these techniques:

- Looping over a set of values of a specific parameter, such as E_b/N_0, the alphabet size, or the oversampling rate, so you can see the parameter's effect on the system
- Processing data in multiple smaller sets rather than in one large set, to reduce the memory requirement
- Dynamically determining how much data to process to get reliable results, instead of trying to guess at the beginning

This section discusses these issues and provides examples of constructs that you can use in your simulations of communication systems. The topics are as follows:

Using BERTool to Run Simulations

The Communications Toolbox includes a graphical user interface (GUI) called BERTool that is designed to solve problems like the following:

Problem Modify the modulation example in so that it computes the BER for integer values of `EbNo` between 0 and 7. Plot the BER as a function of `EbNo` using a logarithmic scale for the vertical axis.

BERTool solves the problem by managing a series of simulations with different values of E_b/N_0, collecting the results, and creating a plot. You provide the core of the simulation, which in this case is a minor modification of the example in .

This section introduces BERTool as well as some simulation-related issues, in these topics:

However, this section is not a comprehensive description of BERTool; for more information about BERTool, see Chapter 4, "BERTool: A Bit Error Rate Analysis GUI".

Solution of Problem

This solution uses code from as well as code from a template file that is tailored for use with BERTool. To view the original code in an editor window, enter these commands in the MATLAB Command Window.

```
edit commdoc_gray
edit bertooltemplate
```

To view a completed M-file for this example, enter `edit commdoc_bertool` in the MATLAB Command Window.

1. Save Template in Your Own Directory. Navigate to a directory where you want to save your own files. Save the BERTool template (`bertooltemplate`) under the filename `my_commdoc_bertool` to avoid overwriting the original template.

Also, change the first line of `my_commdoc_bertool`, which is the function declaration, to use the new filename.

```
function [ber, numBits] = my_commdoc_bertool(EbNo, maxNumErrs, maxNumBits)
```

2. Copy Setup Code Into Template. In the `my_commdoc_bertool` file, replace

```
% --- Set up parameters. ---
% --- INSERT YOUR CODE HERE.
```

with the following setup code adapted from the example in .

```
% Setup
% Define parameters.
```

```
M = 16; % Size of signal constellation
k = log2(M); % Number of bits per symbol
n = 1000; % Number of bits to process
nsamp = 1; % Oversampling rate
```

To save time in the simulation, the code above changes the value of `n` from its original value. At small values of `EbNo`, it is not necessary to process tens of thousands of symbols to compute an accurate BER; at large values of `EbNo`, the loop structure in the template file (described later) causes the simulation to include at least 100 errors even if it must iterate several times through the loop to accumulate that many errors.

3. Copy Simulation Code Into Template. In the `my_commdoc_bertool` file, replace

```
% --- Proceed with simulation.
% --- Be sure to update totErr and numBits.
% --- INSERT YOUR CODE HERE.
```

with the rest of the code (that is, the code following the `Setup` section) from the example in .

Also, type a semicolon at the end of the last line of the pasted code (the `biterr` command) to suppress screen output when BERTool runs the simulation.

6. Update numBits and totErr. After the pasted code from the last step and before the `end` statement from the template, insert the following code.

```
%% Update totErr and numBits.
totErr = totErr + number_of_errors;
numBits = numBits + n;
```

These commands enable the function to keep track of the number of bits processed and the number of errors detected.

5. Suppress Earlier Plots. Running multiple iterations would result in a large number of plots, which this example suppresses for simplicity. In the `my_commdoc_bertool` file, remove the lines of code that use these functions: `stem`, `title`, `xlabel`, `ylabel`, `figure`, `scatterplot`, `hold`, `legend`, and `axis`.

6. Omit Direct Assignment of EbNo. When BERTool invokes a simulation function, it specifies a value of `EbNo`. The `my_commdoc_bertool` function must not directly assign `EbNo`. Therefore, remove or comment out the line that you pasted into `my_commdoc_bertool` (within the `Channel` section) that assigns `EbNo` directly.

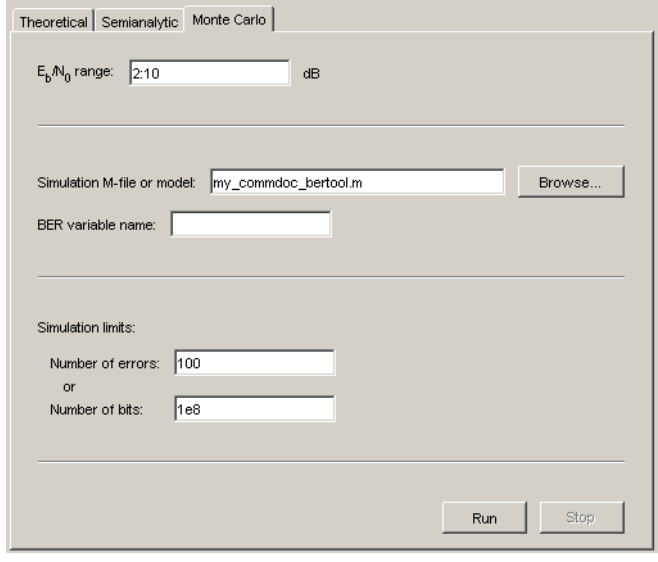
```
% EbNo = 10; % In dB % COMMENT OUT FOR BERTOOL
```

7. Save Simulation Function. The simulation function, `my_commdoc_bertool`, is complete. Save the file so that BERTool can use it.

8. Open BERTool and Enter Parameters. To open BERTool, enter

```
bertool
```

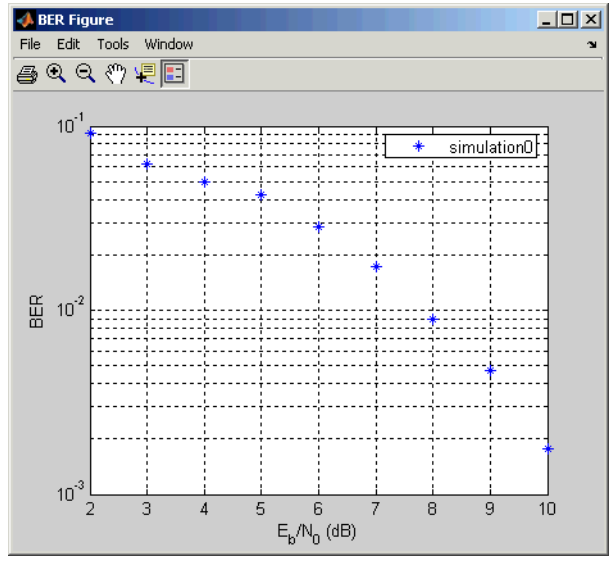
in the MATLAB Command Window. Then click the **Monte Carlo** tab and enter parameters as shown below.



These parameters tell BERTool to run your simulation function, `my_commdoc_bertool`, for each value of `EbNo` in the vector `2:10` (that is, the vector `[2 3 4 5 6 7 8 9 10]`). Each time the simulation runs, it continues

processing data until it detects 100 bit errors or processes a total of 1e8 bits, whichever occurs first.

9. Use BERTool to Simulate and Plot. Click the **Run** button on BERTool. BERTool begins the series of simulations and eventually reports the results to you in a plot like the one below.

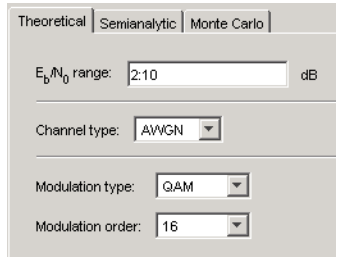


To compare these BER results with theoretical results, leave BERTool open and use the procedure below.

Comparing with Theoretical Results

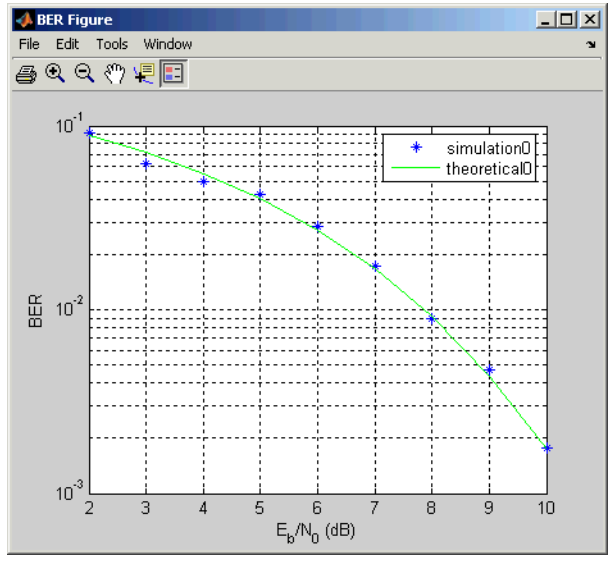
To check whether the results from the solution above are correct, use BERTool again. This time, use its **Theoretical** panel to plot theoretical BER results in the same window as the simulation results from before. Follow this procedure:

1 In the BERTool GUI, click the **Theoretical** tab and enter parameters as shown below.



The parameters tell BERTool to compute theoretical BER results for 16-QAM over an AWGN channel, for E_b/N_0 values in the vector `2:10`.

2 Click the **Plot** button. The resulting plot shows a solid curve for the theoretical BER results and plotting markers for the earlier simulation results.



Notice that the plotting markers are close to the theoretical curve. It is relevant that the simulation code used a Gray-coded signal constellation, unlike the first modulation example of this chapter (in “Modulating a Random Signal” on page 1-4). The theoretical performance results assume a Gray-coded signal constellation.

To continue exploring BERTool, you can select the **Fit** check box to fit a curve to the simulation data, or set **Confidence Level** to a numerical value to include confidence intervals in the plot. See also Chapter 4, "BERTool: A Bit Error Rate Analysis GUI" for more about BERTool.

More About the Simulation Structure

Looking more closely at the simulation function in this example, you might make a few observations about its structure, and particularly about the loop marked with the comments

```
% Simulate until number of errors exceeds maxNumErrs
% or number of bits processed exceeds maxNumBits.
```

The loop structure means that the simulation processes some data, accumulates bit errors, and then decides whether to repeat the process with another set of data. The advantage of this approach is that you do not have to guess in advance how much data you need to process to obtain an accurate BER estimate. This is very useful when your series of simulations spans a large E_b/N_0 range because simulations at higher values of E_b/N_0 require more data processing to maintain the same level of accuracy in the BER estimate. Another advantage of this approach is that you avoid memory problems caused by excessively large data sets.

However, a potential complication from dividing large data sets into a series of smaller data sets that you process in a loop is that you might need to take steps to ensure the continuity of computations from one iteration to the next. For example, continuity is important when the simulation includes convolutional decoding, convolutional interleaving/deinterleaving, continuous phase modulation, fading channels, and equalization. To learn more about how to maintain continuity, see the examples in

- The `vitdec` reference page
- The `viterbisim` demonstration function (designed to be used with BERTool)
- The `muxdeintrlv` reference page
- The `mskdemod` reference page
- "Fading Channels" on page 10-6

- "Equalizing Using a Loop" on page 11-22
- "Equalizing in Continuous Operation Mode" on page 11-29

If you divide your data set into a series of very small data sets, then the large number of function calls might make the simulation slow. You can use the Profiler tool in MATLAB to help you make your code faster.

Varying Parameters and Managing a Set of Simulations

A common task in analyzing a communication system is to vary a parameter, possibly a parameter other than E_b/N_0, and find out how the system responds. This section addresses the following problem:

Problem Modify the modulation example in "Modulating a Random Signal" on page 1-4 so that it computes the BER for alphabet sizes (`M`) of 4, 8, 16, and 32 and for integer values of `EbNo` between 0 and 7. For each value of `M`, plot the BER as a function of `EbNo` using a logarithmic scale for the vertical axis.

The earlier section ("Modulating a Random Signal" on page 1-4) presented a model of the system that computes the BER for specific values of `M` and `EbNo`. Therefore, the only remaining task is to vary `M` and `EbNo` and collect multiple error rates. For simplicity, this solution uses the same number of bits for each value of `M` and `EbNo`, unlike the example in "Using BERTool to Run Simulations" on page 1-23.

Solution of Problem

This solution modifies the code from "Modulating a Random Signal" on page 1-4 by introducing and exploiting a nested loop structure. To view the original code in an editor window, enter the following command in the MATLAB Command Window.

```
edit commdoc_mod
```

To view a completed M-file for this example, enter `edit commdoc_mcurves` in the MATLAB Command Window.

1. Define the Set of Values for the Parameter. At the beginning of the script, introduce variables that list all the values of `M` and `EbNo` that the problem requires. Also, preallocate space for error statistics corresponding to each combination of `M` and `EbNo`.

```
%% Ranges of Variables
Mvec = [4 8 16 32]; % Values of M to consider
EbNovec = [0:7]; % Values of EbNo to consider

%% Preallocate space for results.
number_of_errors = zeros(length(Mvec),length(EbNovec));
bit_error_rate = zeros(length(Mvec),length(EbNovec));
```

2. Introduce a Loop Structure. After `Mvec` and `EbNovec` are defined and space is preallocated for statistics, all the subsequent commands can go inside a loop, as illustrated below.

```
%% Simulation loops
for idxM = 1:length(Mvec)
   for idxEbNo = 1:length(EbNovec)

      % OTHER COMMANDS

   end % End of loop over EbNo values
end % End of loop over M values
```

3. Inside the Loop, Parameterize as Appropriate. The M-code from specifies fixed values of `M` and `EbNo`, while this problem requires using a different value for each iteration of the loop. Therefore, change the definitions of `M` (within the `Setup` section) and `EbNo` (within the `Channel` section) as follows.

```
M = Mvec(idxM); % Size of signal constellation

EbNo = EbNovec(idxEbNo); % In dB
```

Also, the original M-code returns scalar values for the BER and number of errors, while it makes sense in this case to save the whole array of error statistics instead of overwriting the variables in each iteration. Therefore, replace the `BER Computation` section with the following.

```
%% BER Computation
% Compare x and z to obtain the number of errors and
% the bit error rate.
[number_of_errors(idxM,idxEbNo),bit_error_rate(idxM,idxEbNo)] = ...
   biterr(x,z);
```

Note An earlier step preallocated space for the matrices `number_of_errors` and `bit_error_rate`. While not strictly necessary, this is a better MATLAB programming habit than expanding the matrices' size in each iteration. To learn more, see "Preallocating Arrays" in the MATLAB documentation set.

---

4. Suppress Earlier Plots. Running multiple iterations would result in a large number of plots, which this example suppresses for simplicity. Remove the lines of code that use these functions:`stem`, `title`, `xlabel`, `ylabel`, `figure`, `scatterplot`, `hold`, `legend`, and `axis`.

5. Create BER Plot. The `semilogy` function in MATLAB creates a plot with a logarithmic scale in the vertical axis. The following commands, placed just before the end of the loop over `M` values, create the desired BER plot curve by curve during the simulation.

```
%% Plot a Curve.
markerchoice = '.xo*';
plotsym = [markerchoice(idxM) '-']; % Plotting style for this curve
semilogy(EbNovec,bit_error_rate(idxM,:),plotsym); % Plot one curve.
drawnow; % Update the plot instead of waiting until the end.
hold on; % Make sure next iteration does not remove this curve.
```

You might also want to customize the plot at the end by adding this code after the end of both loops.

```
%% Complete the plot.
title('Performance of M-QAM for Varying M');
xlabel('EbNo (dB)'); ylabel('BER');
legend('M = 4','M = 8','M = 16','M = 32',...
   'Location','SouthWest');
```

6. Run the Entire Script. The script creates a plot like the one below.

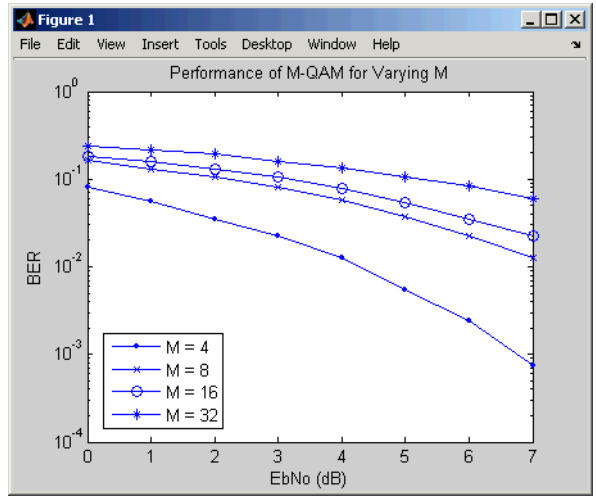


Figure 1
File Edit View Insert Tools Desktop Window Help
Performance of M-QAM for Varying M
BER
EbNo (dB)
M = 4
M = 8
M = 16
M = 32

Learning More

You can learn more about the Communications Toolbox from the following sources.

Online Help

To find online documentation, select **Full Product Family Help** from the **Help** menu in the MATLAB desktop. This launches the Help browser. For a more detailed explanation of any of the topics covered in this chapter, see the documentation listed under Communications Toolbox in the left pane of the Help browser.

Besides this chapter, the online documentation set contains these components:

- A chapter about each of the core areas of functionality of the toolbox (such as error-control coding, modulation, and equalizers)
- A reference page for each function in the toolbox, indexed alphabetically and by category

You can also use the online index of examples to find code examples that are relevant for the tasks you want to do.

Demos

To see more Communications Toolbox examples, select **Demos** from the **Help** menu in the MATLAB desktop. This opens the Help browser to the demonstration area. Double-click **Toolboxes** and then select **Communications** to list the available demos.

The MathWorks Online

To read the documentation for the Communications Toolbox on the MathWorks Web site, point your Web browser to

```
http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml
```

Other resources for the Communications Toolbox are available at

```
http://www.mathworks.com/products/communications/
```

2

Signal Sources

Every communication system has one or more signal sources. This chapter describes how to use the Communications Toolbox to generate random signals, which are useful for simulating noise, errors, or signal sources. The sections are as follows.

For more general random number generators, see the online reference pages for the built-in MATLAB functions `rand` and `randn`.

White Gaussian Noise

The `wgn` function generates random matrices using a white Gaussian noise distribution. You specify the power of the noise in either dBW (decibels relative to a watt), dBm, or linear units. You can generate either real or complex noise.

For example, the command below generates a column vector of length 50 containing real white Gaussian noise whose power is 2 dBW. The function assumes that the load impedance is 1 ohm.

```
y1 = wgn(50,1,2);
```

To generate complex white Gaussian noise whose power is 2 Watts, across a load of 60 ohms, use either of the commands below. Notice that the ordering of the string inputs does not matter.

```
y2 = wgn(50,1,2,60,'complex','linear');
y3 = wgn(50,1,2,60,'linear','complex');
```

To send a signal through an additive white Gaussian noise channel, use the `awgn` function. See “AWGN Channel” on page 10-3 for more information.

Random Symbols

The `randsrc` function generates random matrices whose entries are chosen independently from an alphabet that you specify, with a distribution that you specify. A special case generates bipolar matrices.

For example, the command below generates a 5-by-4 matrix whose entries are independently chosen and uniformly distributed in the set {1,3,5}. (Your results might vary because these are random numbers.)

```
a = randsrc(5,4,[1,3,5])

a =

     3     5     1     5
     1     5     3     3
     1     3     3     1
     1     1     3     5
     3     1     1     3
```

If you want 1 to be twice as likely to occur as either 3 or 5, then use the command below to prescribe the skewed distribution. Notice that the third input argument has two rows, one of which indicates the possible values of `b` and the other indicates the probability of each value.

```
b = randsrc(5,4,[1,3,5; .5,.25,.25])

b =

     3     3     5     1
     1     1     1     1
     1     5     1     1
     1     3     1     3
     3     1     3     1
```

Random Integers

The `randint` function generates random integer matrices whose entries are in a range that you specify. A special case generates random binary matrices.

For example, the command below generates a 5-by-4 matrix containing random integers between 2 and 10.

```
c = randint(5,4,[2,10])

c =

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

If your desired range is [0,10] instead of [2,10] then you can use either of the commands below. They produce different numerical results, but use the same distribution.

```
d = randint(5,4,[0,10]);
e = randint(5,4,11);
```

Random Bit Error Patterns

The `randerr` function generates matrices whose entries are either 0 or 1. However, its options are rather different from those of `randint`, because `randerr` is meant for testing error-control coding. For example, the command below generates a 5-by-4 binary matrix having the property that each row contains exactly one 1.

```
f = randerr(5,4)

f =

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

You might use such a command to perturb a binary code that consists of five four-bit codewords. Adding the random matrix `f` to your code matrix (modulo 2) would introduce exactly one error into each codeword.

On the other hand, if you want to perturb each codeword by introducing one error with probability 0.4 and two errors with probability 0.6, then the command below should replace the one above.

```
% Each row has one '1' with probability 0.4, otherwise two '1's
g = randerr(5,4,[1,2; 0.4,0.6])

g =

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

Note The probability matrix that is the third argument of `randerr` affects only the *number* of 1s in each row, not their placement.

As another application, you can generate an equiprobable binary 100-element column vector using any of the commands below. The three commands produce different numerical outputs, but use the same *distribution*. Notice that the third input arguments vary according to each function's particular way of specifying its behavior.

```
binarymatrix1 = randsrc(100,1,[0 1]); % Possible values are 0,1.
binarymatrix2 = randint(100,1,2); % Two possible values
binarymatrix3 = randerr(100,1,[0 1;.5 .5]); % No 1s, or one 1
```

3

Performance Evaluation

Simulating a communication system often involves analyzing its response to the noise inherent in real-world components, studying its behavior using graphical means, and determining whether the resulting performance meets standards of acceptability. The sections in this chapter are as follows.

Because error analysis is often a component of communication system simulation, other portions of this guide provide additional examples.

Performance Results via Simulation

One way to compute the bit error rate or symbol error rate for a communication system is to simulate the transmission of data messages and compare all messages before and after transmission. The simulation of the communication system components using functions in the Communications Toolbox is covered in other parts of this guide. This section describes how to perform the comparison of the data messages that enter and leave the simulation. An additional example of computing performance results via simulation is in "Curve Fitting for Error Rate Plots" on page 3-13 in the discussion of curve fitting.

Using Simulated Data to Compute Bit and Symbol Error Rates

The `biterr` function compares two sets of data and computes the number of bit errors and the bit error rate. The `symerr` function compares two sets of data and computes the number of symbol errors and the symbol error rate. An error is a discrepancy between corresponding points in the two sets of data.

Of the two sets of data, typically one represents messages entering a transmitter and the other represents recovered messages leaving a receiver. You might also compare data entering and leaving other parts of your communication system: for example, data entering an encoder and data leaving a decoder.

If your communication system uses several bits to represent one symbol, then counting bit errors is different from counting symbol errors. In either the bit- or symbol-counting case, the error rate is the number of errors divided by the total number (of bits or symbols) transmitted.

Note To ensure an accurate error rate, you should typically simulate enough data to produce at least 100 errors.

If the error rate is very small (for example, 10^{-6} or smaller), then the semianalytic technique might compute the result more quickly than a

simulation-only approach. See “Performance Results via the Semianalytic Technique” on page 3-5 for more information on how to use this technique.

Example: Computing Error Rates

The script below uses the `symerr` function to compute the symbol error rates for a noisy linear block code. After artificially adding noise to the encoded message, it compares the resulting noisy code to the original code. Then it decodes and compares the decoded message to the original one.

```
m = 3; n = 2^m-1; k = n-m; % Prepare to use Hamming code.
msg = randint(k*200,1,2); % 200 messages of k bits each
code = encode(msg,n,k,'hamming');
codenoisy = rem(code+(rand(n*200,1)>.95),2); % Add noise.
% Decode and correct some errors.
newmsg = decode(codenoisy,n,k,'hamming');
% Compute and display symbol error rates.
[codenum,coderate] = symerr(code,codenoisy);
[msgnum,msgrate] = symerr(msg,newmsg);
disp(['Error rate in the received code: ',num2str(coderate)])
disp(['Error rate after decoding: ',num2str(msgrate)])
```

The output is below. The error rate decreases after decoding because the Hamming decoder corrects some of the errors. Your results might vary because the example uses random numbers.

```
Error rate in the received code: 0.054286
Error rate after decoding: 0.03
```

Comparison of Symbol Error Rate and Bit Error Rate

In the example above, the symbol errors and bit errors are the same because each symbol is a bit. The commands below illustrate the difference between symbol errors and bit errors in other situations.

```
a = [1 2 3]'; b = [1 4 4]';
format rat % Display fractions instead of decimals.
[snum,srate] = symerr(a,b)
[bnum,brate] = biterr(a,b)
```

The output is below.

```
snum =

      2


srate =

     2/3


bnum =

      5


brate =

     5/9
```

`bnum` is 5 because the second entries differ in two bits and the third entries differ in three bits. `brate` is 5/9 because the total number of bits is nine. The total number of bits is, by definition, the number of entries in `a` or `b` times the maximum number of bits among all entries of `a` and `b`.

Performance Results via the Semianalytic Technique

The technique described in "Performance Results via Simulation" on page 3-2 works well for a large variety of communication systems, but can be prohibitively time-consuming if the system's error rate is very small (for example, 10^{-6} or smaller). This section describes how to use the semianalytic technique as an alternative way to compute error rates. For certain types of systems, the semianalytic technique can produce results much more quickly than a nonanalytic method that uses only simulated data.

The semianalytic technique uses a combination of simulation and analysis to determine the error rate of a communication system. The `semianalytic` function in the Communications Toolbox helps you implement the semianalytic technique by performing some of the analysis.

The topics in this section are

For more background information on the semianalytic technique, refer to [3] .

When to Use the Semianalytic Technique

The semianalytic technique works well for certain types of communication systems, but not for others. The semianalytic technique is applicable if a system has all of these characteristics:

- Any effects of multipath fading, quantization, and amplifier nonlinearities must *precede* the effects of noise in the actual channel being modeled.
- The receiver is perfectly synchronized with the carrier, and timing jitter is negligible. Because phase noise and timing jitter are slow processes, they reduce the applicability of the semianalytic technique to a communication system.
- The noiseless simulation has no errors in the received signal constellation. Distortions from sources other than noise should be mild enough to keep each signal point in its correct decision region. If this is not the case, then

the calculated BER will be too low. For instance, if the modeled system has a phase rotation that places the received signal points outside their proper decision regions, then the semianalytic technique is not suitable to predict system performance.

Furthermore, the `semianalytic` function assumes that the noise in the actual channel being modeled is Gaussian. For details on how to adapt the semianalytic technique for non-Gaussian noise, see the discussion of generalized exponential distributions in [3].

Procedure for the Semianalytic Technique

The procedure below describes how you would typically implement the semianalytic technique using the `semianalytic` function:

1. Generate a message signal containing *at least* M^L symbols, where M is the alphabet size of the modulation and L is the length of the impulse response of the channel, in symbols. 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.

2. Modulate a carrier with the message signal using baseband modulation. Supported modulation types are listed on the reference page for `semianalytic`.

3. Filter the modulated signal with a transmit filter. This filter is often a square-root raised cosine filter, but you can also use a Butterworth, Bessel, Chebyshev type 1 or 2, elliptic, or more general FIR or IIR filter. Store the result of this step as `txsig` for later use.

4. Run the filtered signal through a *noiseless* channel. This channel can include multipath fading effects, phase shifts, amplifier nonlinearities, quantization, and additional filtering, but it must not include noise. Store the result of this step as `rxsig` for later use.

5. Invoke the `semianalytic` function using the `txsig` and `rxsig` data from earlier steps. Specify a receive filter as a pair of input arguments, unless you want to use the function's default filter. The function filters `rxsig` and then determines the error probability of each received signal point by analytically applying the Gaussian noise distribution to each point. The

function averages the error probabilities over the entire received signal to determine the overall error probability. If the error probability calculated in this way is a symbol error probability, then the function converts it to a bit error rate, typically by assuming Gray coding. The function returns the bit error rate (or, in the case of DQPSK modulation, an upper bound on the bit error rate).

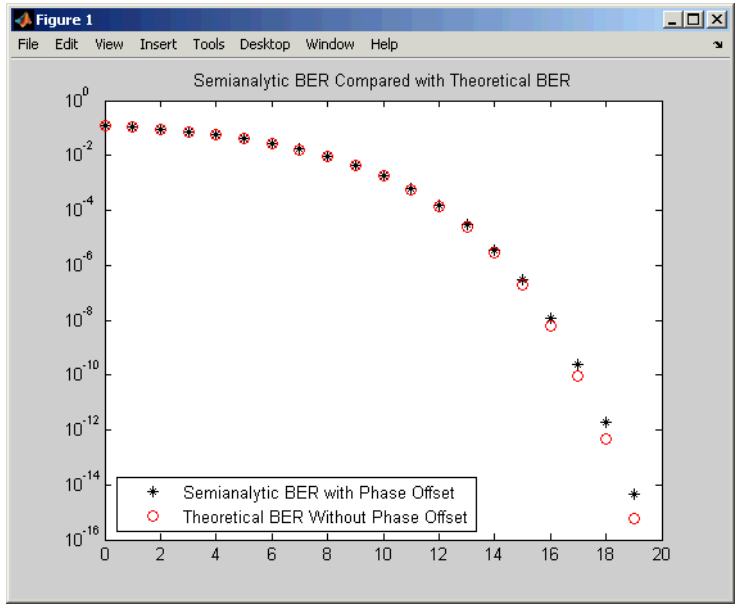
Example: Using the Semianalytic Technique

The example below illustrates the procedure described above, using 16-QAM modulation. It also compares the error rates obtained from the semianalytic technique with the theoretical error rates obtained from published formulas and computed using the `berawgn` function. The resulting plot shows that the error rates obtained using the two methods are nearly identical. The discrepancies between the theoretical and computed error rates are largely due to the phase offset in this example's channel model.

```
% Step 1. Generate message signal of length >= M^L.
M = 16; % Alphabet size of modulation
L = 1; % Length of impulse response of channel
msg = [0:M-1 0]; % M-ary message sequence of length > M^L

% Step 2. Modulate the message signal using baseband modulation.
modsig = qammod(msg,M); % Use 16-QAM.
Nsamp = 16;
modsig = rectpulse(modsig,Nsamp); % Use rectangular pulse shaping.

% Step 3. Apply a transmit filter.
txsig = modsig; % No filter in this example

% Step 4. Run txsig through a noiseless channel.
rxsig = txsig*exp(j*pi/180); % Static phase offset of 1 degree
% Step 5. Use the semianalytic function.
% Specify the receive filter as a pair of input arguments.
% In this case, num and den describe an ideal integrator.
num = ones(Nsamp,1)/Nsamp;
den = 1;
EbNo = [0:20]; % Range of Eb/No values under study
ber = semianalytic(txsig,rxsig,'qam',M,Nsamp,num,den,EbNo);

% For comparison, calculate theoretical BER.
```

```
bertheory = berawgn(EbNo,'qam',M);

% Plot computed BER and theoretical BER.
figure; semilogy(EbNo,ber,'k*');
hold on; semilogy(EbNo,bertheory,'ro');
title('Semianalytic BER Compared with Theoretical BER');
legend('Semianalytic BER with Phase Offset',...
   'Theoretical BER Without Phase Offset','Location','SouthWest');
hold off;
```

The example creates a figure like the one below.



Theoretical Performance Results

While the `biterr` function discussed above can help you gather empirical error statistics, you might also want to compare those results with theoretical error statistics. Certain types of communication systems are associated with closed-form expressions for the bit error rate or a bound on it. The functions listed in the table below compute the closed-form expressions for some types of communication systems, where such expressions exist.

Type of Communication System	Function
Uncoded AWGN channel	`berawgn`
Coded AWGN channel	`bercoding`
Uncoded Rayleigh fading channel	`berfading`
Uncoded AWGN channel with imperfect synchronization	`bersync`

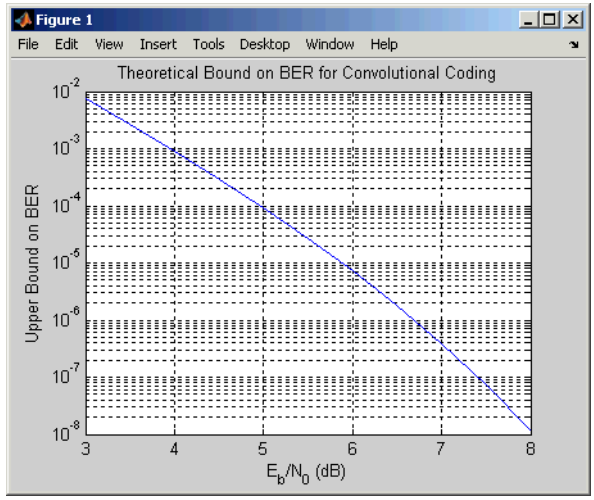
Each function's reference page lists one or more books containing the closed-form expressions that the function implements.

Plotting Theoretical Error Rates

The example below uses the `bercoding` function to compute upper bounds on bit error rates for convolutional coding with a soft-decision decoder. The data used for the generator and distance spectrum are from [5] and [2], respectively.

```
coderate = 1/4; % Code rate
% Create a structure dspec with information about distance spectrum.
dspec.dfree = 10; % Minimum free distance of code
dspec.weight = [1 0 4 0 12 0 32 0 80 0 192 0 448 0 1024 ...
   0 2304 0 5120 0]; % Distance spectrum of code
EbNo = 3:0.5:8;
berbound = bercoding(EbNo,'conv','soft',coderate,dspec);
semilogy(EbNo,berbound) % Plot the results.
xlabel('E_b/N_0 (dB)'); ylabel('Upper Bound on BER');
title('Theoretical Bound on BER for Convolutional Coding');
grid on;
```

The example produces the following plot.



Comparing Theoretical and Empirical Error Rates

The example below uses the `berawgn` function to compute symbol error rates for pulse amplitude modulation (PAM) with a series of E_b/N_0 values. For comparison, the code simulates 8–PAM with an AWGN channel and computes empirical symbol error rates. The code also plots the theoretical and empirical symbol error rates on the same set of axes.

```
% 1. Compute theoretical error rate using BERAWGN.
M = 8; EbNo = [0:13];
ser = berawgn(EbNo,'pam',M).*log2(M);
% Plot theoretical results.
figure; semilogy(EbNo,ser,'r');
xlabel('E_b/N_0 (dB)'); ylabel('Symbol Error Rate');
grid on; drawnow;

% 2. Compute empirical error rate by simulating.
% Set up.
n = 10000; % Number of symbols to process
k = log2(M); % Number of bits per symbol
% Convert from EbNo to SNR.
% Note: Because No = 2*noiseVariance^2, we must add 3 dB
```

```
% to get SNR. For details, see Proakis book listed in
% "Selected Bibliography for Performance Evaluation."
snr = EbNo+3+10*log10(k);
ynoisy=zeros(n,length(snr)); % Preallocate to save time.

% Main steps in the simulation
x = randint(n,1,M); % Create message signal.
y = pammod(x,M); % Modulate.
% Send modulated signal through AWGN channel.
% Loop over different SNR values.
for jj = 1:length(snr)
   ynoisy(:,jj) = awgn(real(y),snr(jj),'measured');
end
z = pamdemod(ynoisy,M); % Demodulate.

% Compute symbol error rate from simulation.
[num,rt] = symerr(x,z);

% 3. Plot empirical results, in same figure.
hold on; semilogy(EbNo,rt,'b.');
legend('Theoretical SER','Empirical SER');
title('Comparing Theoretical and Empirical Error Rates');
hold off;
```

The example produces a plot like the one below. Your plot might vary because the simulation uses random numbers.

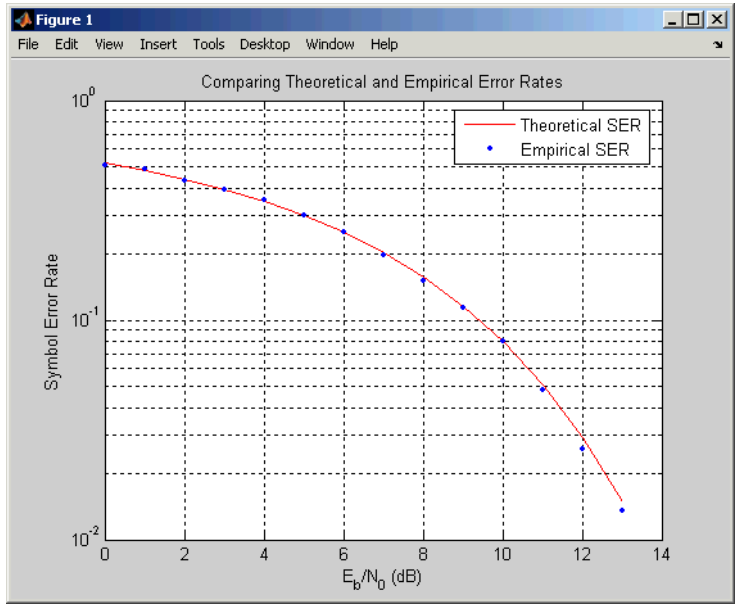


Figure 1
File Edit View Insert Tools Desktop Window Help
Comparing Theoretical and Empirical Error Rates
Theoretical SER
Empirical SER
Symbol Error Rate
10^0
10^{-1}
10^{-2}
0 2 4 6 8 10 12 14
E_b/N_0 (dB)

Error Rate Plots

Error rate plots provide a visual way to examine the performance of a communication system, and they are often included in publications. This section mentions some of the tools that you can use to create error rate plots, modify them to suit your needs, and do curve fitting on error rate data. It also provides an example of curve fitting. For more detailed discussions about the more general plotting capabilities in MATLAB, see the MATLAB documentation set.

Creating Error Rate Plots Using semilogy

In many error rate plots, the horizontal axis indicates E_b/N_0 values in dB and the vertical axis indicates the error rate using a logarithmic (base 10) scale. To see an example of such a plot, as well as the code that creates it, see "Comparing Theoretical and Empirical Error Rates" on page 3-10. The part of that example that creates the plot uses the `semilogy` function to produce a logarithmic scale on the vertical axis and a linear scale on the horizontal axis.

Other examples that illustrate the use of `semilogy` are in these sections:

- "Example: Using the Semianalytic Technique" on page 3-7, which also illustrates
  - Plotting two sets of data on one pair of axes
  - Adding a title
  - Adding a legend
- "Plotting Theoretical Error Rates" on page 3-9, which also illustrates
  - Adding axis labels
  - Adding grid lines

Curve Fitting for Error Rate Plots

Curve fitting is useful when you have a small or imperfect data set but want to plot a smooth curve for presentation purposes. The `berfit` function in the Communications Toolbox offers curve fitting capabilities that are well-suited to the situation when the empirical data describes error rates at different E_b/N_0 values. This function enables you to

- Customize various relevant aspects of the curve-fitting process, such as the type of closed-form function (from a list of preset choices) used to generate the fit.
- Plot empirical data along with a curve that `berfit` fits to the data.
- Interpolate points on the fitted curve between E_b/N_0 values in your empirical data set, to make the plot smoother-looking.
- Collect relevant information about the fit, such as the numerical values of points along the fitted curve and the coefficients of the fit expression.

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.

For a full list of inputs and outputs for `berfit`, see its reference page.

Example: Curve Fitting for an Error Rate Plot

This example simulates a simple DBPSK (differential binary phase shift keying) communication system and plots error rate data for a series of E_b/N_0 values. It uses the `berfit` function to fit a curve to the somewhat rough set of empirical error rates. Because the example is long, this discussion presents it in multiple steps:

Setting Up Parameters for the Simulation

The first step in the example is to set up parameters that will be used during the simulation. Parameters include the range of E_b/N_0 values to consider and the minimum number of errors that must occur before the simulation computes an error rate for that E_b/N_0 value.

Note For most applications, you should base an error rate computation on a larger number of errors than is used here (for instance, you might change `numerrmin` to `100` in the code below). However, as shown, this example uses a small number of errors merely to illustrate how curve fitting can smooth out a rough data set.

```
% Set up initial parameters.
siglen = 1000; % Number of bits in each trial
M = 2; % DBPSK is binary.
EbNomin = 0; EbNomax = 10; % EbNo range, in dB
numerrmin = 5; % Compute BER only after 5 errors occur.
EbNovec = EbNomin:1:EbNomax; % Vector of EbNo values
numEbNos = length(EbNovec); % Number of EbNo values
% Preallocate space for certain data.
ber = zeros(1,numEbNos); % BER values
intv = cell(1,numEbNos); % Cell array of confidence intervals
```

Simulating the System Using a Loop

The next step in the example is to use a `for` loop to vary the E_b/N_0 value (denoted by `EbNo` in the code) and simulate the communication system for each value. The inner `while` loop ensures that the simulation continues to use a given `EbNo` value until at least the predefined minimum number of errors has occurred. When the system is very noisy, this requires only one pass through the `while` loop, but in other cases, this requires multiple passes.

The communication system simulation uses these toolbox functions:

- `randint` to generate a random message sequence
- `dpskmod` to perform DBPSK modulation
- `awgn` to model a channel with additive white Gaussian noise
- `dpskdemod` to perform DBPSK demodulation
- `biterr` to compute the number of errors for a given pass through the `while` loop
- `berconfint` to compute the final error rate and confidence interval for a given value of `EbNo`

As the example progresses through the `for` loop, it collects data for later use in curve fitting and plotting:

- `ber`, a vector containing the bit error rates for the series of `EbNo` values
- `intv`, a cell array containing the confidence intervals for the series of `EbNo` values. Each entry in `intv` is a two-element vector that gives the endpoints of the interval.

```
% Loop over the vector of EbNo values.
for jj = 1:numEbNos
   EbNo = EbNovec(jj);
   snr = EbNo; % Because of binary modulation
   ntrials = 0; % Number of passes through the while loop below
   numerr = 0; % Number of errors for this EbNo value
   % Simulate until numerrmin errors occur.
   while (numerr < numerrmin)
      msg = randint(siglen, 1, M); % Generate message sequence.
      txsig = dpskmod(msg,M); % Modulate.
      rxsig = awgn(txsig, snr, 'measured'); % Add noise.
      decodmsg = dpskdemod(rxsig,M); % Demodulate.
      newerrs = biterr(msg,decodmsg); % Errors in this trial
      numerr = numerr + newerrs; % Total errors for this EbNo value
      ntrials = ntrials + 1; % Update trial index.
   end
   % Error rate and 98% confidence interval for this EbNo value
   [ber(jj), intv1] = berconfint(numerr,(ntrials * siglen),.98);
   intv{jj} = intv1; % Store in cell array for later use.
   disp(['EbNo = ' num2str(EbNo) ' dB, ' num2str(numerr) ...
         ' errors, BER = ' num2str(ber(jj))])
end
```

This part of the example displays output in the Command Window as it progresses through the `for` loop. Your exact output might be different, because the example uses random numbers.

```
EbNo = 0 dB, 182 errors, BER = 0.182
EbNo = 1 dB, 156 errors, BER = 0.156
EbNo = 2 dB, 104 errors, BER = 0.104
EbNo = 3 dB, 66 errors, BER = 0.066
EbNo = 4 dB, 42 errors, BER = 0.042
```

```
EbNo = 5 dB, 27 errors, BER = 0.027
EbNo = 6 dB, 13 errors, BER = 0.0065
EbNo = 7 dB, 7 errors, BER = 0.007
EbNo = 8 dB, 5 errors, BER = 0.00125
EbNo = 9 dB, 5 errors, BER = 0.000625
EbNo = 10 dB, 5 errors, BER = 0.00041667
```

Plotting the Empirical Results and the Fitted Curve

The final part of this example fits a curve to the BER data collected from the simulation loop. It also plots error bars using the output from the `berconfint` function.

```
% Use BERFIT to plot the best fitted curve,
% interpolating to get a smooth plot.
fitEbNo = EbNomin:0.25:EbNomax; % Interpolation values
berfit(EbNovec,ber,fitEbNo);

% Also plot confidence intervals.
hold on;
for jj=1:numEbNos
   semilogy([EbNovec(jj) EbNovec(jj)],intv{jj},'g-+');
end
hold off;
```

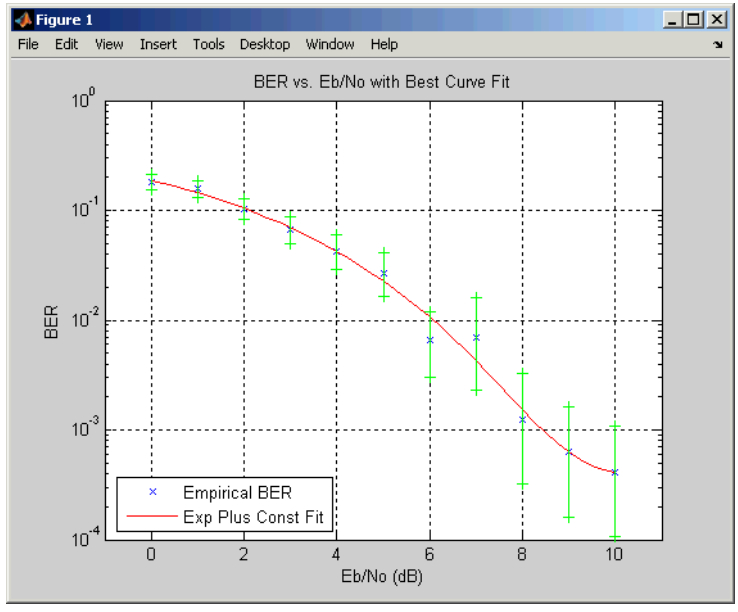


Figure 1
File Edit View Insert Tools Desktop Window Help
BER vs. Eb/No with Best Curve Fit
BER
Eb/No (dB)
Empirical BER
Exp Plus Const Fit

Eye Diagrams

An eye diagram is a simple and convenient tool for studying the effects of intersymbol interference and other channel impairments in digital transmission. To construct an eye diagram, plot the received signal against time on a fixed-interval axis. At the end of the fixed time interval, wrap around to the beginning of the time axis. Thus the diagram consists of many overlapping curves. One way to use an eye diagram is to look for the place where the "eye" is most widely opened, and use that point as the decision point when demapping a demodulated signal to recover a digital message.

To produce an eye diagram from a signal, use the `eyediagram` function. The signal can have different formats, as the table below indicates.

Representing In-Phase and Quadrature Components of Signal

Signal Format	Source of In-Phase Components	Source of Quadrature Components
Real matrix with two columns	First column	Second column
Complex vector	Real part	Imaginary part
Real vector	Vector contents	Quadrature component is always zero

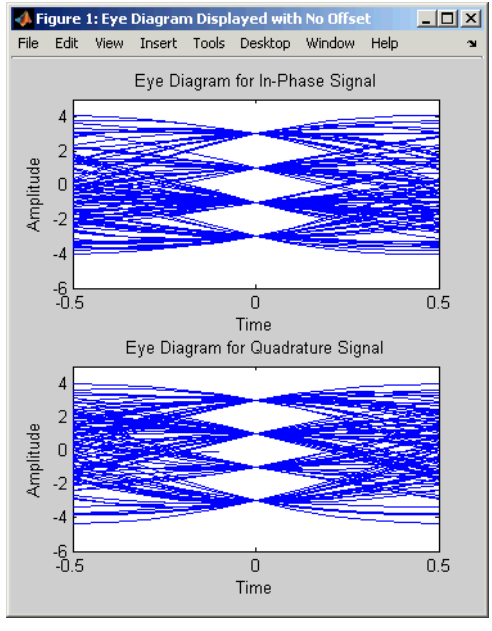
Example: Eye Diagrams

The code below illustrates the use of the eye diagram for finding the best decision point. It maps a random digital signal to a 16-QAM waveform, then uses a raised cosine filter to simulate a noisy transmission channel. Several commands manipulate the filtered data to isolate its steady-state behavior. Then the `eyediagram` command produces an eye diagram from the resulting signal.

```
% Define the M-ary number and sampling rates.
M = 16; Fd = 1; Fs = 10;
Pd = 100; % Number of points in the calculation
```

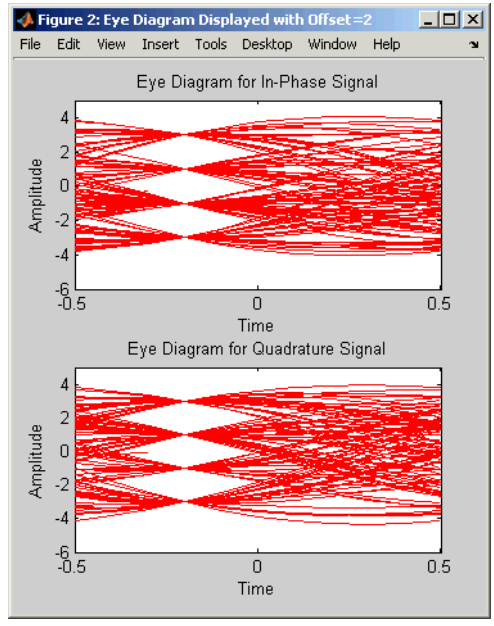
```
msg_d = randint(Pd,1,M); % Random integers in the range [0,M-1]
% Modulate using square QAM.
msg_a = qammod(msg_d,M);
% Assume the channel is equivalent to a raised cosine filter.
delay = 3; % Delay of the raised cosine filter
rcv = rcosflt(msg_a,Fd,Fs,'fir/normal',.5,delay);

% Truncate the output of rcosflt to remove response tails.
N = Fs/Fd;
propdelay = delay .* N + 1; % Propagation delay of filter
rcv1 = rcv(propdelay:end-(propdelay-1),:); % Truncated version

% Plot the eye diagram of the resulting signal sampled and
% displayed with no offset.
offset1 = 0;
h1 =  eyediagram(rcv1,N,1/Fd,offset1);
set(h1,'Name','Eye Diagram Displayed with No Offset');
```

Notice that a vertical line down the center of the diagram would cross the "eye" at its most widely opened point, as in the image below.



If the `eyediagram` command used a different offset value, then a vertical line down the center of the diagram would *not* cross the eye at the most widely opened point. The code and image to illustrate this are below.

```
offset2 = 2;
h2 = eyediagram(rcv1,N,1/Fd,offset2,'r-');
set(h2,'Name','Eye Diagram Displayed with Offset=2');
```



As an additional example of using the `eyediagram` function, the commands below display the eye diagram with no offset, but based on data that is sampled with an offset of two samples. This sampling offset simulates errors in timing that result from being two samples away from perfect synchronization.

```
h3 = eyediagram(rcv1(1+offset2:end,:),N,1/Fd,0);
set(h3,'Name','Eye Diagram Sampled with Offset=2');
```

Scatter Plots

A scatter plot of a signal shows the signal's value at a given decision point. In the best case, the decision point should be at the time when the eye of the signal's eye diagram is the most widely open.

To produce a scatter plot from a signal, use the `scatterplot` function. The signal can have different formats, as in the case of the `eyediagram` function. See the table Representing In-Phase and Quadrature Components of Signal on page 3-19 for details.

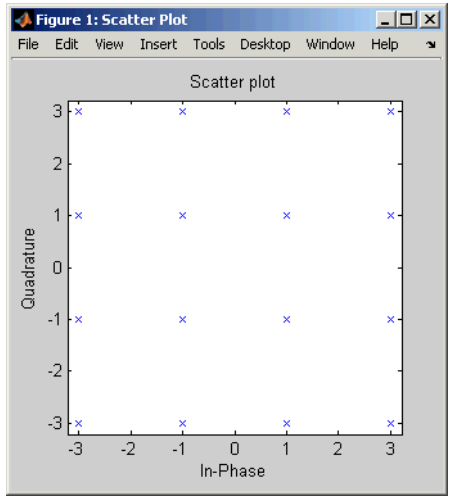
Scatter plots are often used to visualize the signal constellation associated with digital modulation. For more information, see "Plotting Signal Constellations" on page 8-11.

Example: Scatter Plots

The code below is similar to the example from the section "Example: Eye Diagrams" on page 3-19. It produces a scatter plot from the received analog signal, instead of an eye diagram.

```
% Define the M-ary number and sampling rates.
M = 16; Fd = 1; Fs = 10; N = Fs/Fd;
Pd = 200; % Number of points in the calculation
msg_d = randint(Pd,1,M); % Random integers in the range [0,M-1]
% Modulate using square QAM.
msg_a = qammod(msg_d,M);
% Upsample the modulated signal.
msg_a = rectpulse(msg_a,N);
% Assume the channel is equivalent to a raised cosine filter.
rcv = rcosflt(msg_a,Fd,Fs);
% Create the scatter plot of the received signal,
% ignoring the first three and the last four symbols.
rcv_a = rcv(3*N+1:end-4*N,:);
h = scatterplot(rcv_a,N,0,'bx');
```

Varying the third parameter in the `scatterplot` command changes the offset. An offset of zero yields optimal results, shown below.



The image below illustrates two offsets that are not optimal. The x's and +'s reflect offsets that are too late and too early, respectively. Notice that in the diagram, the dots are the actual constellation points, while the other symbols are perturbations of those points.

```
hold on;
scatterplot(rcv_a,N,N+1,'r+',h); % Plot +'s
scatterplot(rcv_a,N,N-1,'mx',h); % Plot x's
scatterplot(rcv_a,N,0,'b.',h); % Plot dots
```

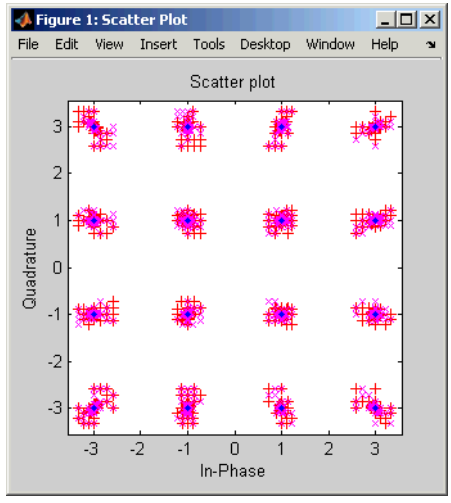


Figure 1: Scatter Plot
File Edit View Insert Tools Desktop Window Help
Scatter plot
Quadrature
In-Phase
3
2
1
0
-1
-2
-3
-3 -2 -1 0 1 2 3

Selected Bibliography for Performance Evaluation

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

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

[3] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[4] Lindsey, William C., and Marvin K. Simon, *Telecommunication Systems Engineering*, Englewood Cliffs, N.J., Prentice-Hall, 1973.

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

[6] Spilker, James J., *Digital Communications by Satellite*, Englewood Cliffs, N.J., Prentice-Hall, 1977.

BERTool: A Bit Error Rate Analysis GUI

The following sections describe the Bit Error Rate Analysis Tool (BERTool) and provide examples showing how to use this GUI.

Summary of Features

BERTool is an interactive GUI for analyzing communication systems' bit error rate (BER) performance. Using BERTool you can

- Generate BER data for a communication system using
  - Closed-form expressions for theoretical BER performance of selected types of communication systems.
  - The semianalytic technique.
  - Simulations contained in MATLAB simulation functions or Simulink models. After you create a function or model that simulates the system, BERTool iterates over your choice of E_b/N_0 values and collects the results.
- Plot one or more BER data sets on a single set of axes. For example, you can graphically compare simulation data with theoretical results, or simulation data from a series of similar models of a communication system.
- Fit a curve to a set of simulation data.
- Send BER data to the MATLAB workspace or to a file for any further processing that you might want to perform.

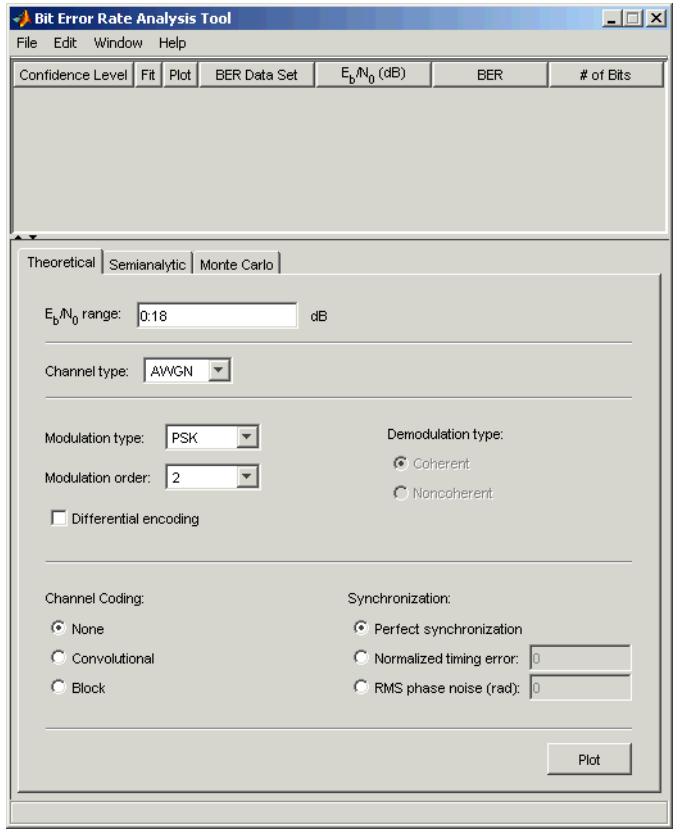
For an animated demonstration of BERTool, see the Bit Error Rate Analysis Tool demo.

Note BERTool is designed for analyzing bit error rates only, not symbol error rates, word error rates, or other types of error rates. If, for example, your simulation computes a symbol error rate (SER), then you should convert the SER to a BER before using the simulation with BERTool.

Opening BERTool

To open BERTool, type

```
bertool
```



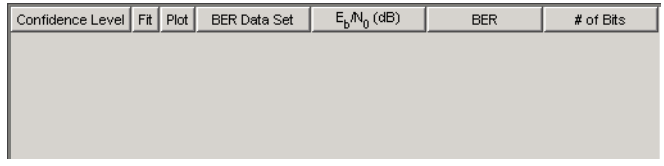
The BERTool Environment

This section gives an overview of the components of BERTool and how they interact with each other.

Components of BERTool

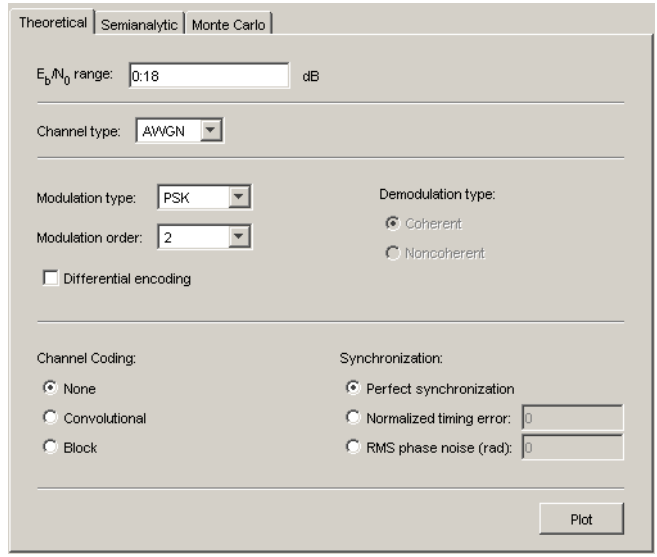
BERTool includes these components:

- A data viewer pane at the top. It is initially empty.



  After you instruct BERTool to generate one or more BER data sets, they appear in the data viewer. An example that shows how data sets look in the data viewer is in "Example: Using a MATLAB Simulation with BERTool" on page 4-20.

- A set of tabbed panels on the bottom. Labeled **Theoretical**, **Semianalytic**, and **Monte Carlo**, the panels correspond to the different methods by which BERTool can generate BER data.



To learn more about each of the methods, see

- "Computing Theoretical BERs" on page 4-7
- "Using the Semianalytic Technique to Compute BERs" on page 4-14
- "Running MATLAB Simulations" on page 4-20 or "Running Simulink Simulations" on page 4-35

- A separate **BER Figure** window, which displays some or all of the BER data sets that are listed in the data viewer. BERTool opens the **BER Figure** window after it has at least one data set to display, so you do not see the **BER Figure** window when you first open BERTool. For an example of how the **BER Figure** window looks, see "Example: Using the Theoretical Panel in BERTool" on page 4-8.

Interaction Among BERTool Components

The components of BERTool act as one integrated tool. These behaviors reflect their integration:

- If you select a data set in the data viewer, then BERTool reconfigures the tabbed panels to reflect the parameters associated with that data set and also highlights the corresponding data in the **BER Figure** window. This is

useful if the data viewer displays multiple data sets and you want to recall the meaning and origin of each data set.

- If you click data plotted in the **BER Figure** window, then BERTool reconfigures the tabbed panels to reflect the parameters associated with that data and also highlights the corresponding data set in the data viewer.
- If you configure the **Semianalytic** or **Theoretical** panel in a way that is already reflected in an existing data set, then BERTool highlights that data set in the data viewer. This prevents BERTool from duplicating its computations and its entries in the data viewer, while still showing you the results that you requested.
- If you close the **BER Figure** window, then you can reopen it by choosing **BER Figure** from the **Window** menu in BERTool.
- If you select options in the data viewer that affect the BER plot, then the **BER Figure** window reflects your selections immediately. Such options relate to data set names, confidence intervals, curve fitting, and the presence or absence of specific data sets in the BER plot.

Note If you want to observe the integration yourself but do not yet have any data sets in BERTool, then first try the procedure in "Example: Using the Theoretical Panel in BERTool" on page 4-8.

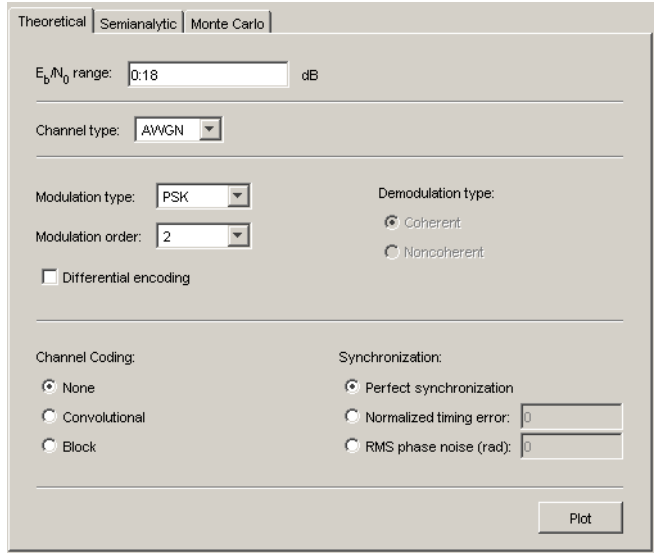
Note If you save the **BER Figure** window using the window's **File** menu, then the resulting file contains the contents of the window but not the BERTool data that led to the plot. To save an entire BERTool session, see "Saving a BERTool Session" on page 4-53.

Computing Theoretical BERs

You can use BERTool to generate and analyze theoretical BER data. Theoretical data is useful for comparison with your simulation results. However, closed-form BER expressions exist only for certain kinds of communication systems.

To access the capabilities of BERTool related to theoretical BER data, use this procedure:

1 Open BERTool and go to the **Theoretical** panel.



2 Set parameters to reflect the system whose performance you want to analyze. Some parameters are visible and active only when other parameters have specific values. See “Available Sets of Theoretical BER Data” on page 4-10 for details.

3 Click **Plot**.

For an example that shows how to generate and analyze theoretical BER data via BERTool, see “Example: Using the Theoretical Panel in BERTool” on page 4-8.

Also, “Available Sets of Theoretical BER Data” on page 4-10 indicates which combinations of parameters are available on the **Theoretical** panel, as well as which underlying functions perform computations.

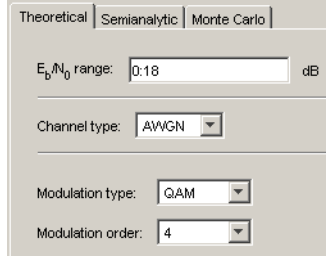
Example: Using the Theoretical Panel in BERTool

This example illustrates how to use BERTool to generate and plot theoretical BER data. In particular, the example compares the performance of a communication system that uses an AWGN channel and QAM modulation of different orders.

Running the Theoretical Example

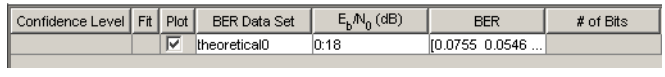
1 Open BERTool and go to the **Theoretical** panel.

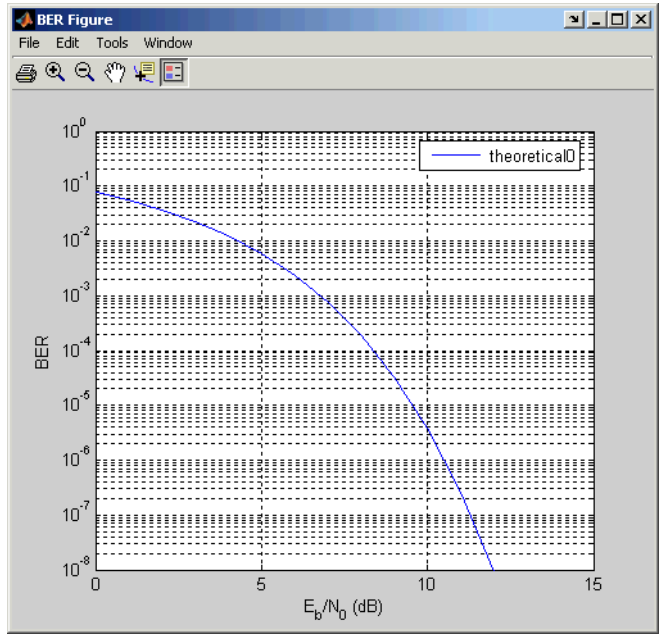
2 Set parameters as shown below.



3 Click **Plot**.

BERTool creates an entry in the data viewer and plots the data in the **BER Figure** window. Even though the parameters above requested that E_b/N_0 go up to 18, BERTool plots only those BER values that are at least 10^{-8}.





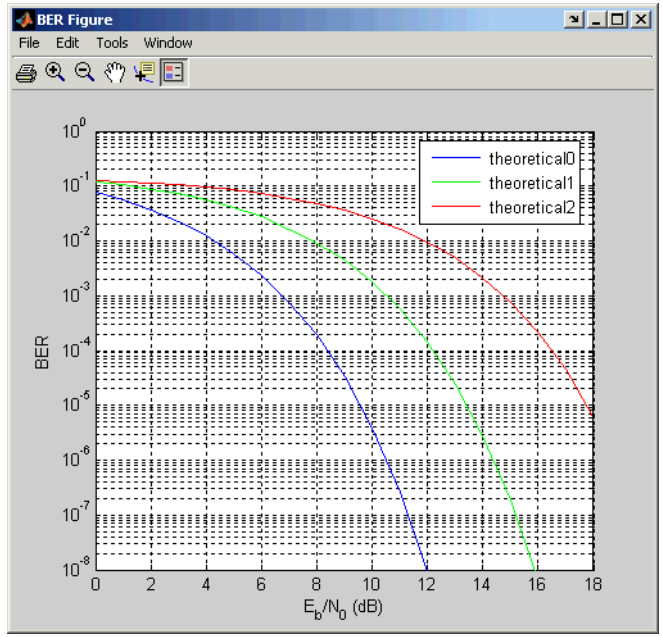
4 Change the **Modulation order** parameter to 16 and click **Plot**.

BERTool creates another entry in the data viewer and plots the new data in the same **BER Figure** window (not pictured).

5 Change the **Modulation order** parameter to 64 and click **Plot**.

BERTool creates another entry in the data viewer and plots the new data in the same **BER Figure** window.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	theoretical0	0:18	[0.0755 0.0546 ...	
		☑	theoretical1	0:18	[0.1197 0.1043 ...	
		☑	theoretical2	0:18	[0.1280 0.1216 ...	



6 To recall which value of **Modulation order** corresponds to a given curve, click the curve; BERTool responds by adjusting the parameters in the **Theoretical** panel to reflect the values that correspond to that curve.

7 To remove the last curve from the plot (but not from the data viewer), clear the check box in the last entry of the data viewer in the **Plot** column. To restore the curve to the plot, check the check box again.

Available Sets of Theoretical BER Data

Available combinations of valid combinations of parameters for the **Theoretical** panel in BERTool depend on the types of systems for which closed-form expressions exist for error statistics. The **Theoretical** panel adjusts itself to your choices, so that the combination of parameters is always valid. Note that you can set the **Modulation order** parameter by selecting a choice from the menu or by typing a value in the field. The **Normalized timing error** must be between 0 and 0.5.

Combinations of Parameters for AWGN Channel Systems

The table below lists the available sets of theoretical BER data for systems that use an AWGN channel.

Modulation	Modulation Order	Other Choices
PSK (assuming Gray-coded signal constellation)	2	Differential or nondifferential encoding; Channel coding = **None**; other choices related to synchronization
		Differential or nondifferential encoding; **Block** or **Convolutional** coding; **Hard** or **Soft** decisions; other choices related to code
	4	Differential or nondifferential encoding; Channel coding = **None**
		Differential or nondifferential encoding; **Block** or **Convolutional** coding; **Hard** or **Soft** decisions; other choices related to code
	8, 16, 32, or a higher power of 2	
DPSK	2, 4, 8, 16, 32, or a higher power of 2	
PAM (assuming Gray-coded signal constellation)	2, 4, 8, 16, 32, or a higher power of 2	
QAM (assuming Gray-coded signal constellation)	4, 8, 16, 32, 64, 128, 256, 512, 1024, or a higher power of 2	

Modulation	Modulation Order	Other Choices
FSK	2, 4, 8, 16, or a higher power of 2	**Coherent** demodulation
	2, 4, 8, 16, 32, or 64	**Noncoherent** demodulation
MSK	2	Differential or nondifferential encoding
CPFSK	2, 4, 8, 16, or a higher power of 2	**Modulation index** > 0

For more information about specific combinations of parameters, including bibliographic references that contain closed-form expressions, see these functions' reference pages:

- `berawgn` — For systems with no coding and perfect synchronization
- `bercoding` — For systems with channel coding
- `bersync` — For systems with BPSK modulation, no coding, and imperfect synchronization

Combinations of Parameters for Rayleigh Channel Systems

The table below lists the available sets of theoretical BER data for systems that use a Rayleigh channel.

Modulation	Modulation Order	Other Choices
PSK	2 or 4	**Diversity order** $\geqq 1$
	8, 16, 32, or a higher power of 2	
DPSK	2 or 4	**Diversity order** $\geqq 1$
	8, 16, 32, or a higher power of 2	

Modulation	Modulation Order	Other Choices
FSK	2	**Diversity order** $\geqq 1$; **Coherent** or **Noncoherent** demodulation
	4, 8, 16, or a higher power of 2	**Diversity order** $\geqq 1$

For more information about specific combinations of parameters, including bibliographic references that contain closed-form expressions, see the reference page for the `berfading` function.

Combinations of Parameters for Rician Channel Systems

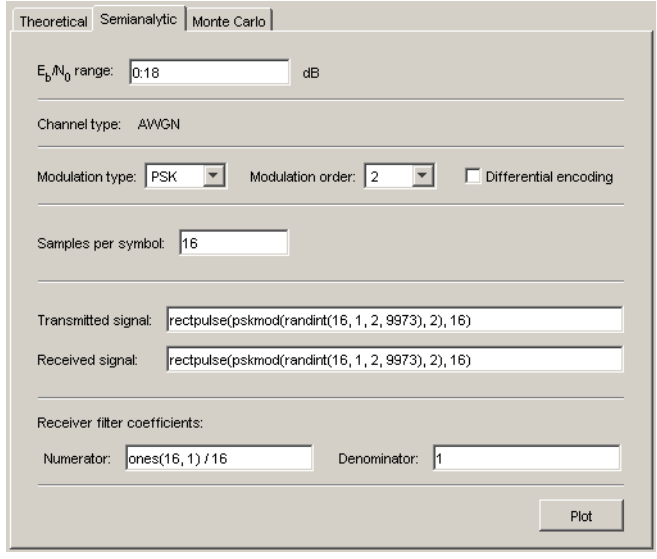
The table below lists the available sets of theoretical BER data for systems that use a Rician channel.

Name	Range
Modulation type	**PSK**
Modulation order	2
Diversity order	1
Demodulation type	**Coherent**
K factor	[0, inf]
RMS phase noise	[0, inf]

Using the Semianalytic Technique to Compute BERs

You can use BERTool to generate and analyze BER data via the semianalytic technique. The semianalytic technique is discussed in "Performance Results via the Semianalytic Technique" on page 3-5, and "When to Use the Semianalytic Technique" on page 3-5 is particularly relevant as background material.

To access the semianalytic capabilities of BERTool, open the **Semianalytic** panel.



These topics describe how to use the semianalytic technique via BERTool:

- "Example: Using the Semianalytic Panel in BERTool" on page 4-15
- "Procedure for Using the Semianalytic Panel in BERTool" on page 4-17

For further details about how BERTool applies the semianalytic technique, see the reference page for the `semianalytic` function, which BERTool uses to perform computations.

Example: Using the Semianalytic Panel in BERTool

This example illustrates how BERTool applies the semianalytic technique, using 16-QAM modulation. This example is a variation on the example in "Example: Using the Semianalytic Technique" on page 3-7, tailored to use BERTool instead of using the `semianalytic` function directly.

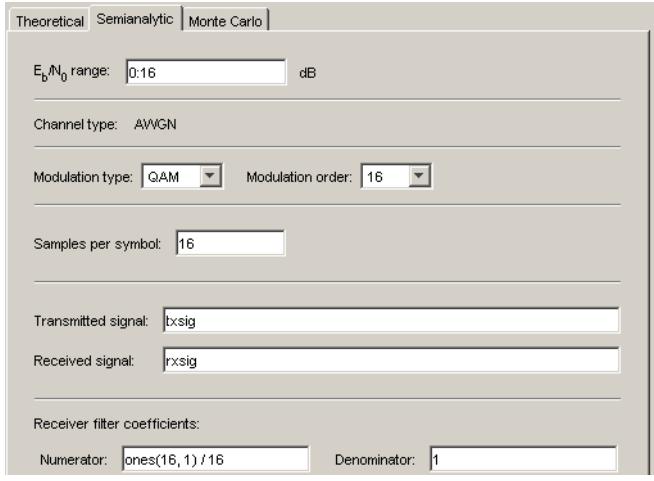
Running the Semianalytic Example

1 To set up the transmitted and received signals, run steps 1 through 4 from the code example in "Example: Using the Semianalytic Technique" on page 3-7. The code is repeated below.

```
% Step 1. Generate message signal of length >= M^L.
M = 16; % Alphabet size of modulation
L = 1; % Length of impulse response of channel
msg = [0:M-1 0]; % M-ary message sequence of length > M^L

% Step 2. Modulate the message signal using baseband modulation.
modsig = qammod(msg,M); % Use 16-QAM.
Nsamp = 16;
modsig = rectpulse(modsig,Nsamp); % Use rectangular pulse shaping.

% Step 3. Apply a transmit filter.
txsig = modsig; % No filter in this example

% Step 4. Run txsig through a noiseless channel.
rxsig = txsig*exp(j*pi/180); % Static phase offset of 1 degree
```

2 Open BERTool and go to the **Semianalytic** panel.

3 Set parameters as shown below.



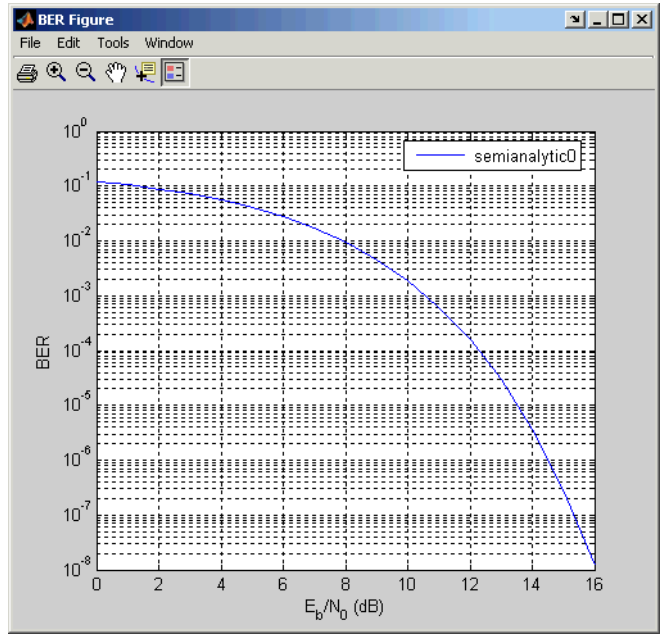
4 Click **Plot**.

Visible Results of the Semianalytic Example

After you click **Plot**, BERTool creates a listing for the resulting data in the data viewer.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	semianalytic0	0:16	[0.1199 0.1044 ...	[272]

BERTool plots the data in the **BER Figure** window.



Procedure for Using the Semianalytic Panel in BERTool

The procedure below describes how you would typically implement the semianalytic technique using BERTool:

1. Generate a message signal containing *at least* M^L symbols, where M is the alphabet size of the modulation and L is the length of the impulse response of the channel, in symbols. 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.

2. Modulate a carrier with the message signal using baseband modulation. Supported modulation types are listed on the reference page for `semianalytic`.

3. Filter the modulated signal with a transmit filter. This filter is often a square-root raised cosine filter, but you can also use a Butterworth, Bessel,

Chebyshev type 1 or 2, elliptic, or more general FIR or IIR filter. Store the result of this step as `txsig` for later use.

4. Run the filtered signal through a *noiseless* channel. This channel can include multipath fading effects, phase shifts, amplifier nonlinearities, quantization, and additional filtering, but it must not include noise. Store the result of this step as `rxsig` for later use.

5. On the **Semianalytic** panel of BERTool, enter parameters as in the table below.

Parameter Name	Meaning
Eb/No range	A vector that lists the values of E_b/N_0 for which you want to collect BER data. The value in this field can be a MATLAB expression or the name of a variable in the MATLAB workspace.
Modulation type	These parameters describe the modulation scheme that you used earlier in this procedure.
Modulation order	
Differential encoding	This check box, which is visible and active for MSK and PSK modulation, enables you to choose between differential and nondifferential encoding.
Samples per symbol	The number of samples per symbol in the transmitted signal. This value is also the sampling rate of the transmitted and received signals, in Hz.
Transmitted signal	The `txsig` signal that you generated earlier in this procedure
Received signal	The `rxsig` signal that you generated earlier in this procedure
Numerator	Coefficients of the receiver filter that BERTool applies to the received signal
Denominator	

Note Consistency among the values in the GUI is important. For example, if the signal referenced in the **Transmitted signal** field was generated using DPSK and you set **Modulation type** to `MSK` then the results might not be meaningful.

6 Click **Plot**.

Semianalytic Computations and Results

After you click **Plot**, BERTool performs these tasks:

- Filters `rxsig` and then determines the error probability of each received signal point by analytically applying the Gaussian noise distribution to each point. BERTool averages the error probabilities over the entire received signal to determine the overall error probability. If the error probability calculated in this way is a symbol error probability, then BERTool converts it to a bit error rate, typically by assuming Gray coding. (If the modulation type is DQPSK or cross QAM, the result is an upper bound on the bit error rate rather than the bit error rate itself.)
- Enters the resulting BER data in the data viewer of the BERTool window.
- Plots the resulting BER data in the **BER Figure** window.

Running MATLAB Simulations

You can use BERTool in conjunction with your own MATLAB simulation functions to generate and analyze BER data. The MATLAB function simulates the communication system whose performance you want to study. BERTool invokes the simulation for E_b/N_0 values that you specify, collects the BER data from the simulation, and creates a plot. BERTool also enables you to easily change the E_b/N_0 range and stopping criteria for the simulation. The topics in this section are

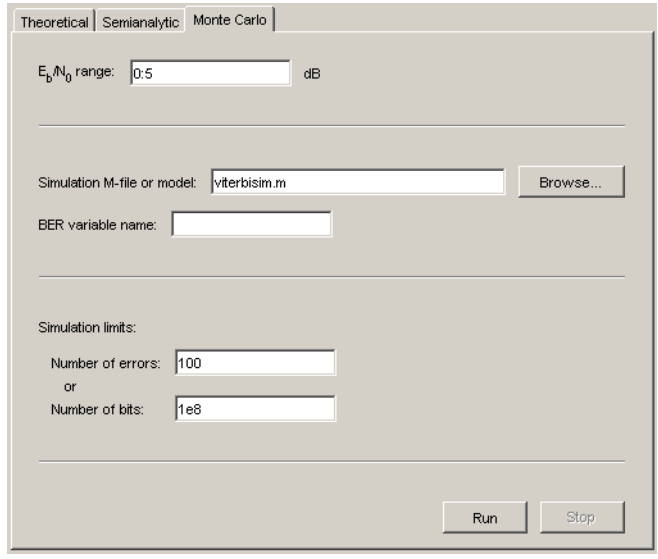
To learn how to make your own simulation functions compatible with BERTool, see "Preparing Simulation Functions for Use with BERTool" on page 4-27.

Example: Using a MATLAB Simulation with BERTool

This example illustrates how BERTool can run a MATLAB simulation function. The function is `viterbisim`, one of the demonstration files included with the Communications Toolbox.

To run this example, follow these steps:

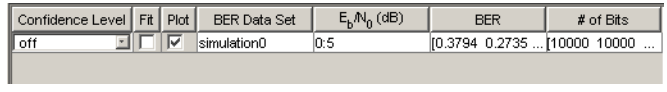
1. Open BERTool and go to the **Monte Carlo** panel. (The default parameters depend on whether you have the Communications Blockset installed. Also note that the **BER variable name** field applies only to Simulink models.)
2. Set parameters as shown below.



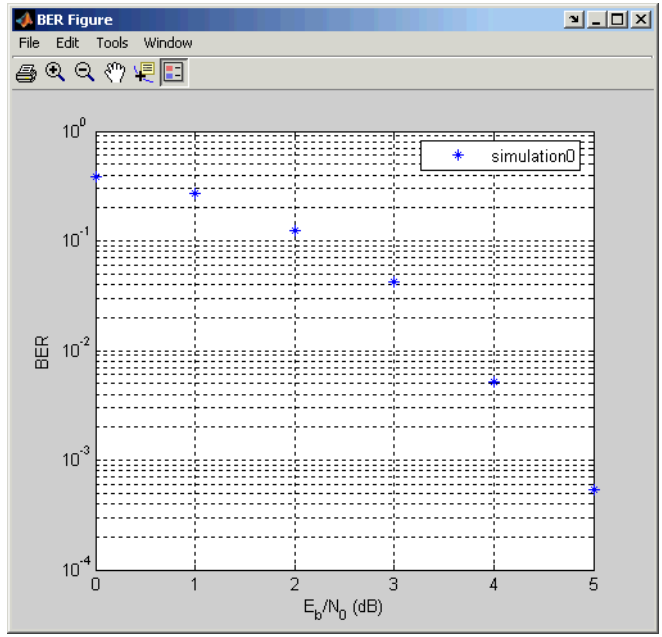
3 Click **Run**.

BERTool runs the simulation function once for each specified value of E_b/N_0 and gathers BER data. (Note that while BERTool is busy with this task, it cannot process certain other tasks, including plotting data from the other panels of the GUI.)

Then BERTool creates a listing in the data viewer.



BERTool plots the data in the **BER Figure** window.

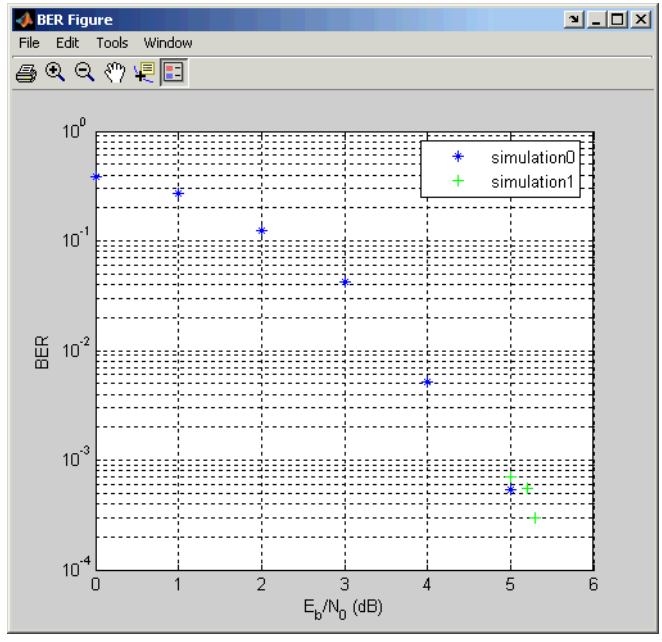


4 To change the range of E_b/N_0 while reducing the number of bits processed in each case, type `[5 5.2 5.3]` in the **Eb/No range** field, type `1e5` in the **Number of bits** field, and click **Run**.

BERTool runs the simulation function again for each new value of E_b/N_0 and gathers new BER data. Then BERTool creates another listing in the data viewer.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☐	☑	simulation0	0:5	[0.3794 0.2735 ...	[10000 10000 ...
off		☑	simulation1	[5 5.2 5.3]	[7.01E-4 5.45E-...	[89744 89744 ...

BERTool plots the data in the **BER Figure** window, adjusting the horizontal axis to accommodate the new data.



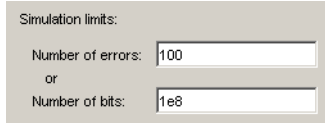
Notice that the two points corresponding to 5 dB from the two data sets are different; this is because the smaller value of **Number of bits** in the second simulation caused the simulation to end before observing many errors. To learn more about the criteria that BERTool uses for ending simulations, see “Varying the Stopping Criteria” on page 4-23.

For another example that uses BERTool to run a MATLAB simulation function, see “Example: Preparing a Simulation Function for Use with BERTool” on page 4-31.

Varying the Stopping Criteria

When you create a MATLAB simulation function for use with BERTool, you must control the flow so that the simulation ends when it either detects a target number of errors or processes a maximum number of bits, whichever occurs first. To learn more about this requirement, see “Requirements for Functions” on page 4-27; for an example, see “Example: Preparing a Simulation Function for Use with BERTool” on page 4-31.

After creating your function, you set the target number of errors and the maximum number of bits in the **Monte Carlo** panel of BERTool.



Typically, a **Number of errors** value of at least `100` produces an accurate error rate. The **Number of bits** value prevents the simulation from running too long, especially at large values of E_b/N_0. However, if the **Number of bits** value is so small that the simulation collects very few errors, then the error rate might not be accurate. You can use confidence intervals to gauge the accuracy of the error rates that your simulation produces; the larger the confidence interval, the less accurate the computed error rate.

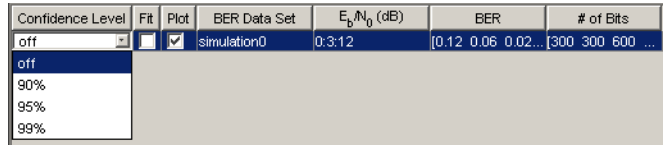
As an example, follow the procedure described in "Example: Using a MATLAB Simulation with BERTool" on page 4-20 and then set **Confidence Level** to `95` for each of the two data sets. Notice that the confidence intervals for the second data set are larger than those for the first data set. This is because the second data set uses a small value for **Number of bits**, relative to the communication system properties and the values in **Eb/No range**, resulting in BER values based on only a small number of observed errors.

Note You can also use the **Stop** button in BERTool to stop a series of simulations prematurely, as long as your function is set up to detect and react to the button press.

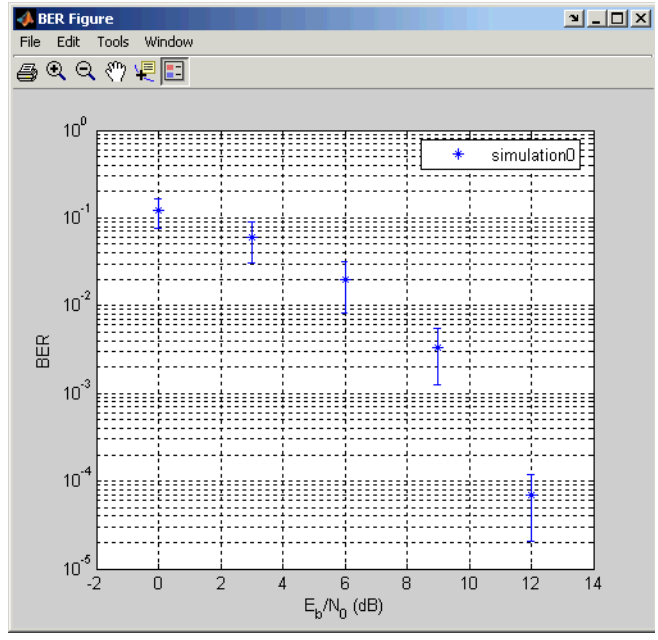
Plotting Confidence Intervals

After you run a simulation with BERTool, the resulting data set in the data viewer has an active menu in the **Confidence Level** column. The default value is `off`, so that the simulation data in the **BER Figure** window does not show confidence intervals.

To show confidence intervals in the **BER Figure** window, set **Confidence Level** to a numerical value: `90%`, `95%`, or `99%`.



The plot in the **BER Figure** window responds immediately to your choice. A sample plot is below.



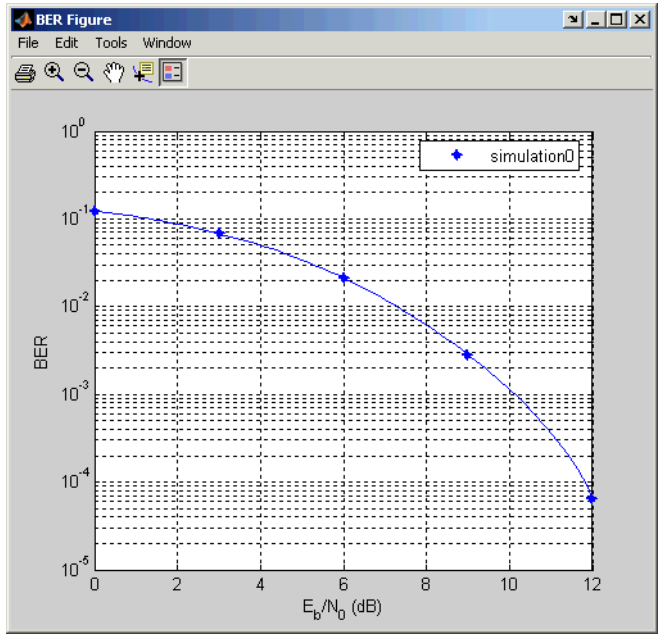
For an example that plots confidence intervals for a Simulink simulation, see “Example: Using a Simulink Model with BERTool” on page 4-36.

To find confidence intervals for levels not listed in the **Confidence Level** menu, use the `berconfint` function.

Fitting BER Points to a Curve

After you run a simulation with BERTool, the **BER Figure** window plots individual BER data points. To fit a curve to a data set that contains at least four points, check the box in the **Fit** column of the data viewer.

The plot in the **BER Figure** window responds immediately to your choice. A sample plot is below.



For an example that performs curve fitting for data from a Simulink simulation and generates the plot shown above, see “Example: Using a Simulink Model with BERTool” on page 4-36. For an example that performs curve fitting for data from a MATLAB simulation function, see “Example: Preparing a Simulation Function for Use with BERTool” on page 4-31.

For greater flexibility in the process of fitting a curve to BER data, use the `berfit` function.

Preparing Simulation Functions for Use with BERTool

A MATLAB simulation function that you use with BERTool must have certain properties. This section lists the requirements, provides a template that you can use when adapting your code to work with BERTool, and provides an example. The topics are

Requirements for Functions

When you create a MATLAB function for use with BERTool, you must ensure that the function interacts properly with the GUI. This section describes the inputs, outputs, and basic operation of a BERTool-compatible function.

Input Arguments

BERTool evaluates your entries in fields of the GUI and passes data to the function as these input arguments, in sequence:

- One value from the **Eb/No range** vector each time BERTool invokes the simulation function
- The **Number of errors** value
- The **Number of bits** value

Output Arguments

Your simulation function must compute and return these output arguments, in sequence:

- Bit error rate of the simulation
- Number of bits processed when computing the BER

BERTool uses these output arguments when reporting and plotting results.

Simulation Operation

Your simulation function must perform these tasks:

- Simulate the communication system for the E_b/N_0 value specified in the first input argument.
- Stop simulating when the number of errors or the number of processed bits equals or exceeds the corresponding threshold specified in the second or third input argument, respectively.
- Detect whether you click the **Stop** button in BERTool and abort the simulation in that case.

Template for a Simulation Function

Below is a template that you can use when adapting your code to work with BERTool. You can open it in an editor by entering `edit bertooltemplate` in the MATLAB Command Window. The description in "Understanding the Template" on page 4-29 explains the template's key sections, while "Using the Template" on page 4-30 indicates how to use the template with your own simulation code. Alternatively, you can develop your simulation function without using the template, but be sure that it satisfies the requirements described in "Requirements for Functions" on page 4-27.

Note The template is not yet ready for use with BERTool. You must insert your own simulation code in the places marked `INSERT YOUR CODE HERE`. For a complete example based on this template, see "Example: Preparing a Simulation Function for Use with BERTool" on page 4-31.

```
function [ber, numBits] = bertooltemplate(EbNo, maxNumErrs, maxNumBits)
% Import Java class for BERTool.
import com.mathworks.toolbox.comm.BERTool;

% Initialize variables related to exit criteria.
totErr = 0;  % Number of errors observed
numBits = 0; % Number of bits processed

% --- Set up parameters. ---
% --- INSERT YOUR CODE HERE.
```

```
% Simulate until number of errors exceeds maxNumErrs
% or number of bits processed exceeds maxNumBits.
while((totErr < maxNumErrs) && (numBits < maxNumBits))

   % Check if the user clicked the Stop button of BERTool.
   if (BERTool.getSimulationStop)
      break;
   end

   % --- Proceed with simulation.
   % --- Be sure to update totErr and numBits.
   % --- INSERT YOUR CODE HERE.
end % End of loop

% Compute the BER.
ber = totErr/numBits;
```

Understanding the Template

From studying the code in the function template above, you can observe how the function either satisfies the requirements listed in "Requirements for Functions" on page 4-27 or indicates where your own insertions of code should do so. In particular,

- The function has appropriate input and output arguments.
- The function includes a placeholder for code that simulates a system for the given E_b/N_0 value.
- The function uses a loop structure to stop simulating when the number of errors exceeds `maxNumErrs` or the number of bits exceeds `maxNumBits`, whichever occurs first.

Note Although the `while` statement of the loop describes the exit criteria, your own code inserted into the section marked `Proceed with simulation` must compute the number of errors and the number of bits. If you do not perform these computations in your own code, then clicking **Stop** is the only way to terminate the loop.

- In each iteration of the loop, the function detects when the user clicks the **Stop** button in BERTool.

Using the Template

Here is a procedure for using the template with your own simulation code:

1. Determine the setup tasks that you must perform. For example, you might want to initialize variables containing the modulation alphabet size, filter coefficients, a convolutional coding trellis, or the states of a convolutional interleaver. Place the code for these setup tasks in the template section marked `Set up parameters`.

2. Determine the core simulation tasks, assuming that all setup work has already been performed. For example, these tasks might include error-control coding, modulation/demodulation, and channel modeling. Place the code for these core simulation tasks in the template section marked `Proceed with simulation`.

3. Also in the template section marked `Proceed with simulation`, include code that updates the values of `totErr` and `numBits`. The quantity `totErr` represents the number of errors observed so far. The quantity `numBits` represents the number of bits processed so far. The computations to update these variables depend on how your core simulation tasks work.

   Note Updating the numbers of errors and bits is important for ensuring that the loop terminates. However, if you accidentally create an infinite loop early in your development work using the function template, then you can use the **Stop** button in BERTool to abort the simulation.

4. Omit any setup code that initializes `EbNo`, `maxNumErrs`, or `maxNumBits`, because BERTool passes these quantities to the function as input arguments, after evaluating the data entered in the GUI.

5. Adjust your code or the template's code as necessary to use consistent variable names and meanings. For example, if your original code uses a variable called `ebn0` and the template's function declaration (first line) uses the variable name `EbNo`, then you must change one of the names so that

they match. As another example, if your original code uses SNR instead of E_b/N_0, then you must convert quantities appropriately.

Example: Preparing a Simulation Function for Use with BERTool

This section adapts the function template given in "Template for a Simulation Function" on page 4-28 to use simulation code from the documentation example in "Example: Curve Fitting for an Error Rate Plot" on page 3-14.

Preparing the Function

To prepare the function for use with BERTool, follow these steps:

1. Copy the template from "Template for a Simulation Function" on page 4-28 into a new M-file in the MATLAB Editor. Save it in a directory on your MATLAB path, using the filename `bertool_simfcn`.

2. From the original example, the following lines are setup tasks. They are modified from the original example to rely on the input arguments that BERTool provides to the function, instead of defining variables such as `EbNovec` and `numerrmin` directly.

   ```
   % Set up initial parameters.
   siglen = 1000; % Number of bits in each trial
   M = 2; % DBPSK is binary.
   snr = EbNo; % Because of binary modulation
   ntrials = 0; % Number of passes through the loop
   ```

   Place these lines of code in the template section marked `Set up parameters.`

3. From the original example, the following lines are the core simulation tasks, after all setup work has been performed.

   ```
   msg = randint(siglen, 1, M); % Generate message sequence.
   txsig = dpskmod(msg,M); % Modulate.
   rxsig = awgn(txsig, snr, 'measured'); % Add noise.
   decodmsg = dpskdemod(rxsig,M); % Demodulate.
   newerrs = biterr(msg,decodmsg); % Errors in this trial
   ntrials = ntrials + 1; % Update trial index.
   ```

Place the code for these core simulation tasks in the template section marked `Proceed with simulation`.

4. Also in the template section marked `Proceed with simulation` (after the code from the previous step), include the following new lines of code that update the values of `totErr` and `numBits`.

```
% Update the total number of errors.
totErr = totErr + newerrs;

% Update the total number of bits processed.
numBits = ntrials * siglen;
```

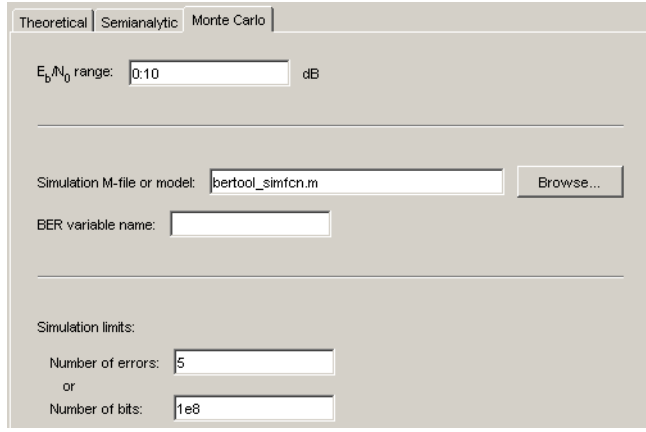
The `bertool_simfcn` function is now compatible with BERTool. Note that unlike the original example, the function here does *not* initialize `EbNovec`, define `EbNo` as a scalar, or use `numerrmin` as the target number of errors; this is because BERTool provides input arguments for similar quantities. The `bertool_simfcn` function also excludes code related to plotting, curve fitting, and confidence intervals in the original example because BERTool enables you to do similar tasks interactively without writing code.

Using the Prepared Function

To use `bertool_simfcn` in conjunction with BERTool, continue the example by following these steps:

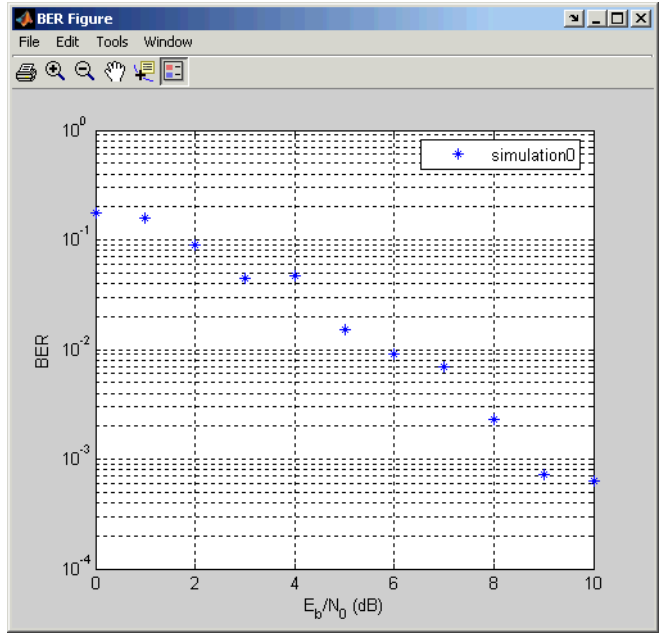
1. Open BERTool and go to the **Monte Carlo** panel.

2. Set parameters on the **Monte Carlo** panel as shown below.



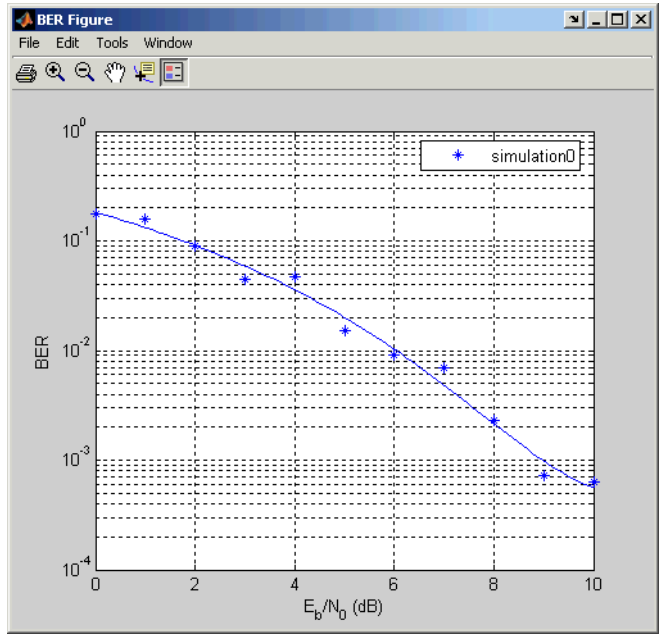
3 Click **Run**.

BERTool spends some time computing results and then plots them. Note that they do not appear to fall along a smooth curve because the simulation required only five errors for each value in `EbNo`.



4 To fit a curve to the series of points in the **BER Figure** window, check the box next to **Fit** in the data viewer.

BERTool plots the curve, as below.

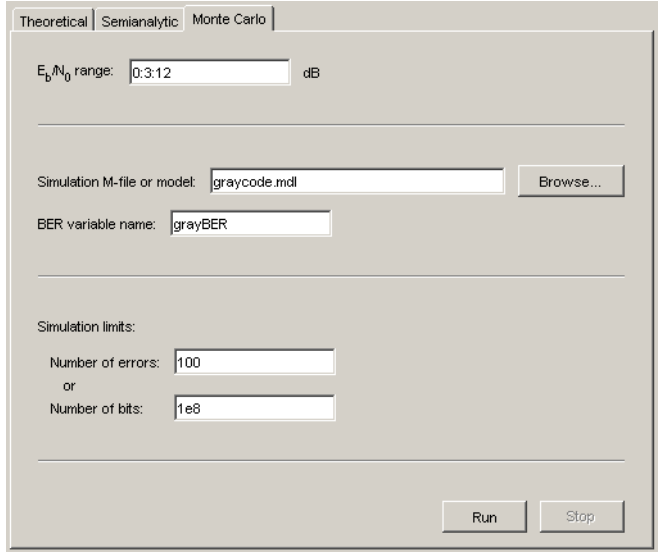


Running Simulink Simulations

You can use BERTool in conjunction with Simulink models to generate and analyze BER data. The Simulink model simulates the communication system whose performance you want to study, while BERTool manages a series of simulations using the model and collects the BER data.

Note To use Simulink models within BERTool, you must have a Simulink license. Also, the Communications Blockset is highly recommended. The rest of this section assumes that you have a license for both Simulink and the Communications Blockset.

To access the capabilities of BERTool related to Simulink models, open the **Monte Carlo** panel.



These topics describe how to use Simulink models in conjunction with BERTool:

For further details about confidence intervals and curve fitting for simulation data, see "Plotting Confidence Intervals" on page 4-24 and "Fitting BER Points to a Curve" on page 4-26, respectively.

Example: Using a Simulink Model with BERTool

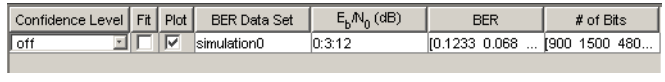
This example illustrates how BERTool can manage a series of simulations of a Simulink model, and how you can vary the plot. The model is `graycode`, one of the demonstration models included with the Communications Blockset. The example assumes that you have the Communications Blockset installed.

To run this example, follow these steps:

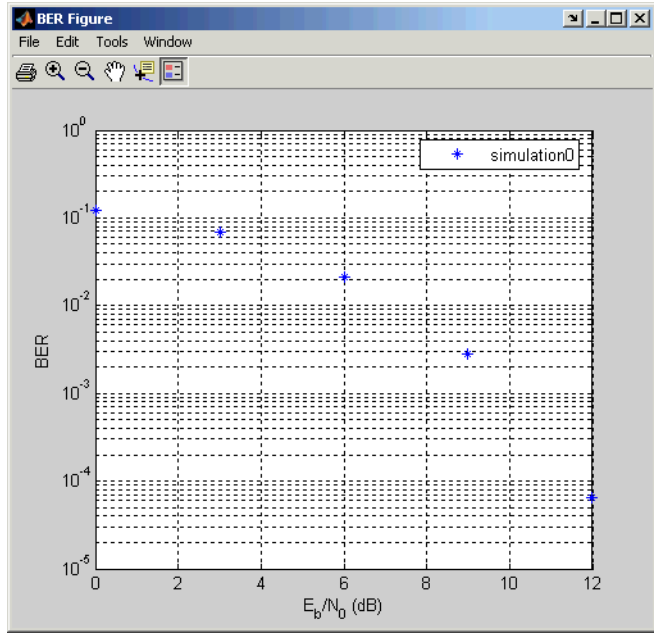
1. Open BERTool and go to the **Monte Carlo** panel. Notice that the model's filename, `graycode.mdl`, appears as the **Simulation M-file or model** parameter. (If `viterbisim.m` appears there, then check that the Communications Blockset is installed.)

2. Click **Run**.

   BERTool loads the model into memory (which in turn initializes several variables in the MATLAB workspace), runs the simulation once for each value of E_b/N_0, and gathers BER data. Then BERTool creates a listing in the data viewer.

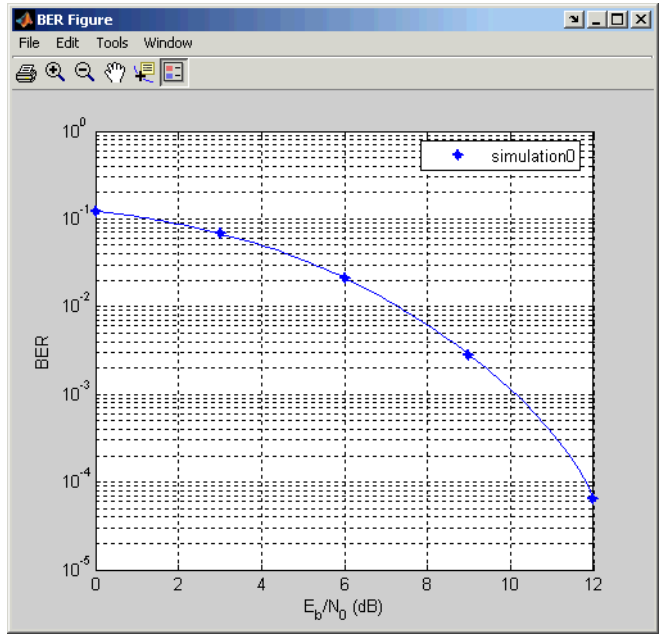


   BERTool plots the data in the **BER Figure** window.



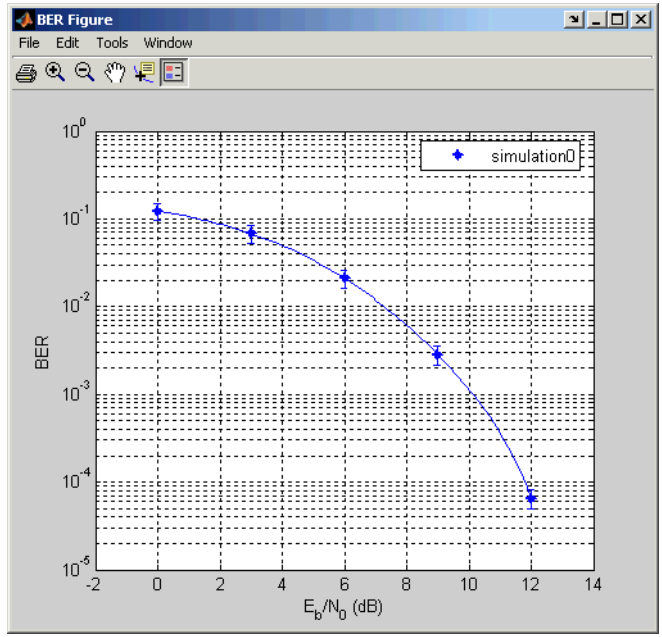
3 To fit a curve to the series of points in the **BER Figure** window, check the box next to **Fit** in the data viewer.

BERTool plots the curve, as below.



4 To indicate the 99% confidence interval around each point in the simulation data, set **Confidence Level** to 99% in the data viewer.

BERTool displays error bars to represent the confidence intervals, as below.

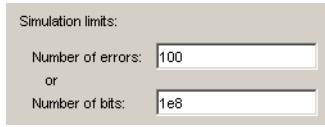


Another example that uses BERTool to manage a series of Simulink simulations is in “Example: Preparing a Model for Use with BERTool” on page 4-44.

Varying the Stopping Criteria

When you create a Simulink model for use with BERTool, you must set it up so that the simulation ends when it either detects a target number of errors or processes a maximum number of bits, whichever occurs first. To learn more about this requirement, see “Requirements for Models” on page 4-41; for an example, see “Example: Preparing a Model for Use with BERTool” on page 4-44.

After creating your Simulink model, you set the target number of errors and the maximum number of bits in the **Monte Carlo** panel of BERTool.



Typically, a **Number of errors** value of at least 100 produces an accurate error rate. The **Number of bits** value prevents the simulation from running too long, especially at large values of E_b/N_0. However, if the **Number of bits** value is so small that the simulation collects very few errors, then the error rate might not be accurate. You can use confidence intervals to gauge the accuracy of the error rates that your simulation produces; the larger the confidence interval, the less accurate the computed error rate.

You can also use the **Stop** button in BERTool to stop a series of simulations prematurely.

Preparing Simulink Models for Use with BERTool

A Simulink model that you use with BERTool must have certain properties. In many cases, it is not difficult to make new or existing models have these properties. This section lists the requirements, offers tips, and provides an example. The topics are

Requirements for Models

A Simulink model must satisfy these requirements before you can use it with BERTool, where the case-sensitive variable names must be exactly as shown below:

- The channel block must use the variable `EbNo` rather than a hard-coded value for E_b/N_0.
- The simulation must stop when the error count reaches the value of the variable `maxNumErrs` or when the number of processed bits reaches the value of the variable `maxNumBits`, whichever occurs first.

  You can configure the Error Rate Calculation block in the Communications Blockset to stop the simulation based on such criteria.
- The simulation must send the final error rate data to the MATLAB workspace as a variable whose name you enter in the **BER variable name** field in BERTool. The variable must be a three-element vector that lists the BER, the number of bit errors, and the number of processed bits.

  This three-element vector format is supported by the Error Rate Calculation block.

Tips for Preparing Models

Here are some tips for preparing a Simulink model for use with BERTool:

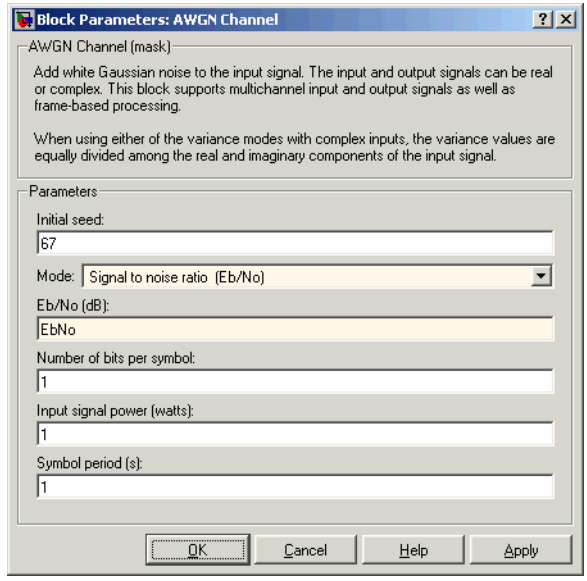
- To avoid using an undefined variable name in the dialog box for a Simulink block in the steps that follow, set up variables in the MATLAB workspace using a command such as the one below.

```
EbNo = 0; maxNumErrs = 100; maxNumBits = 1e8;
```

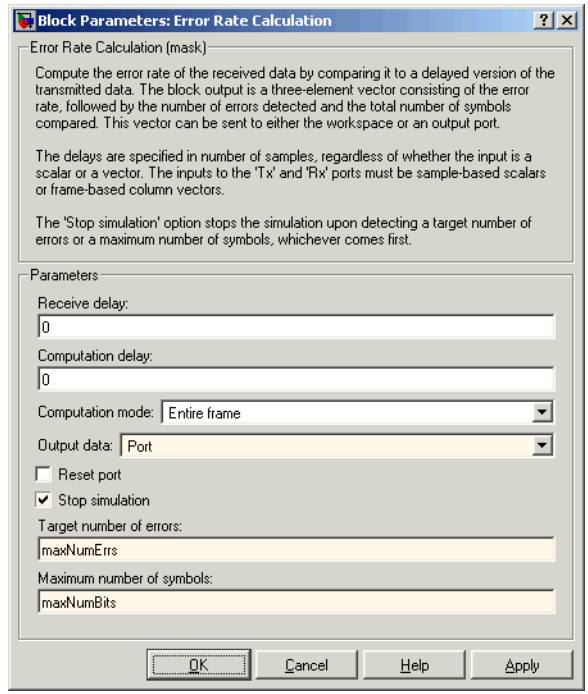
You might also want to put the same command in the model's preload function callback, to initialize the variables if you reopen the model in a future MATLAB session.

When you use BERTool, it provides the actual values based on what you enter in the GUI, so the initial values above are somewhat arbitrary.

- To model the channel, use the AWGN Channel block in the Communications Blockset with these parameters:
  - **Mode** = `Signal to noise ratio (Eb/No)`
  - **Eb/No** = `EbNo`



- To compute the error rate, use the Error Rate Calculation block in the Communications Blockset with these parameters:
  - Check **Stop simulation**.
  - **Target number of errors** = `maxNumErrs`
  - **Maximum number of symbols** = `maxNumBits`



- To send data from the Error Rate Calculation block to the MATLAB workspace, set **Output data** to `Port`, attach a Signal to Workspace block from the Signal Processing Blockset, and set the latter block's **Limit data points to last** parameter to `1`. The **Variable name** parameter in the Signal to Workspace block must match the value you enter in the **BER variable name** field of BERTool.
- If your model computes a symbol error rate instead of a bit error rate, then use the Integer to Bit Converter block in the Communications Blockset to convert symbols to bits.
- Frame-based simulations often run faster than sample-based simulations for the same number of bits processed. Note that the number of errors or number of processed bits might exceed the values that you enter in BERTool, because the simulation always processes a fixed amount of data in each frame.
- If you have an existing model that uses the AWGN Channel block using a **Mode** parameter other than `Signal to noise ratio (Eb/No)`, then you

can adapt the block to use the Eb/No mode instead. To learn about how the block's different modes are related to each other, press the AWGN Channel block's **Help** button to view the online reference page.

- If your model uses a preload function or other callback to initialize variables in the MATLAB workspace upon loading, then make sure before you use the **Run** button in BERTool that one of these conditions is met:
  - The model is not currently in memory. In this case, BERTool loads the model into memory and runs the callback functions.
  - The model is in memory (whether in a window or not), and the variables are intact.

  If you clear or overwrite the model's variables and want to restore their values before using the **Run** button in BERTool, then you can use the `bdclose` function in the MATLAB Command Window to clear the model from memory. This causes BERTool to reload the model after you press **Run**. Similarly, if you refresh your workspace by issuing a `clear all` or `clear variables` command, then you should also clear the model from memory by using `bdclose all`.

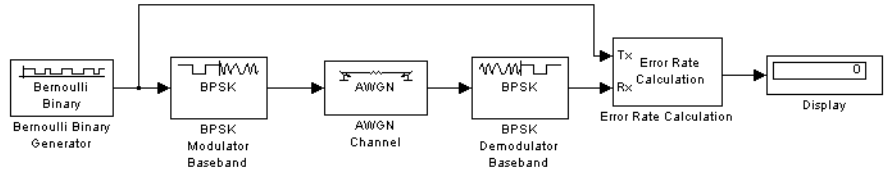
Example: Preparing a Model for Use with BERTool

This example starts from a Simulink model originally created as an example in the Communications Blockset Getting Started documentation, and shows how to tailor the model for use with BERTool. The example also illustrates how to compare the BER performance of a Simulink simulation with theoretical BER results. The example assumes that you have the Communications Blockset installed.

To prepare the model for use with BERTool, follow these steps, using the exact case-sensitive variable names as shown:

1. Open the model by entering the following command in the MATLAB Command Window.

   ```
   bpskdoc
   ```



2 To initialize parameters in the MATLAB workspace and avoid using undefined variables as block parameters, enter the following command in the MATLAB Command Window.

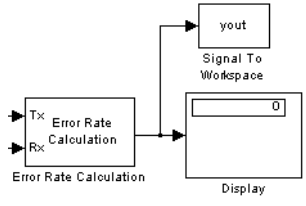
```
EbNo = 0; maxNumErrs = 100; maxNumBits = 1e8;
```

3 To ensure that BERTool uses the correct amount of noise each time it runs the simulation, open the dialog box for the AWGN Channel block by double-clicking the block. Set **Es/No** to `EbNo` and click **OK**. In this particular model, E_s/N_0 is equivalent to E_b/N_0 because the modulation type is BPSK.

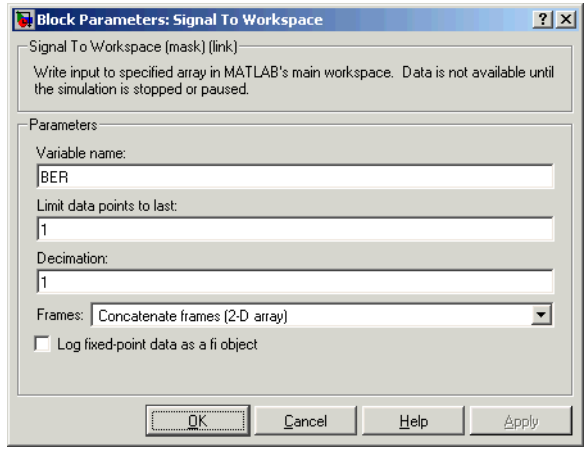
4 To ensure that BERTool uses the correct stopping criteria for each iteration, open the dialog box for the Error Rate Calculation block. Set **Target number of errors** to `maxNumErrs`, set **Maximum number of symbols** to `maxNumBits`, and click **OK**.

5 To enable BERTool to access the BER results that the Error Rate Calculation block computes, insert a Signal to Workspace block in the model and connect it to the output of the Error Rate Calculation block.

Note The Signal to Workspace block is in the Signal Processing Blockset and is different from the To Workspace block in Simulink.



6 To configure the newly added Signal to Workspace block, open its dialog box. Set **Variable name** to `BER`, set **Limit data points to last** to `1`, and click **OK**.



7 (*Optional*) To make the simulation run faster, especially at high values of E_b/N_0, open the dialog box for the Bernoulli Binary Generator block. Check **Frame-based outputs** and set **Samples per frame** to `1000`.

8 Save the model in a directory on your MATLAB path, using the filename `bertool_bpskdoc.mdl`.

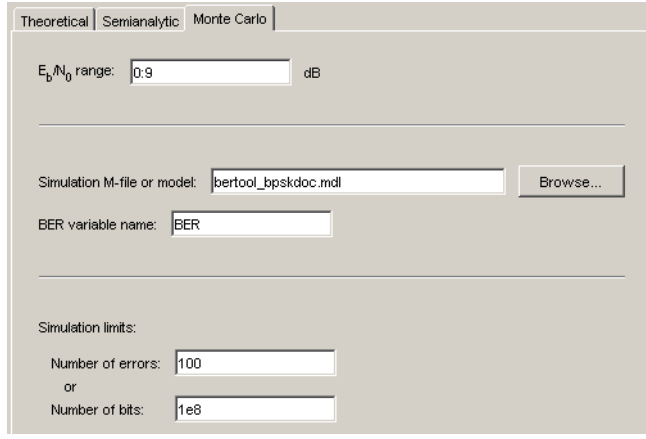
9 (*Optional*) To cause Simulink to initialize parameters if you reopen this model in a future MATLAB session, enter the following command in the MATLAB Command Window and then resave the model.

```
set_param('bertool_bpskdoc','preLoadFcn',...
   'EbNo = 0; maxNumErrs = 100; maxNumBits = 1e8;');
```

The `bertool_bpskdoc` model is now compatible with BERTool. To use it in conjunction with BERTool, continue the example by following these steps:

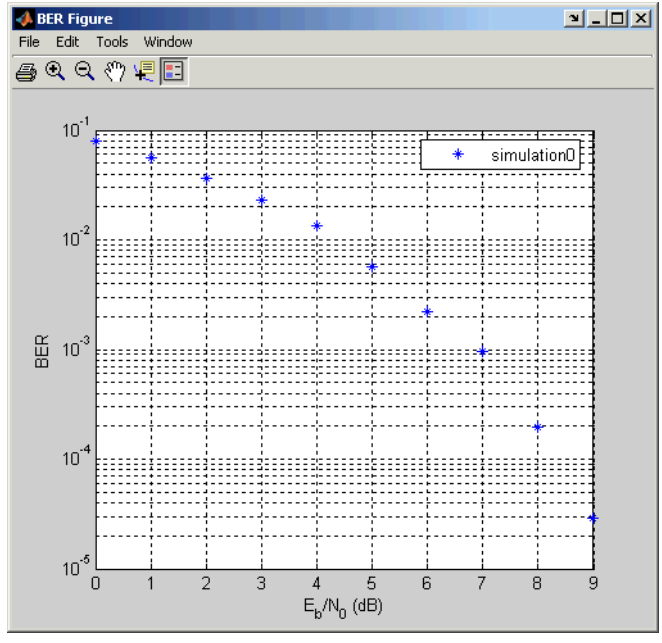
10 Open BERTool and go to the **Monte Carlo** panel.

11 Set parameters on the **Monte Carlo** panel as shown below.

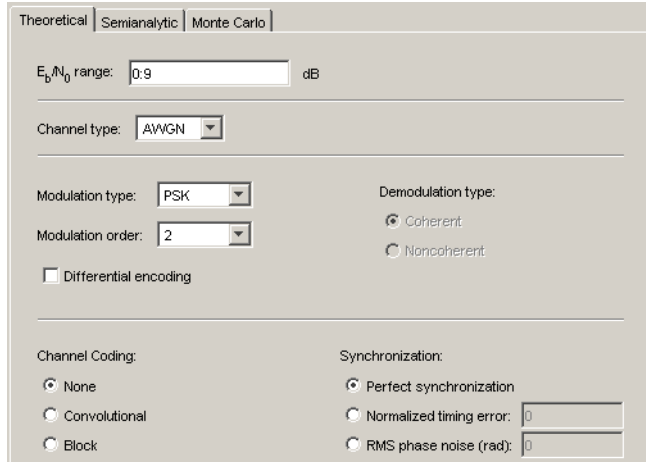


12 Click **Run**.

BERTool spends some time computing results and then plots them.

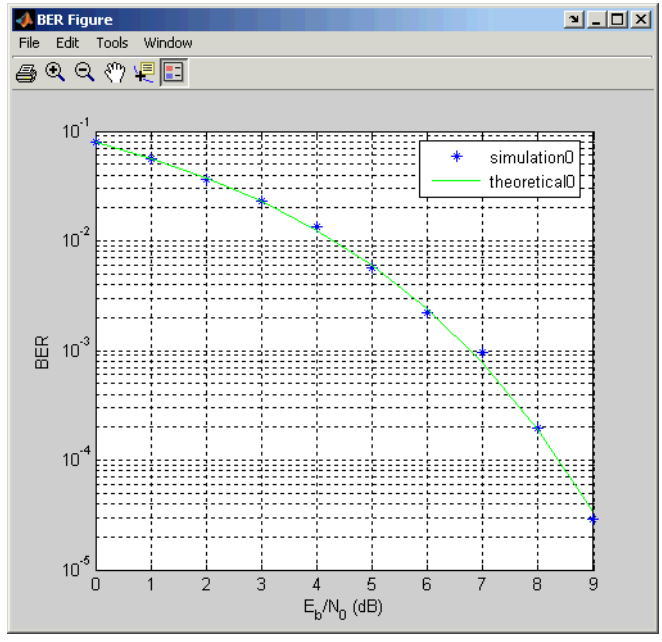


13 To compare these simulation results with theoretical results, go to the **Theoretical** panel in BERTool and set parameters as shown below.



14 Click **Plot**.

BERTool plots the theoretical curve in the **BER Figure** window along with the earlier simulation results.


BER Figure
File Edit Tools Window
simulation0
theoretical0
BER
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0 1 2 3 4 5 6 7 8 9
E_b/N_0 (dB)

Managing BER Data

After you generate BER data using BERTool, you can manage the data in various ways. This section describes how to accomplish tasks listed in the table below.

Task	Section
Manipulate data from BERTool using MATLAB commands	"Exporting Data Sets" on page 4-51 and "Examining an Exported Structure" on page 4-52
Save an individual data set from the data viewer	"Exporting Data Sets" on page 4-51
Save all data from the data viewer	"Saving a BERTool Session" on page 4-53
Load previously saved data into BERTool	"Importing Data Sets or BERTool Sessions" on page 4-54
Rename or delete sets of data in the data viewer	"Managing Data in the Data Viewer" on page 4-55
Change the sequence of columns in the data viewer	"Managing Data in the Data Viewer" on page 4-55

Exporting Data Sets or BERTool Sessions

BERTool enables you to export individual data sets to the MATLAB workspace or to MAT-files. One option for exporting is convenient for processing the data outside BERTool. For example, to create a highly customized plot using data from BERTool, export the BERTool data set to the MATLAB workspace and use any of the plotting commands in MATLAB. Another option for exporting enables you to reimport the data into BERTool later.

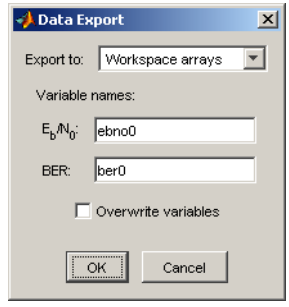
BERTool also enables you to save an entire session, which is useful if your session contains multiple data sets that you want to return to in a later session.

This section describes these capabilities, in these topics:

Exporting Data Sets

To export an individual data set, follow these steps:

1. In the data viewer, select the data set you want to export.
2. Choose **Export Data** from the **File** menu.



3. Set **Export to** to indicate the format and destination of the data.
   a. If you want to reimport the data into BERTool later, then you *must* choose either `Workspace structure` or `MAT-file structure` to create a structure in the MATLAB workspace or a MAT-file, respectively.

      A new field called **Structure name** appears. Set it to the name that you want BERTool to use for the structure it creates.

      If you selected `Workspace structure` and you want BERTool to use your chosen variable name even if a variable by that name already exists in the workspace, then check **Overwrite variables**.

   b. If you do *not* need to reimport the data into BERTool later, then a convenient way to access the data outside BERTool is to have BERTool create a pair of arrays in the MATLAB workspace. One array contains

E_b/N_0 values, while the other contains BER values. To choose this option, set **Export to** to `Workspace arrays`.

Then type two variable names in the fields under **Variable names**.

If you want BERTool to use your chosen variable names even if variables by those names already exist in the workspace, then check **Overwrite variables**.

4 Click **OK**. If you selected `MAT-file structure`, then BERTool prompts you for the path to the MAT-file that you want to create.

To reimport a structure later, see "Importing Data Sets" on page 4-54.

Examining an Exported Structure

This section briefly describes the contents of the structure that BERTool exports to the workspace or to a MAT-file. The structure's fields are indicated in the table below. The fields that are most relevant for you when you want to manipulate exported data are `paramsEvaled` and `data`.

Name of Field	Significance
`params`	The parameter values in the BERTool GUI, some of which might be invisible and hence irrelevant for computations.
`paramsEvaled`	The parameter values that BERTool uses when computing the data set.
`data`	The E_b/N_0, BER, and number of bits processed.
`dataView`	Information about the appearance in the data viewer. Used by BERTool for data reimport.
`cellEditabilities`	Indicates whether the data viewer has an active **Confidence Level** or **Fit** entry. Used by BERTool for data reimport.

Parameter Fields. The `params` and `paramsEvaled` fields are similar to each other, except that `params` describes the exact state of the GUI whereas `paramsEvaled` indicates the values that are actually used for computations. As an example of the difference, for a theoretical system with an AWGN channel, `params` records but `paramsEvaled` omits a diversity order parameter. The diversity order is not used in the computations because it is relevant only for systems with Rayleigh channels. As another example, if you type `[0:3]+1` in the GUI as the range of E_b/N_0 values, then `params` indicates `[0:3]+1` while `paramsEvaled` indicates `1 2 3 4`.

The length and exact contents of `paramsEvaled` depend on the data set because only relevant information appears. If the meaning of the contents of `paramsEvaled` is not clear upon inspection, then one way to learn more is to reimport the data set into BERTool and inspect the parameter values that appear in the GUI. To reimport the structure, follow the instructions in "Importing Data Sets or BERTool Sessions" on page 4-54.

Data Field. If your exported workspace variable is called `ber0`, then the field `ber0.data` is a cell array that contains the numerical results in these vectors:

- `ber0.data{1}` lists the E_b/N_0 values.
- `ber0.data{2}` lists the BER values corresponding to each of the E_b/N_0 values.
- `ber0.data{3}` indicates, for simulation or semianalytic results, how many bits BERTool processed when computing each of the corresponding BER values.

Saving a BERTool Session

To save an entire BERTool session, follow these steps:

1. Choose **Save Session** from the **File** menu.
2. When BERTool prompts you, enter the path to the file that you want to create.

BERTool creates a text file that records all data sets currently in the data viewer, along with the GUI parameters associated with the data sets.

Note If your BERTool session requires particular workspace variables (such as `txsig` or `rxsig` for the **Semianalytic** panel), then you should save those separately in a MAT-file using the `save` command in MATLAB.

Importing Data Sets or BERTool Sessions

BERTool enables you to reimport individual data sets that you previously exported to a structure, or to reload entire sessions that you previously saved. This section describes these capabilities, in these topics:

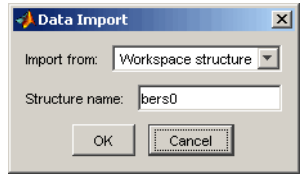
- "Importing Data Sets" on page 4-54
- "Opening a Previous BERTool Session" on page 4-55

To learn more about how to export data sets or save sessions from BERTool, see "Exporting Data Sets or BERTool Sessions" on page 4-50.

Importing Data Sets

To import an individual data set that you previously exported from BERTool to a structure, follow these steps:

1 Choose **Import Data** from the **File** menu.



2 Set **Import from** to either `Workspace structure` or `MAT-file structure`. If you selected `Workspace structure`, then type the name of the workspace variable in the **Structure name** field.

3 Click **OK**. If you selected `MAT-file`, then BERTool prompts you to select the file that contains the structure you want to import.

After you dismiss the **Data Import** dialog (and the file selection dialog, in the case of a MAT-file), the data viewer shows the newly imported data set and the **BER Figure** window plots it.

Opening a Previous BERTool Session

To replace the data sets in the data viewer with data sets from a previous BERTool session, follow these steps:

1 Choose **Open Session** from the **File** menu.

Note If BERTool already contains data sets, then it asks you whether you want to save the current session. If you answer no and continue with the loading process, then BERTool discards the current session upon opening the new session from the file.

---

2 When BERTool prompts you, enter the path to the file that you want to open. It must be a file that you previously created using the **Save Session** option in BERTool.

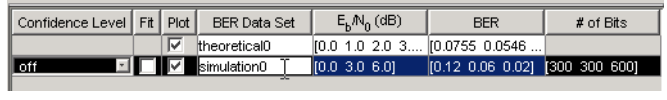
After BERTool reads the session file, the data viewer shows the data sets from the file.

If your BERTool session requires particular workspace variables (such as `txsig` or `rxsig` for the **Semianalytic** panel) that you saved separately in a MAT-file, then you can retrieve them using the `load` command in MATLAB.

Managing Data in the Data Viewer

The data viewer gives you flexibility to rename and delete data sets, and to reorder columns in the data viewer.

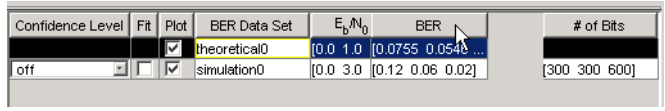
- To rename a data set in the data viewer, double-click its name in the **BER Data Set** column and type a new name.



- To delete a data set from the data viewer, select it and choose **Delete** from the **Edit** menu.

Note If the data set originated from the **Semianalytic** or **Theoretical** panel, then BERTool deletes the data without asking for confirmation. You cannot undo this operation.

- To move a column in the data viewer, drag the column's heading to the left or right with the mouse. For example, the image below shows the mouse dragging the **BER** column to the left of its default position. When you release the mouse button, the columns snap into place.



5

Source Coding

Source coding, also known as *quantization* or *signal formatting*, is a way of processing data in order to reduce redundancy or prepare it for later processing. Analog-to-digital conversion and data compression are two categories of source coding.

This chapter describes the source coding features of the Communications Toolbox, in the sections listed below.

This toolbox does not support vector quantization.

Quantizing a Signal

Scalar quantization is a process that maps all inputs within a specified range to a common value. It maps inputs in a different range of values to a different common value. In effect, scalar quantization digitizes an analog signal. Two parameters determine a quantization: a partition and a codebook. This section describes how to represent these parameters. It also shows, via examples, how to use the partition and codebook with the `quantiz` function.

Representing Partitions

A quantization partition defines several contiguous, nonoverlapping ranges of values within the set of real numbers. To specify a partition in MATLAB, list the distinct endpoints of the different ranges in a vector.

For example, if the partition separates the real number line into the four sets

- $\{x: x \leq 0\}$
- $\{x: 0< x \leq 1\}$
- $\{x: 1 < x \leq 3\}$
- $\{x: 3 < x\}$

then you can represent the partition as the three-element vector

```
partition = [0,1,3];
```

Notice that the length of the partition vector is one less than the number of partition intervals.

Representing Codebooks

A codebook tells the quantizer which common value to assign to inputs that fall into each range of the partition. Represent a codebook as a vector whose length is the same as the number of partition intervals. For example, the vector

```
codebook = [-1, 0.5, 2, 3];
```

is one possible codebook for the partition `[0,1,3]`.

Scalar Quantization Example 1

The code below shows how the `quantiz` function uses `partition` and `codebook` to map a real vector, `samp`, to a new vector, `quantized`, whose entries are either -1, 0.5, 2, or 3.

```
partition = [0,1,3];
codebook = [-1, 0.5, 2, 3];
samp = [-2.4, -1, -.2, 0, .2, 1, 1.2, 1.9, 2, 2.9, 3, 3.5, 5];
[index,quantized] = quantiz(samp,partition,codebook);
quantized
```

The output is below.

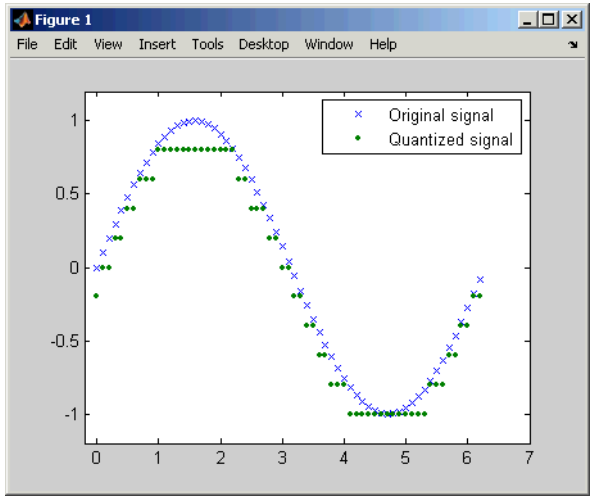
```
quantized =

  Columns 1 through 6

   -1.0000   -1.0000   -1.0000   -1.0000    0.5000    0.5000

  Columns 7 through 12

    2.0000    2.0000    2.0000    2.0000    2.0000    3.0000

  Column 13

    3.0000
```

Scalar Quantization Example 2

This example illustrates the nature of scalar quantization more clearly. After quantizing a sampled sine wave, it plots the original and quantized signals. The plot contrasts the `x`'s that make up the sine curve with the dots that make up the quantized signal. The vertical coordinate of each dot is a value in the vector `codebook`.

```
t = [0:.1:2*pi]; % Times at which to sample the sine function
sig = sin(t); % Original signal, a sine wave
partition = [-1:.2:1]; % Length 11, to represent 12 intervals
codebook = [-1.2:.2:1]; % Length 12, one entry for each interval
[index,quants] = quantiz(sig,partition,codebook); % Quantize.
plot(t,sig,'x',t,quants,'.')
```

```
legend('Original signal','Quantized signal');
axis([-.2 7 -1.2 1.2])
```



Determining Which Interval Each Input Is In

The `quantiz` function also returns a vector that tells which interval each input is in. For example, the output below says that the input entries lie within the intervals labeled 0, 6, and 5, respectively. Here, the 0th interval consists of real numbers less than or equal to 3; the 6th interval consists of real numbers greater than 8 but less than or equal to 9; and the 5th interval consists of real numbers greater than 7 but less than or equal to 8.

```
partition = [3,4,5,6,7,8,9];
index = quantiz([2 9 8],partition)
```

The output is

```
index =

     0
     6
     5
```

If you continue this example by defining a codebook vector such as

```
codebook = [3,3,4,5,6,7,8,9];
```

then the equation below relates the vector `index` to the quantized signal `quants`.

```
quants = codebook(index+1);
```

This formula for `quants` is exactly what the `quantiz` function uses if you instead phrase the example more concisely as below.

```
partition = [3,4,5,6,7,8,9];
codebook = [3,3,4,5,6,7,8,9];
[index,quants] = quantiz([2 9 8],partition,codebook);
```

Optimizing Quantization Parameters

Quantization distorts a signal. You can lessen the distortion by choosing appropriate partition and codebook parameters. However, testing and selecting parameters for large signal sets with a fine quantization scheme can be tedious. One way to produce partition and codebook parameters easily is to optimize them according to a set of so-called *training data*.

Note The training data that you use should be typical of the kinds of signals that you will actually be quantizing.

Example: Optimizing Quantization Parameters

The `lloyds` function optimizes the partition and codebook according to the Lloyd algorithm. The code below optimizes the partition and codebook for one period of a sinusoidal signal, starting from a rough initial guess. Then it uses these parameters to quantize the original signal using the initial guess parameters as well as the optimized parameters. The output shows that the mean square distortion after quantizing is much less for the optimized parameters. Notice that the `quantiz` function automatically computes the mean square distortion and returns it as the third output parameter.

```
% Start with the setup from 2nd example in "Quantizing a Signal."
t = [0:.1:2*pi];
sig = sin(t);
partition = [-1:.2:1];
codebook = [-1.2:.2:1];
% Now optimize, using codebook as an initial guess.
[partition2,codebook2] = lloyds(sig,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
[index2,quant2,distor2] = quantiz(sig,partition2,codebook2);
% Compare mean square distortions from initial and optimized
[distor, distor2] % parameters.
```

The output is

```
ans =

    0.0148    0.0024
```

Differential Pulse Code Modulation

The quantization in the section "Quantizing a Signal" on page 5-2 requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM).

The functions `dpcmenco`, `dpcmdeco`, and `dpcmopt` can help you implement a DPCM predictive quantizer with a linear predictor.

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in "Representing Partitions" on page 5-2 and "Representing Codebooks" on page 5-2, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

y(k) = p(1)x(k-1) + p(2)x(k-2) + ... + p(m-1)x(k-m+1) + p(m)x(k-m)

where x is the original signal, y(k) attempts to predict the value of x(k), and p is an m-tuple of real numbers. Instead of quantizing x itself, the DPCM encoder quantizes the *predictive error*, x-y. The integer m above is called the *predictive order*. The special case when m = 1 is called *delta modulation*.

Representing Predictors

If the guess for the `k`th value of the signal `x`, based on earlier values of `x`, is

```
y(k) = p(1)x(k-1) + p(2)x(k-2) +...+ p(m-1)x(k-m+1) + p(m)x(k-m)
```

then the corresponding predictor vector for toolbox functions is

```
predictor = [0, p(1), p(2), p(3),..., p(m-1), p(m)]
```

Note The initial zero in the predictor vector makes sense if you view the vector as the polynomial transfer function of a finite impulse response (FIR) filter.

Example: DPCM Encoding and Decoding

A simple special case of DPCM quantizes the difference between the signal's current value and its value at the previous step. Thus the predictor is just `y(k) = x(k - 1)`. The code below implements this scheme. It encodes a sawtooth signal, decodes it, and plots both the original and decoded signals. The solid line is the original signal, while the dashed line is the recovered signals. The example also computes the mean square error between the original and decoded signals.

```
predictor = [0 1]; % y(k)=x(k-1)
partition = [-1:.1:.9];
codebook = [-1:.1:1];
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Original signal
% Quantize x using DPCM.
encodedx = dpcmenco(x,codebook,partition,predictor);
% Try to recover x from the modulated signal.
decodedx = dpcmdeco(encodedx,codebook,predictor);
plot(t,x,t,decodedx,'--')
legend('Original signal','Decoded signal','Location','NorthOutside');
distor = sum((x-decodedx).^2)/length(x) % Mean square error
```

The output is

```
distor =

    0.0327
```

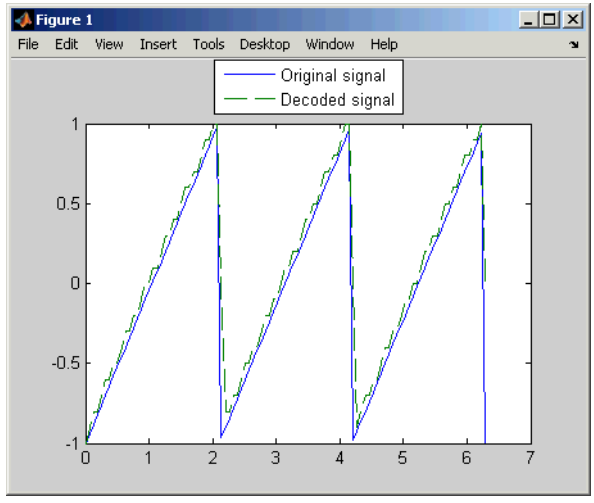


Figure 1
File Edit View Insert Tools Desktop Window Help
Original signal
Decoded signal
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7

Optimizing DPCM Parameters

The section "Optimizing Quantization Parameters" on page 5-6 describes how you can use training data with the `lloyds` function to help find quantization parameters that will minimize signal distortion. This section describes similar procedures for using the `dpcmopt` function in conjunction with the two functions `dpcmenco` and `dpcmdeco`, which first appear in the previous section.

Note The training data that you use with `dpcmopt` should be typical of the kinds of signals that you will actually be quantizing with `dpcmenco`.

Example: Comparing Optimized and Nonoptimized DPCM Parameters

This example is similar to the one in the last section. However, whereas the last example created `predictor`, `partition`, and `codebook` in a straightforward but haphazard way, this example uses the same codebook (now called `initcodebook`) as an initial guess for a new *optimized* codebook parameter. This example also uses the predictive order, 1, as the desired order of the new optimized predictor. The `dpcmopt` function creates these optimized parameters, using the sawtooth signal `x` as training data. The example goes on to quantize the training data itself; in theory, the optimized parameters are suitable for quantizing other data that is similar to `x`. Notice that the mean square distortion here is much less than the distortion in the previous example.

```
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Original signal
initcodebook = [-1:.1:1]; % Initial guess at codebook
% Optimize parameters, using initial codebook and order 1.
[predictor,codebook,partition] = dpcmopt(x,1,initcodebook);
% Quantize x using DPCM.
encodedx = dpcmenco(x,codebook,partition,predictor);
% Try to recover x from the modulated signal.
decodedx = dpcmdeco(encodedx,codebook,predictor);
distor = sum((x-decodedx).^2)/length(x) % Mean square error
```

The output is

```
distor =

    0.0063
```

Companding a Signal

In certain applications, such as speech processing, it is common to use a logarithm computation, called a *compressor*, before quantizing. The inverse operation of a compressor is called an *expander*. The combination of a compressor and expander is called a *compander*.

The `compand` function supports two kinds of companders: μ-law and A-law companders. Its reference page lists both compressor laws.

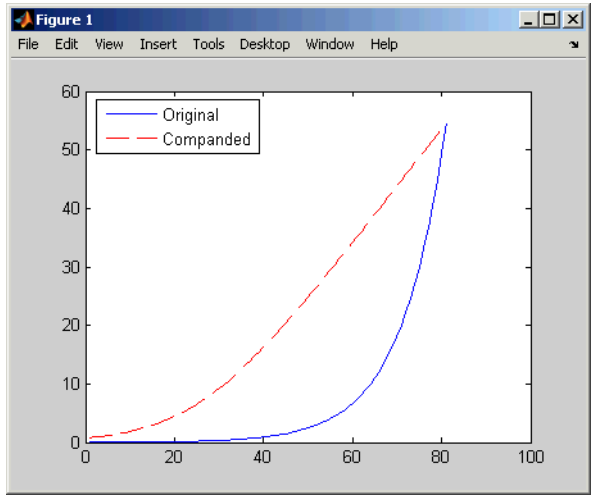
Example: A μ-Law Compander

The code below quantizes an exponential signal in two ways and compares the resulting mean square distortions. First, it uses the `quantiz` function with a partition consisting of length-one intervals. In the second trial, `compand` implements a μ-law compressor, `quantiz` quantizes the compressed data, and finally `compand` expands the quantized data. The output shows that the distortion is smaller for the second scheme. This is because equal-length intervals are well suited to the logarithm of `sig`, but not well suited to `sig`. The figure shows how the compander changes `sig`.

```
Mu = 255; % Parameter for mu-law compander
sig = -4:.1:4;
sig = exp(sig); % Exponential signal to quantize
V = max(sig);
% 1. Quantize using equal-length intervals and no compander.
[index,quants,distor] = quantiz(sig,0:floor(V),0:ceil(V));

% 2. Use same partition and codebook, but compress
% before quantizing and expand afterwards.
compsig = compand(sig,Mu,V,'mu/compressor');
[index,quants] = quantiz(compsig,0:floor(V),0:ceil(V));
newsig = compand(quants,Mu,max(quants),'mu/expander');
distor2 = sum((newsig-sig).^2)/length(sig);
[distor, distor2] % Display both mean square distortions.

plot(sig); % Plot original signal.
hold on;
plot(compsig,'r--'); % Plot companded signal.
legend('Original','Companded','Location','NorthWest')
```

The output and figure are below.

```
ans =

    0.5348    0.0397
```



Huffman Coding

Huffman coding offers a way to compress data. The average length of a Huffman code depends on the statistical frequency with which the source produces each symbol from its alphabet. A Huffman code dictionary, which associates each data symbol with a codeword, has the property that no codeword in the dictionary is a prefix of any other codeword in the dictionary.

The `huffmandict`, `huffmanenco`, and `huffmandeco` functions support Huffman coding and decoding.

Note For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding. To learn how to use arithmetic coding, see "Arithmetic Coding" on page 5-16.

Creating a Huffman Code Dictionary

Huffman coding requires statistical information about the source of the data being encoded. In particular, the `p` input argument in the `huffmandict` function lists the probability with which the source produces each symbol in its alphabet.

For example, consider a data source that produces 1s with probability 0.1, 2s with probability 0.1, and 3s with probability 0.8. The main computational step in encoding data from this source using a Huffman code is to create a dictionary that associates each data symbol with a codeword. The commands below create such a dictionary and then show the codeword vector associated with a particular value from the data source.

```
symbols = [1 2 3]; % Data symbols
p = [0.1 0.1 0.8]; % Probability of each data symbol
dict = huffmandict(symbols,p) % Create the dictionary.
dict{1,:} % Show one row of the dictionary.
```

The output below shows that the most probable data symbol, 3, is associated with a one-digit codeword, while less probable data symbols are associated with two-digit codewords. The output also shows, for example, that a Huffman encoder receiving the data symbol 1 should substitute the sequence 11.

```
dict =

    [1]    [1x2 double]
    [2]    [1x2 double]
    [3]    [         0]


ans =

     1


ans =

     1     1
```

Example: Creating and Decoding a Huffman Code

The example below performs Huffman encoding and decoding, using a source whose alphabet has three symbols. Notice that the `huffmanenco` and `huffmandeco` functions use the dictionary that `huffmandict` created.

```
sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50); % Data to encode
symbols = [1 2 3]; % Distinct data symbols appearing in sig
p = [0.1 0.1 0.8]; % Probability of each data symbol
dict = huffmandict(symbols,p); % Create the dictionary.
hcode = huffmanenco(sig,dict); % Encode the data.
dhsig = huffmandeco(hcode,dict); % Decode the code.
```

Arithmetic Coding

Arithmetic coding offers a way to compress data and can be useful for data sources having a small alphabet. The length of an arithmetic code, instead of being fixed relative to the number of symbols being encoded, depends on the statistical frequency with which the source produces each symbol from its alphabet. For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding.

The `arithenco` and `arithdeco` functions support arithmetic coding and decoding.

Representing Arithmetic Coding Parameters

Arithmetic coding requires statistical information about the source of the data being encoded. In particular, the `counts` input argument in the `arithenco` and `arithdeco` functions lists the frequency with which the source produces each symbol in its alphabet. You can determine the frequencies by studying a set of test data from the source. The set of test data can have any size you choose, as long as each symbol in the alphabet has a nonzero frequency.

For example, before encoding data from a source that produces 10 x's, 10 y's, and 80 z's in a typical 100-symbol set of test data, define

```
counts = [10 10 80];
```

Alternatively, if a larger set of test data from the source contains 22 x's, 23 y's, and 185 z's, then define

```
counts = [22 23 185];
```

Example: Creating and Decoding an Arithmetic Code

The example below performs arithmetic encoding and decoding, using a source whose alphabet has three symbols.

```
seq = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
counts = [10 10 80];
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq));
```

Selected Bibliography for Source Coding

[1] Cover, Thomas M., and Joy A. Thomas, *Elements of Information Theory*, New York, John Wiley & Sons, 1991.

[2] Kondoz, A. M., *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.

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

[4] Sklar, Bernard, *Digital Communications, Fundamentals and Applications*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

6

Error-Control Coding

Error-control coding techniques detect and possibly correct errors that occur when messages are transmitted in a digital communication system. To accomplish this, the encoder transmits not only the information symbols, but also one or more redundant symbols. The decoder uses the redundant symbols to detect and possibly correct whatever errors occurred during transmission. The sections of this chapter are as follows.

Block Coding

Block coding is a special case of error-control coding. Block coding techniques map a fixed number of message symbols to a fixed number of code symbols. A block coder treats each block of data independently and is a memoryless device.

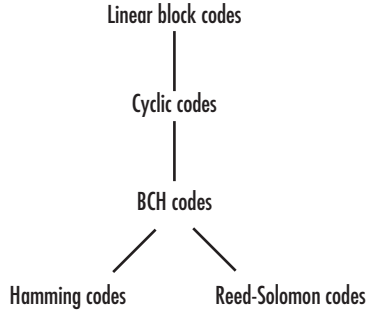
Some topics here are relevant only for specific block coding techniques, while other topics apply to all supported block coding techniques. The table below suggests which topics you should read based on the coding techniques you want to use.

Block Coding Technique	Relevant Sections
All supported block coding techniques	• "Block Coding Features of the Toolbox" on page 6-3 • "Block Coding Terminology" on page 6-4 • "Performing Other Block Code Tasks" on page 6-26 • "Selected Bibliography for Block Coding" on page 6-28
Reed-Solomon	• "Representing Words for Reed-Solomon Codes" on page 6-5 • "Parameters for Reed-Solomon Codes" on page 6-5 • "Creating and Decoding Reed-Solomon Codes" on page 6-7

Block Coding Technique	Relevant Sections
BCH	• "Representing Words for BCH Codes" on page 6-11 • "Parameters for BCH Codes" on page 6-12 • "Creating and Decoding BCH Codes" on page 6-12
Cyclic, Hamming, and generic linear block	• "Representing Words for Linear Block Codes" on page 6-15 • "Parameters for Linear Block Codes" on page 6-18 • "Creating and Decoding Linear Block Codes" on page 6-23

Block Coding Features of the Toolbox

The class of linear block coding techniques includes categories shown below.



The Communications Toolbox supports general linear block codes. It also includes functions to process cyclic, BCH, Hamming, and Reed-Solomon codes (which are all special kinds of linear block codes). Functions in the toolbox can accomplish these tasks:

- Encode or decode a message using one of the techniques mentioned above
- Determine characteristics of a technique, such as error-correction capability or valid message length
- Perform lower level computations associated with a technique, such as
  - Compute a decoding table
  - Compute a generator or parity-check matrix
  - Convert between generator and parity-check matrices
  - Compute a generator polynomial

Note The functions in this toolbox are designed for block codes that use an alphabet whose size is a power of 2.

The table below lists the functions that are related to each supported block coding technique.

Block Coding Technique	Toolbox Functions
Linear block	`encode`, `decode`, `gen2par`, `syndtable`
Cyclic	`encode`, `decode`, `cyclpoly`, `cyclgen`, `gen2par`, `syndtable`
BCH	`bchenc`, `bchdec`, `bchgenpoly`
Hamming	`encode`, `decode`, `hammgen`, `gen2par`, `syndtable`
Reed-Solomon	`rsenc`, `rsdec`, `rsgenpoly`, `rsencof`, `rsdecof`

Block Coding Terminology

Throughout this section, the information to be encoded consists of a sequence of *message* symbols and the code that is produced consists of a sequence of *codewords*.

Each block of k message symbols is encoded into a codeword that consists of n symbols; in this context, k is called the message length, n is called the codeword length, and the code is called an [n,k] code.

Representing Words for Reed-Solomon Codes

This toolbox supports Reed-Solomon codes that use m-bit symbols instead of bits. A message for an [n,k] Reed-Solomon code must be a k-column Galois array in the field GF(2^m). Each array entry must be an integer between 0 and 2^m-1. The code corresponding to that message is an n-column Galois array in GF(2^m). The codeword length n must be between 3 and 2^m-1.

Note For information about Galois arrays and how to create them, see "Representing Elements of Galois Fields" on page 12-4 or the reference page for the `gf` function.

The example below illustrates how to represent words for a [7,3] Reed-Solomon code.

```
n = 7; k = 3; % Codeword length and message length
m = 3; % Number of bits in each symbol
msg = gf([1 6 4; 0 4 3],m); % Message is a Galois array.
c = rsenc(msg,n,k) % Code will be a Galois array.
```

The output is

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

Array elements =

     1     6     4     4     3     6     3
     0     4     3     3     7     4     7
```

Parameters for Reed-Solomon Codes

This section describes several integers related to Reed-Solomon codes and discusses how to find generator polynomials.

Allowable Values of Integer Parameters

The table below summarizes the meanings and allowable values of some positive integer quantities related to Reed-Solomon codes as supported in this toolbox. The quantities `n` and `k` are input parameters for Reed-Solomon functions in this toolbox.

Symbol	Meaning	Value or Range
m	Number of bits per symbol	Integer between 3 and 16
`n`	Number of symbols per codeword	Integer between 3 and 2^m-1
`k`	Number of symbols per message	Positive integer less than `n`, such that `n-k` is even
t	Error-correction capability of the code	`(n-k)/2`

Generator Polynomial

The `rsgenpoly` function produces generator polynomials for Reed-Solomon codes. It is useful if you want to use `rsenc` and `rsdec` with a generator polynomial other than the default, or if you want to examine or manipulate a generator polynomial. `rsgenpoly` represents a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable. If each symbol has m bits, then the Galois row vector is in the field GF(2^m). For example, the command

```
r = rsgenpoly(15,13)

r = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1     6     8
```

finds that one generator polynomial for a [15,13] Reed-Solomon code is $X^2 + (A^2 + A)X + (A^3)$, where A is a root of the default primitive polynomial for GF(16).

Algebraic Expression for Generator Polynomials. The generator polynomials that `rsgenpoly` produces have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where b is an integer, A is a root of the primitive polynomial for the Galois field, and t is `(n-k)/2`. The default value of b is 1. The output from `rsgenpoly` is the result of multiplying the factors and collecting like powers of X. The example below checks this formula for the case of a [15,13] Reed-Solomon code, using b = 1.

```
n = 15;
a = gf(2,log2(n+1)); % Root of primitive polynomial
f1 = [1 a]; f2 = [1 a^2]; % Factors that form generator polynomial
f = conv(f1,f2) % Generator polynomial, same as r above.
```

Creating and Decoding Reed-Solomon Codes

The `rsenc` and `rsdec` functions create and decode Reed-Solomon codes, using the data described in “Representing Words for Reed-Solomon Codes” on page 6-5 and “Parameters for Reed-Solomon Codes” on page 6-5.

This section illustrates how to use `rsenc` and `rsdec`. The topics are

Example: Reed-Solomon Coding Syntaxes

The example below illustrates multiple ways to encode and decode data using a [15,13] Reed-Solomon code. The example shows that you can

- Vary the generator polynomial for the code, using `rsgenpoly` to produce a different generator polynomial.

- Vary the primitive polynomial for the Galois field that contains the symbols, using an input argument in `gf`.
- Vary the position of the parity symbols within the codewords, choosing either the end (default) or beginning.

The example also shows that corresponding syntaxes of `rsenc` and `rsdec` use the same input arguments, except for the first input argument.

```
m = 4; % Number of bits in each symbol
n = 2^m-1; k = 13; % Codeword length and message length
data = randint(4,k,2^m); % Four random integer messages
msg = gf(data,m); % Represent data using a Galois array.

% Simplest syntax for encoding
c1 = rsenc(msg,n,k);
d1 = rsdec(c1,n,k);

% Vary the generator polynomial for the code.
c2 = rsenc(msg,n,k,rsgenpoly(n,k,19,2));
d2 = rsdec(c2,n,k,rsgenpoly(n,k,19,2));

% Vary the primitive polynomial for GF(16).
msg2 = gf(data,m,25);
c3 = rsenc(msg2,n,k);
d3 = rsdec(c3,n,k);

% Prepend the parity symbols instead of appending them.
c4 = rsenc(msg,n,k,'beginning');
d4 = rsdec(c4,n,k,'beginning');

% Check that the decoding worked correctly.
chk = isequal(d1,msg) & isequal(d2,msg) & isequal(d3,msg2) &...
isequal(d4,msg)
```

The output is

```
chk =

     1
```

Example: Detecting and Correcting Errors in a Reed-Solomon Code

The example below illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and invokes `rsdec` to attempt to decode the noisy code. It uses additional output arguments in `rsdec` to gain information about the success of the decoding process.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Codeword length and message length
t = (n-k)/2; % Error-correction capability of the code
nw = 4; % Number of words to process
msgw = gf(randint(nw,k,2^m),m); % Random k-symbol messages
c = rsenc(msgw,n,k); % Encode the data.
noise = (1+randint(nw,n,2^m-1)).*randerr(nw,n,t); % t errors/row
cnoisy = c + noise; % Add noise to the code.
[dc,nerrs,corrcode] = rsdec(cnoisy,n,k); % Decode the noisy code.
% Check that the decoding worked correctly.
isequal(dc,msgw) & isequal(corrcode,c)
nerrs % Find out how many errors rsdec corrected.
```

Notice that the array of noise values contains integers between 1 and `2^m`, and that the addition operation `c + noise` takes place in the Galois field GF(`2^m`) because `c` is a Galois array in GF(`2^m`).

The output from the example is below. The nonzero value of `ans` indicates that the decoder was able to correct the corrupted codewords and recover the original message. The values in the vector `nerrs` indicates that the decoder corrected `t` errors in each codeword.

```
ans =

     1

nerrs =

     2
     2
     2
     2
```

Excessive Noise in Reed-Solomon Codewords

In the previous example, `rsdec` corrected all of the errors. However, each Reed-Solomon code has a finite error-correction capability. If the noise is so great that the corrupted codeword is too far in Hamming distance from the correct codeword, then either

- The corrupted codeword is close to a valid codeword *other than* the correct codeword. The decoder returns the message that corresponds to the other codeword.
- The corrupted codeword is not close enough to any codeword for successful decoding. This situation is called a *decoding failure*. The decoder removes the symbols in parity positions from the corrupted codeword and returns the remaining symbols.

In both cases, the decoder returns the wrong message. However, you can tell when a decoding failure occurs because `rsdec` also returns a value of `-1` in its second output.

To examine cases in which codewords are too noisy for successful decoding, change the previous example so that the definition of `noise` is

```
noise = (1+randint(nw,n,n)).*randerr(nw,n,t+1); % t+1 errors/row
```

Creating Shortened Reed-Solomon Codes

Every Reed-Solomon encoder uses a codeword length that equals 2^m-1 for an integer m. A shortened Reed-Solomon code is one in which the codeword length is not 2^m-1. A shortened [n,k] Reed-Solomon code implicitly uses an $[n_1,k_1]$ encoder, where

- $n_1 = 2^m - 1$, where m is the number of bits per symbol
- $k_1 = k + (n_1 - n)$

The `rsenc` and `rsdec` functions support shortened codes using the same syntaxes that they use for nonshortened codes. You do not need to indicate explicitly that you want to use a shortened code. For example, compare the two similar-looking commands below. The first creates a (nonshortened) [7,5] code. The second causes `rsenc` to create a [5,3] shortened code by implicitly using a [7,5] encoder.

```
m = 3; ordinarycode = rsenc(gf([1 1 1 1 1],m),7,5);
m = 3; shortenedcode = rsenc(gf([1 1 1],m),5,3);
```

How rsenc Creates a Shortened Code. When creating a shortened code, `rsenc` performs these steps:

- Pads each message by prepending zeros
- Encodes each padded message using a Reed-Solomon encoder having an allowable codeword length and the desired error-correction capability
- Removes the extra zeros from the nonparity symbols of each codeword

The example below illustrates this process. Note that forming a [12,8] Reed-Solomon code actually uses a [15,11] Reed-Solomon encoder. Also note that you do not have to indicate in the `rsenc` syntax that this is a shortened code or that the proper encoder to use is [15,11].

```
n = 12; k = 8; % Lengths for the shortened code
m = ceil(log2(n+1)); % Number of bits per symbol
msg = gf(randint(3,k,2^m),m); % Random array of 3 k-symbol words
code = rsenc(msg,n,k); % Create a shortened code.

% Do the shortening manually, just to show how it works.
n_pad = 2^m-1; % Codeword length in the actual encoder
k_pad = k+(n_pad-n); % Message length in the actual encoder
msg_pad=[zeros(3, n_pad-n), msg]; % Prepend zeros to each word.
code_pad = rsenc(msg_pad,n_pad,k_pad); % Encode padded words.
code_eqv = code_pad(:,n_pad-n+1:n_pad); % Remove extra zeros.
ck = isequal(code_eqv,code); % Returns true (1).
```

Representing Words for BCH Codes

A message for an [n,k] BCH code must be a k-column binary Galois array. The code that corresponds to that message is an n-column binary Galois array. Each row of these Galois arrays represents one word.

The example below illustrates how to represent words for a [15, 11] BCH code.

```
n = 15; k = 5; % Codeword length and message length
msg = gf([1 0 0 1 0; 1 0 1 1 1]); % Two messages in a Galois array
cbch = bchenc(msg,n,k) % Two codewords in a Galois array.
```

The output is

```
cbch = GF(2) array.

Array elements =

  Columns 1 through 5

     1     0     0     1     0
     1     0     1     1     1

  Columns 6 through 10

     0     0     1     1     1
     0     0     0     0     1

  Columns 11 through 15

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

Parameters for BCH Codes

BCH codes use special values of n and k:

- n, the codeword length, is an integer of the form 2^m-1 for some integer m > 2.
- k, the message length, is a positive integer less than n. However, only some positive integers less than n are valid choices for k. See the bchenc reference page for a list of some valid values of k corresponding to values of n up to 511.

Creating and Decoding BCH Codes

The bchenc and bchdec functions create and decode BCH codes, using the data described in "Representing Words for BCH Codes" on page 6-11 and "Parameters for BCH Codes" on page 6-12. This section illustrates how to use bchenc and bchdec.

The topics are

Example: BCH Coding Syntaxes

The example below illustrates how to encode and decode data using a [15, 5] Reed-Solomon code. The example shows that

- You can vary the position of the parity symbols within the codewords, choosing either the end (default) or beginning.
- Corresponding syntaxes of `bchenc` and `bchdec` use the same input arguments, except for the first input argument.

```
n = 15; k = 5; % Codeword length and message length
dat = randint(4,k); % Four random binary messages
msg = gf(dat); % Represent data using a Galois array.

% Simplest syntax for encoding
c1 = bchenc(msg,n,k);
d1 = bchdec(c1,n,k);

% Prepend the parity symbols instead of appending them.
c2 = bchenc(msg,n,k,'beginning');
d2 = bchdec(c2,n,k,'beginning');

% Check that the decoding worked correctly.
chk = isequal(d1,msg) & isequal(d2,msg)
```

The output is below.

```
chk =

     1
```

Example: Detecting and Correcting Errors in a BCH Code

The example below illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and invokes `bchdec` to attempt to decode the noisy code. It uses additional output arguments in `bchdec` to gain information about the success of the decoding process.

```
n = 15; k = 5; % Codeword length and message length
[gp,t] = bchgenpoly(n,k); % t is error-correction capability.
nw = 4; % Number of words to process
msgw = gf(randint(nw,k)); % Random k-symbol messages
c = bchenc(msgw,n,k); % Encode the data.
noise = randerr(nw,n,t); % t errors/row
cnoisy = c + noise; % Add noise to the code.
[dc,nerrs,corrcode] = bchdec(cnoisy,n,k); % Decode cnoisy.

% Check that the decoding worked correctly.
chk2 = isequal(dc,msgw) & isequal(corrcode,c)
nerrs % Find out how many errors bchdec corrected.
```

Notice that the array of noise values contains binary values, and that the addition operation `c + noise` takes place in the Galois field GF(2) because `c` is a Galois array in GF(2).

The output from the example is below. The nonzero value of `ans` indicates that the decoder was able to correct the corrupted codewords and recover the original message. The values in the vector `nerrs` indicate that the decoder corrected `t` errors in each codeword.

```
chk2 =

     1


nerrs =

     3     3     3     3
```

Excessive Noise in BCH Codewords. In the previous example, `bchdec` corrected all the errors. However, each BCH code has a finite error-correction capability. To learn more about how `bchdec` behaves when the noise is excessive, see the analogous discussion for Reed-Solomon codes in "Excessive Noise in Reed-Solomon Codewords" on page 6-10.

Representing Words for Linear Block Codes

The cyclic, Hamming, and generic linear block code functionality in this toolbox offers you multiple ways to organize bits in messages or codewords. These topics explain the available formats:

To learn how to represent words for BCH or Reed-Solomon codes, see "Representing Words for BCH Codes" on page 6-11 or "Representing Words for Reed-Solomon Codes" on page 6-5.

Binary Vector Format

Your messages and codewords can take the form of vectors containing 0s and 1s. For example, messages and codes might look like `msg` and `code` in the lines below.

```
n = 6; k = 4; % Set codeword length and message length
% for a [6,4] code.
msg = [1 0 0 1 1 0 1 0 1 0 1 1]'; % Message is a binary column.
code = encode(msg,n,k,'cyclic'); % Code will be a binary column.
msg'
code'
```

The output is below.

```
ans =

  Columns 1 through 5

     1     0     0     1     1

  Columns 6 through 10

     0     1     0     1     0

  Columns 11 through 12

     1     1


ans =

  Columns 1 through 5

     1     1     1     0     0

  Columns 6 through 10

     1     0     0     1     0

  Columns 11 through 15

     1     0     0     1     1

  Columns 16 through 18

     0     1     1
```

In this example, `msg` consists of 12 entries, which are interpreted as three 4-digit (because `k = 4`) messages. The resulting vector `code` comprises three 6-digit (because `n = 6`) codewords, which are concatenated to form a vector of length 18. The parity bits are at the beginning of each codeword.

Binary Matrix Format

You can organize coding information so as to emphasize the grouping of digits into messages and codewords. If you use this approach, then each message or codeword occupies a row in a binary matrix. The example below illustrates this approach by listing each 4-bit message on a distinct row in `msg` and each 6-bit codeword on a distinct row in `code`.

```
n = 6; k = 4; % Set codeword length and message length.
msg = [1 0 0 1; 1 0 1 0; 1 0 1 1]; % Message is a binary matrix.
code = encode(msg,n,k,'cyclic'); % Code will be a binary matrix.
msg
code
```

The output is below.

```
msg =

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


code =

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

Note In the binary matrix format, the message matrix must have `k` columns. The corresponding code matrix has `n` columns. The parity bits are at the beginning of each row.

Decimal Vector Format

Your messages and codewords can take the form of vectors containing integers. Each element of the vector gives the decimal representation of the bits in one message or one codeword.

Note If `2^n` or `2^k` is very large, then you should 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.

Note When you use the decimal vector format, `encode` expects the *leftmost* bit to be the least significant bit.

The syntax for the `encode` command must mention the decimal format explicitly, as in the example below. Notice that `/decimal` is appended to the fourth argument in the `encode` command.

```
n = 6; k = 4; % Set codeword length and message length.
msg = [9;5;13]; % Message is a decimal column vector.
% Code will be a decimal vector.
code = encode(msg,n,k,'cyclic/decimal')
```

The output is below.

```
code =

    39
    20
    54
```

Note The three examples above used cyclic coding. The formats for messages and codes are similar for Hamming and generic linear block codes.

Parameters for Linear Block Codes

This subsection describes the items that you might need in order to process [n,k] cyclic, Hamming, and generic linear block codes. The table below lists the items and the coding techniques for which they are most relevant.

Parameters Used in Block Coding Techniques

Parameter	Block Coding Technique
"Generator Matrix" on page 6-19	Generic linear block
"Parity-Check Matrix" on page 6-19	Generic linear block
"Generator Polynomial" on page 6-21	Cyclic
"Decoding Table" on page 6-22	Generic linear block, Hamming

Generator Matrix

The process of encoding a message into an [n,k] linear block code is determined by a k-by-n generator matrix G. Specifically, the 1-by-k message vector v is encoded into the 1-by-n codeword vector vG. If G has the form $[I_k\ P]$ or $[P\ I_k]$, where P is some k-by-(n-k) matrix and I_k is the k-by-k identity matrix, then G is said to be in *standard form*. (Some authors, e.g., Clark and Cain [2], use the first standard form, while others, e.g., Lin and Costello [3], use the second.) Most functions in this toolbox assume that a generator matrix is in standard form when you use it as an input argument.

Some examples of generator matrices are in the next section, "Parity-Check Matrix" on page 6-19

Parity-Check Matrix

Decoding an [n,k] linear block code requires an (n-k)-by-n parity-check matrix H. It satisfies $GH^{tr} = 0 \pmod 2$, where H^{tr} denotes the matrix transpose of H, G is the code's generator matrix, and this zero matrix is k-by-(n-k). If $G = [I_k\ P]$ then $H = [-P^{tr}\ I_{n-k}]$. Most functions in this toolbox assume that a parity-check matrix is in standard form when you use it as an input argument.

The table below summarizes the standard forms of the generator and parity-check matrices for an [n,k] binary linear block code.

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

I_k is the identity matrix of size k and the ' symbol indicates matrix transpose. (For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, -1 = 1 in the binary field.)

Examples. In the command below, `parmat` is a parity-check matrix and `genmat` is a generator matrix for a Hamming code in which [n,k] = [2^3-1, n-3] = [7,4]. Notice that `genmat` has the standard form [P I_k].

```
[parmat,genmat] = hammgen(3)
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
```

The next example finds parity-check and generator matrices for a [7,3] cyclic code. The `cyclpoly` function is mentioned below in "Generator Polynomial" on page 6-21.

```
genpoly = cyclpoly(7,3);
[parmat,genmat] = cyclgen(7,genpoly)
parmat =

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

genmat =
```

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

The example below converts a generator matrix for a [5,3] linear block code into the corresponding parity-check matrix.

```
genmat = [1 0 0 1 0; 0 1 0 1 1; 0 0 1 0 1];
parmat = gen2par(genmat)

parmat =

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

The same function `gen2par` can also convert a parity-check matrix into a generator matrix.

Generator Polynomial

Cyclic codes have algebraic properties that allow a polynomial to determine the coding process completely. This so-called *generator polynomial* is a degree-(n-k) divisor of the polynomial x^n-1. Van Lint [5] explains how a generator polynomial determines a cyclic code.

The `cyclpoly` function produces generator polynomials for cyclic codes. `cyclpoly` represents a generator polynomial using a row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable. For example, the command

```
genpoly = cyclpoly(7,3)

genpoly =

     1     0     1     1     1
```

finds that one valid generator polynomial for a [7,3] cyclic code is $1 + x^2 + x^3 + x^4$.

Decoding Table

A decoding table tells a decoder how to correct errors that might have corrupted the code during transmission. Hamming codes can correct any single-symbol error in any codeword. Other codes can correct, or partially correct, errors that corrupt more than one symbol in a given codeword.

This toolbox represents a decoding table as a matrix with `n` columns and `2^(n-k)` rows. Each row gives a correction vector for one received codeword vector. A Hamming decoding table has `n+1` rows. The `syndtable` function generates a decoding table for a given parity-check matrix.

Example: Using a Decoding Table. The script below shows how to use a Hamming decoding table to correct an error in a received message. The `hammgen` function produces the parity-check matrix, while the `syndtable` function produces the decoding table. The transpose of the parity-check matrix is multiplied on the left by the received codeword, yielding the *syndrome*. The decoding table helps determine the correction vector. The corrected codeword is the sum (modulo 2) of the correction vector and the received codeword.

```
% Use a [7,4] Hamming code.
m = 3; n = 2^m-1; k = n-m;
parmat = hammgen(m); % Produce parity-check matrix.
trt = syndtable(parmat); % Produce decoding table.
recd = [1 0 0 1 1 1 1] % Suppose this is the received vector.
syndrome = rem(recd * parmat',2);
syndrome_de = bi2de(syndrome,'left-msb'); % Convert to decimal.
disp(['Syndrome = ',num2str(syndrome_de),...
      ' (decimal), ',num2str(syndrome),' (binary)'])
corrvect = trt(1+syndrome_de,:) % Correction vector
% Now compute the corrected codeword.
correctedcode = rem(corrvect+recd,2)
```

The output is below.

```
recd =

     1     0     0     1     1     1     1

Syndrome = 3 (decimal), 0  1  1 (binary)
```

```
corrvect =

     0     0     0     0     1     0     0


correctedcode =

     1     0     0     1     0     1     1
```

Creating and Decoding Linear Block Codes

The functions for encoding and decoding cyclic, Hamming, and generic linear block codes are `encode` and `decode`. This section discusses how to use these functions to create and decode generic linear block codes, cyclic codes, and Hamming codes.

Generic Linear Block Codes

Encoding a message using a generic linear block code requires a generator matrix. If you have defined variables `msg`, `n`, `k`, and `genmat`, then either of the commands

```
code = encode(msg,n,k,'linear',genmat);
code = encode(msg,n,k,'linear/decimal',genmat);
```

encodes the information in `msg` using the [n,k] code that the generator matrix `genmat` determines. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See “Representing Words for Linear Block Codes” on page 6-15 or the reference page for `encode` for a description of the formats of `msg` and `code`.

Decoding the code requires the generator matrix and possibly a decoding table. If you have defined variables `code`, `n`, `k`, `genmat`, and possibly also `trt`, then the commands

```
newmsg = decode(code,n,k,'linear',genmat);
newmsg = decode(code,n,k,'linear/decimal',genmat);
newmsg = decode(code,n,k,'linear',genmat,trt);
newmsg = decode(code,n,k,'linear/decimal',genmat,trt);
```

decode the information in code, using the [n,k] code that the generator matrix genmat determines. decode also corrects errors according to instructions in the decoding table that trt represents.

Example: Generic Linear Block Coding. The example below encodes a message, artificially adds some noise, decodes the noisy code, and keeps track of errors that the decoder detects along the way. Because the decoding table contains only zeros, the decoder does not correct any errors.

```
n = 4; k = 2;
genmat = [[1 1; 1 0], eye(2)]; % Generator matrix
msg = [0 1; 0 0; 1 0]; % Three messages, two bits each
% Create three codewords, four bits each.
code = encode(msg,n,k,'linear',genmat);
noisycode = rem(code + randerr(3,4,[0 1;.7 .3]),2); % Add noise.
trt = zeros(2^(n-k),n);  % No correction of errors
% Decode, keeping track of all detected errors.
[newmsg,err] = decode(noisycode,n,k,'linear',genmat,trt);
err_words = find(err~=0) % Find out which words had errors.
```

The output indicates that errors occurred in the first and second words. Your results might vary because this example uses random numbers as errors.

```
err_words =

     1
     2
```

Cyclic Codes

A cyclic code is a linear block code with the property that cyclic shifts of a codeword (expressed as a series of bits) are also codewords. An alternative characterization of cyclic codes is based on its generator polynomial, as mentioned in “Generator Polynomial” on page 6-21and discussed in [5].

Encoding a message using a cyclic code requires a generator polynomial. If you have defined variables msg, n, k, and genpoly, then either of the commands

```
code = encode(msg,n,k,'cyclic',genpoly);
code = encode(msg,n,k,'cyclic/decimal',genpoly);
```

encodes the information in `msg` using the [`n`,`k`] code determined by the generator polynomial `genpoly`. `genpoly` is an optional argument for `encode`. The default generator polynomial is `cyclpoly(n,k)`. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See "Representing Words for Linear Block Codes" on page 6-15 or the reference page for `encode` for a description of the formats of `msg` and `code`.

Decoding the code requires the generator polynomial and possibly a decoding table. If you have defined variables `code`, `n`, `k`, `genpoly`, and `trt`, then the commands

```
newmsg = decode(code,n,k,'cyclic',genpoly);
newmsg = decode(code,n,k,'cyclic/decimal',genpoly);
newmsg = decode(code,n,k,'cyclic',genpoly,trt);
newmsg = decode(code,n,k,'cyclic/decimal',genpoly,trt);
```

decode the information in `code`, using the [`n`,`k`] code that the generator matrix `genmat` determines. `decode` also corrects errors according to instructions in the decoding table that `trt` represents. `genpoly` is an optional argument in the first two syntaxes above. The default generator polynomial is `cyclpoly(n,k)`.

Example. You can modify the example in the section "Generic Linear Block Codes" on page 6-23 so that it uses the cyclic coding technique, instead of the linear block code with the generator matrix `genmat`. Make the changes listed below:

- Replace the second line by

  ```
  genpoly = [1 0 1]; % generator poly is 1 + x^2
  ```

- In the fifth and ninth lines (`encode` and `decode` commands), replace `genmat` by `genpoly` and replace `'linear'` by `'cyclic'`.

Another example of encoding and decoding a cyclic code is on the reference page for `encode`.

Hamming Codes

The reference pages for `encode` and `decode` contain examples of encoding and decoding Hamming codes. Also, the section “Decoding Table” on page 6-22 illustrates error correction in a Hamming code.

Performing Other Block Code Tasks

This section describes functions that compute typical parameters associated with linear block codes, as well as functions that convert information from one format to another. The topics are

Finding a Generator Polynomial

To find a generator polynomial for a cyclic, BCH, or Reed-Solomon code, use the `cyclpoly`, `bchgenpoly`, or `rsgenpoly` function, respectively. The commands

```
genpolyCyclic = cyclpoly(15,5) % 1+X^5+X^10
genpolyBCH = bchgenpoly(15,5)  % x^10+x^8+x^5+x^4+x^2+x+1
genpolyRS = rsgenpoly(15,5)
```

find generator polynomials for block codes of different types. The output is below.

```
genpolyCyclic =

     1     0     0     0     0     1     0     0     0     0     1


genpolyBCH = GF(2) array.

Array elements =

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

```
genpolyRS = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1     4     8    10    12     9     4     2    12     2     7
```

The formats of these outputs vary:

- `cyclpoly` represents a generator polynomial using an integer row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable.
- `bchgenpoly` and `rsgenpoly` represent a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable.
- `rsgenpoly` uses coefficients in a Galois field other than the binary field GF(2). For more information on the meaning of these coefficients, see "How Integers Correspond to Galois Field Elements" on page 12-7 and "Polynomials over Galois Fields" on page 12-30.

Nonuniqueness of Generator Polynomials. Some pairs of message length and codeword length do not uniquely determine the generator polynomial. The syntaxes for functions in the example above also include options for retrieving generator polynomials that satisfy certain constraints that you specify. See the functions' reference pages for details about syntax options.

Algebraic Expression for Generator Polynomials. The generator polynomials produced by `bchgenpoly` and `rsgenpoly` have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where A is a primitive element for an appropriate Galois field, and b and t are integers. See the functions' reference pages for more information about this expression.

Finding the Error-Correction Capability

The `bchgenpoly` and `rsgenpoly` functions can return an optional second output argument that indicates the error-correction capability of a BCH or Reed-Solomon code. For example, the commands

```
[g,t] = bchgenpoly(31,16);
t
t =

     3
```

find that a [31, 16] BCH code can correct up to 3 errors in each codeword.

Finding Generator and Parity-Check Matrices

To find a parity-check and generator matrix for a Hamming code with codeword length `2^m-1`, use the `hammgen` function as below. `m` must be at least three.

```
[parmat,genmat] = hammgen(m); % Hamming
```

To find a parity-check and generator matrix for a cyclic code, use the `cyclgen` function. You must provide the codeword length and a valid generator polynomial. You can use the `cyclpoly` function to produce one possible generator polynomial after you provide the codeword length and message length. For example,

```
[parmat,genmat] = cyclgen(7,cyclpoly(7,4)); % Cyclic
```

Converting Between Parity-Check and Generator Matrices

The `gen2par` function converts a generator matrix into a parity-check matrix, and vice versa. Examples to illustrate this are on the reference page for `gen2par`.

Selected Bibliography for Block Coding

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

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

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

[4] Peterson, W. Wesley, and E. J. Weldon, Jr., *Error-correcting Codes*, 2nd ed., Cambridge, Mass., MIT Press, 1972.

[5] van Lint, J. H., *Introduction to Coding Theory*, New York, Springer-Verlag, 1982.

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

Convolutional Coding

Convolutional coding is a special case of error-control coding. Unlike a block coder, a convolutional coder is not a memoryless device. Even though a convolutional coder accepts a fixed number of message symbols and produces a fixed number of code symbols, its computations depend not only on the current set of input symbols but on some of the previous input symbols.

This section

- Outlines the convolutional coding features of the Communications Toolbox
- Defines the two supported ways to describe a convolutional encoder:
  - Polynomial description
  - Trellis description
- Describes how to encode and decode using the `convenc` and `vitdec` functions
- Gives additional examples of convolutional coding

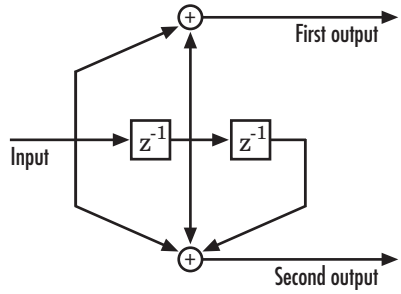
Convolutional Coding Features of the Toolbox

The Communications Toolbox supports feedforward or feedback convolutional codes that can be described by a trellis structure or a set of generator polynomials. It uses the Viterbi algorithm to implement hard-decision and soft-decision decoding.

For background information about convolutional coding, see the works listed in “Selected Bibliography for Convolutional Coding” on page 6-43.

Polynomial Description of a Convolutional Encoder

A polynomial description of a convolutional encoder describes the connections among shift registers and modulo-2 adders. For example, the figure below depicts a feedforward convolutional encoder that has one input, two outputs, and two shift registers.



A polynomial description of a convolutional encoder has either two or three components, depending on whether the encoder is a feedforward or feedback type:

- Constraint lengths
- Generator polynomials
- Feedback connection polynomials (for feedback encoders only)

Constraint Lengths

The constraint lengths of the encoder form a vector whose length is the number of inputs in the encoder diagram. The elements of this vector indicate the number of bits stored in each shift register, *including* the current input bits.

In the figure above, the constraint length is three. It is a scalar because the encoder has one input stream, and its value is one plus the number of shift registers for that input.

Generator Polynomials

If the encoder diagram has k inputs and n outputs, then the code generator matrix is a k-by-n matrix. The element in the ith row and jth column indicates how the ith input contributes to the jth output.

For *systematic* bits of a systematic feedback encoder, match the entry in the code generator matrix with the corresponding element of the feedback connection vector. See “Feedback Connection Polynomials” on page 6-32 below for details.

In other situations, you can determine the (i,j) entry in the matrix as follows:

1. Build a binary number representation by placing a 1 in each spot where a connection line from the shift register feeds into the adder, and a 0 elsewhere. The leftmost spot in the binary number represents the current input, while the rightmost spot represents the oldest input that still remains in the shift register.

2. Convert this binary representation into an octal representation by considering consecutive triplets of bits, starting from the rightmost bit. The rightmost bit in each triplet is the least significant. If the number of bits is not a multiple of three, then place zero bits at the left end as necessary. (For example, interpret 1101010 as 001 101 010 and convert it to 152.)

For example, the binary numbers corresponding to the upper and lower adders in the figure above are 110 and 111, respectively. These binary numbers are equivalent to the octal numbers 6 and 7, respectively. Thus the generator polynomial matrix is [6 7].

Note You can perform the binary-to-octal conversion in MATLAB by using code like `str2num(dec2base(bin2dec('110'),8))`.

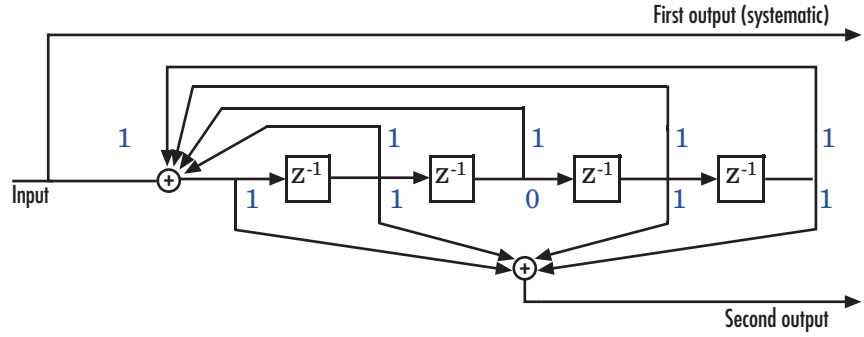
For a table of some good convolutional code generators, refer to [2] in the section "Selected Bibliography for Block Coding" on page 6-28, especially that book's appendices.

Feedback Connection Polynomials

If you are representing a feedback encoder, then you need a vector of feedback connection polynomials. The length of this vector is the number of inputs in the encoder diagram. The elements of this vector indicate the feedback connection for each input, using an octal format. First build a binary number representation as in step 1 above. Then convert the binary representation into an octal representation as in step 2 above.

If the encoder has a feedback configuration and is also systematic, then the code generator and feedback connection parameters corresponding to the systematic bits must have the same values.

For example, the diagram below shows a rate 1/2 systematic encoder with feedback.



This encoder has a constraint length of 5, a generator polynomial matrix of [37 33], and a feedback connection polynomial of 37.

The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits. The feedback polynomial is represented by the binary vector [1 1 1 1 1], corresponding to the upper row of binary digits in the diagram. These digits indicate connections from the outputs of the registers to the adder. Note that the initial 1 corresponds to the input bit. The octal representation of the binary number 11111 is 37.

The second generator polynomial is represented by the binary vector [1 1 0 1 1], corresponding to the lower row of binary digits in the diagram. The octal number corresponding to the binary number 11011 is 33.

Using the Polynomial Description in MATLAB

To use the polynomial description with the functions `convenc` and `vitdec`, first convert it into a trellis description using the `poly2trellis` function. For example, the command below computes the trellis description of the encoder pictured in the section "Polynomial Description of a Convolutional Encoder" on page 6-30.

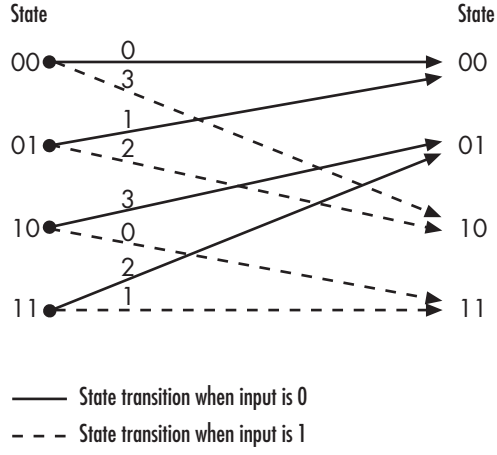
```
trellis = poly2trellis(3,[6 7]);
```

The MATLAB structure `trellis` is a suitable input argument for `convenc` and `vitdec`.

Trellis Description of a Convolutional Encoder

A trellis description of a convolutional encoder shows how each possible input to the encoder influences both the output and the state transitions of the encoder. This section describes trellises, describes how to represent trellises in MATLAB, and gives an example of a MATLAB trellis.

The figure below depicts a trellis for the convolutional encoder from the previous section. The encoder has four states (numbered in binary from 00 to 11), a one-bit input, and a two-bit output. (The ratio of input bits to output bits makes this encoder a rate-1/2 encoder.) Each solid arrow shows how the encoder changes its state if the current input is zero, and each dashed arrow shows how the encoder changes its state if the current input is one. The octal numbers above each arrow indicate the current output of the encoder.



As an example of interpreting this trellis diagram, if the encoder is in the 10 state and receives an input of zero, then it outputs the code symbol 3 and changes to the 01 state. If it is in the 10 state and receives an input of one, then it outputs the code symbol 0 and changes to the 11 state.

Note that any polynomial description of a convolutional encoder is equivalent to some trellis description, although some trellises have no corresponding polynomial descriptions.

Specifying a Trellis in MATLAB

To specify a trellis in MATLAB, use a specific form of a MATLAB structure called a trellis structure. A trellis structure must have five fields, as in the table below.

Fields of a Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputsymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in decimal) for all combinations of current state and current input

Note While your trellis structure can have any name, its fields must have the *exact* names as in the table. Field names are case sensitive.

In the nextStates matrix, each entry is an integer between 0 and numStates-1. The element in the ith row and jth column denotes the next state when the starting state is i-1 and the input bits have decimal representation j-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the nextStates matrix stores the next states when the current set of input values is {0,...,0,1}. To learn how to assign numbers to states, see the reference page for istrellis.

In the outputs matrix, the element in the ith row and jth column denotes the encoder's output when the starting state is i-1 and the input bits have decimal representation j-1. To convert to decimal value, use the first output bit as the MSB.

How to Create a MATLAB Trellis Structure

Once you know what information you want to put into each field, you can create a trellis structure in any of these ways:

- Define each of the five fields individually, using structurename.fieldname notation. For example, set the first field of a structure called s using the command below. Use additional commands to define the other fields.

  ```
  s.numInputSymbols = 2;
  ```

  The reference page for the istrellis function illustrates this approach.

- Collect all field names and their values in a single struct command. For example:

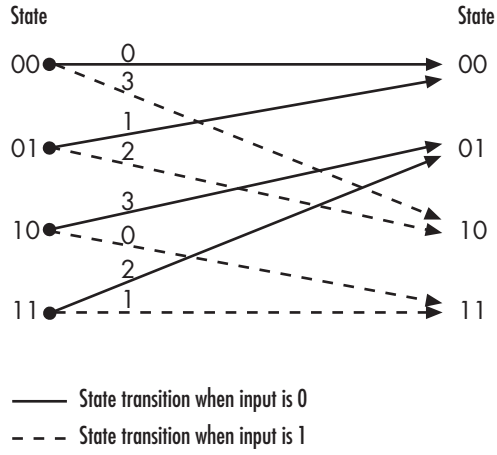
  ```
  s = struct('numInputSymbols',2,'numOutputSymbols',2,...
     'numStates',2,'nextStates',[0 1;0 1],'outputs',[0 0;1 1]);
  ```

- Start with a polynomial description of the encoder and use the poly2trellis function to convert it to a valid trellis structure. The polynomial description of a convolutional encoder is described in "Polynomial Description of a Convolutional Encoder" on page 6-30.

To check whether your structure is a valid trellis structure, use the istrellis function.

Example: A MATLAB Trellis Structure

Consider the trellis shown below.



To build a trellis structure that describes it, use the command below.

```
trellis = 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]);
```

The number of input symbols is 2 because the trellis diagram has two types of input path, the solid arrow and the dashed arrow. The number of output symbols is 4 because the numbers above the arrows can be either 0, 1, 2, or 3. The number of states is 4 because there are four bullets on the left side of the trellis diagram (equivalently, four on the right side). To compute the matrix of next states, create a matrix whose rows correspond to the four current states on the left side of the trellis, whose columns correspond to the inputs of 0 and 1, and whose elements give the next states at the end of the arrows on the right side of the trellis. To compute the matrix of outputs, create a matrix whose rows and columns are as in the next states matrix, but whose elements give the octal outputs shown above the arrows in the trellis.

Creating and Decoding Convolutional Codes

The functions for encoding and decoding convolutional codes are `convenc` and `vitdec`. This section discusses using these functions to create and decode convolutional codes.

Encoding

A simple way to use `convenc` to create a convolutional code is shown in the commands below.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Define trellis.
code = convenc(ones(100,1),t); % Encode a string of ones.
```

The first command converts a polynomial description of a feedforward convolutional encoder to the corresponding trellis description. The second command encodes 100 bits, or 50 two-bit symbols. Because the code rate in this example is 2/3, the output vector `code` contains 150 bits (that is, 100 input bits times 3/2).

To check whether your trellis corresponds to a catastrophic convolutional code, use the `iscatastrophic` function.

Hard-Decision Decoding

To decode using hard decisions, use the `vitdec` function with the flag `'hard'` and with *binary* input data. Because the output of `convenc` is binary, hard-decision decoding can use the output of `convenc` directly, without additional processing. This example extends the previous example and implements hard decision decoding.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Define trellis.
code = convenc(ones(100,1),t); % Encode a string of ones.
tb = 2; % Traceback length for decoding
decoded = vitdec(code,t,tb,'trunc','hard'); % Decode.
```

Soft-Decision Decoding

To decode using soft decisions, use the `vitdec` function with the flag `'soft'`. You must also specify the number, `nsdec`, of soft-decision bits and use input data consisting of integers between 0 and `2^nsdec-1`.

An input of 0 represents the most confident 0, while an input of `2^nsdec-1` represents the most confident 1. Other values represent less confident decisions. For example, the table below lists interpretations of values for 3-bit soft decisions.

Input Values for 3-bit Soft Decisions

Input Value	Interpretation
0	Most confident 0
1	Second most confident 0
2	Third most confident 0
3	Least confident 0
4	Least confident 1
5	Third most confident 1
6	Second most confident 1
7	Most confident 1

Example: Soft-Decision Decoding. The script below illustrates decoding with 3-bit soft decisions. First it creates a convolutional code with `convenc` and adds white Gaussian noise to the code with `awgn`. Then, to prepare for soft-decision decoding, the example uses `quantiz` to map the noisy data values to appropriate decision-value integers between 0 and 7. The second argument in `quantiz` is a partition vector that determines which data values map to 0, 1, 2, etc. The partition is chosen so that values near 0 map to 0, and values near 1 map to 7. (You can refine the partition to obtain better decoding performance if your application requires it.) Finally, the example decodes the code and computes the bit error rate. Notice that when comparing the decoded data with the original message, the example must take the decoding delay into account. The continuous operation mode of `vitdec` causes a delay equal to the traceback length, so `msg(1)` corresponds to `decoded(tblen+1)` rather than to `decoded(1)`.

```
msg = randint(4000,1,2,139); % Random data
```

```
t = poly2trellis(7,[171 133]); % Define trellis.
code = convenc(msg,t); % Encode the data.
ncode = awgn(code,6,'measured',244); % Add noise.

% Quantize to prepare for soft-decision decoding.
qcode = quantiz(ncode,[0.001,.1,.3,.5,.7,.9,.999]);

tblen = 48; delay = tblen; % Traceback length
decoded = vitdec(qcode,t,tblen,'cont','soft',3); % Decode.

% Compute bit error rate.
[number,ratio] = biterr(decoded(delay+1:end),msg(1:end-delay))
```

The output is below.

```
number =

     5


ratio =

    0.0013
```

Examples of Convolutional Coding

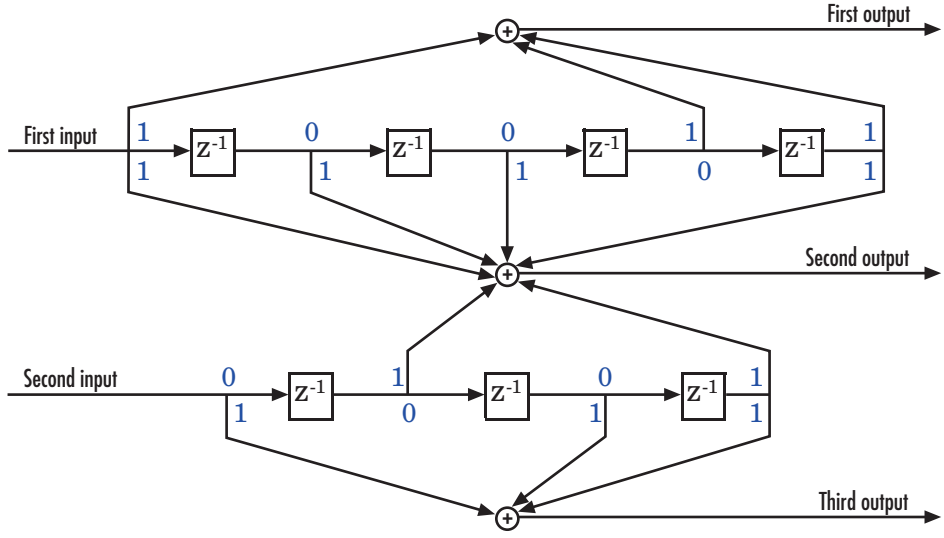
This section contains more examples of convolutional coding:

- The first example determines the correct trellis parameter for its encoder and then uses it to process a code. The decoding process uses hard decisions and the continuous operation mode. This operation mode causes a decoding delay, which the error rate computation takes into account.
- The second example processes a punctured convolutional code. The decoding process uses the unquantized decision type.

Example: A Rate-2/3 Feedforward Encoder

The example below uses the rate 2/3 feedforward encoder depicted in the schematic below. The accompanying description explains how to determine

the trellis structure parameter from a schematic of the encoder and then how to perform coding using this encoder.



Determining Coding Parameters. The `convenc` and `vitdec` functions can implement this code if their parameters have the appropriate values.

The encoder's constraint length is a vector of length 2 because the encoder has two inputs. The elements of this vector indicate the number of bits stored in each shift register, including the current input bits. Counting memory spaces in each shift register in the diagram and adding one for the current inputs leads to a constraint length of [5 4].

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, notice that the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13. The full value of the code generator matrix is [23 35 0; 0 5 13].

To use the constraint length and code generator parameters in the `convenc` and `vitdec` functions, use the `poly2trellis` function to convert those parameters into a trellis structure. The command to do this is below.

```
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Define trellis.
```

Using the Encoder. Below is a script that uses this encoder.

```
len = 1000;
msg = randint(2*len,1); % Random binary message of 2-bit symbols
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Trellis
code = convenc(msg,trel); % Encode the message.
ncode = rem(code + randerr(3*len,1,[0 1;.96 .04]),2); % Add noise.
decoded = vitdec(ncode,trel,34,'cont','hard'); % Decode.
[number,ratio] = biterr(decoded(68+1:end),msg(1:end-68));
```

Notice that `convenc` accepts a vector containing 2-bit symbols and produces a vector containing 3-bit symbols, while `vitdec` does the opposite. Also notice that `biterr` ignores the first 68 elements of `decoded`. That is, the decoding delay is 68, which is the number of bits per symbol (2) of the recovered message times the traceback depth value (34) in the `vitdec` function. The first 68 elements of `decoded` are 0s, while subsequent elements represent the decoded messages.

Example: A Punctured Convolutional Code

This example processes a punctured convolutional code. It begins by generating 3000 random bits and encoding them using a rate-1/2 convolutional encoder. The resulting vector contains 6000 bits, which are mapped to values of -1 and 1 for transmission. The puncturing process removes every third value and results in a vector of length 4000. The punctured code, `punctcode`, passes through an additive white Gaussian noise channel. Afterwards, the example inserts values to reverse the puncturing process. While the puncturing process removed both -1s and 1s from code, the insertion process inserts zeros. Then `vitdec` decodes the vector of -1s, 1s, and 0s using the `'unquant'` decision type. This unquantized decision type is appropriate here for these reasons:

- `tcode` uses -1 to represent the 1s in `code`.
- `tcode` uses 1 to represent the 0s in `code`.

- The inserted 0s are acceptable for the `'unquant'` decision type, which allows any real values as input.

Finally, the example computes the bit error rate and the number of bit errors.

```
len = 3000; msg = randint(len,1,2,94384); % Random data
t = poly2trellis(7,[171 133]); % Define trellis.
code = convenc(msg,t); % Length is 2*len.
tcode = -2*code+1; % Transmit -1s and 1s.

% Puncture by removing every third value.
punctcode = tcode;
punctcode(3:3:end)=[]; % Length is (2*len)*2/3.

ncode = awgn(punctcode,8,'measured',1234); % Add noise.

% Insert zeros.
nicode = zeros(2*len,1); % Zeros represent inserted data.
nicode(1:3:end) = ncode(1:2:end); % Write actual data.
nicode(2:3:end) = ncode(2:2:end); % Write actual data.

decoded = vitdec(nicode,t,96,'trunc','unquant'); % Decode.
[number,ratio]=biterr(decoded,msg); % Bit error rate
```

Selected Bibliography for Convolutional Coding

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

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

7

Interleaving

An interleaver permutes symbols according to a mapping. A corresponding deinterleaver uses the inverse mapping to restore the original sequence of symbols. Interleaving and deinterleaving can be useful for reducing errors caused by burst errors in a communication system. This chapter describes the interleaving features of the Communications Toolbox, in the sections listed below.

Each interleaver function in this toolbox has a corresponding deinterleaver function. In typical usage of the interleaver/deinterleaver pairs, the inputs of the deinterleaver match those of the interleaver, except for the data being rearranged.

Block Interleavers

A block interleaver accepts a set of symbols and rearranges them, without repeating or omitting any of the symbols in the set. The number of symbols in each set is fixed for a given interleaver.

Block Interleaving Features of the Toolbox

The set of block interleavers in this toolbox includes a general block interleaver as well as several special cases. Each special-case interleaver function uses the same computational code that the general block interleaver function uses, but provides a syntax that is more suitable for the special case. The interleaver functions are described below.

Type of Interleaver	Interleaver Function	Description
General block interleaver	`intrlv`	Uses the permutation table given explicitly as an input argument.
Algebraic interleaver	`algintrlv`	Derives a permutation table algebraically, using the Takeshita-Costello or Welch-Costas method. These methods are described in [4].
Helical scan interleaver	`helscanintrlv`	Fills a matrix with data row by row and then sends the matrix contents to the output in a helical fashion.
Matrix interleaver	`matintrlv`	Fills a matrix with data elements row by row and then sends the matrix contents to the output column by column.
Random interleaver	`randintrlv`	Chooses a permutation table randomly using the initial state input that you provide.

Example: Block Interleavers

The following example illustrates how an interleaver improves the error rate in a communication system whose channel produces a burst of errors. A random interleaver rearranges the bits of numerous codewords before two adjacent codewords are each corrupted by three errors.

Three errors exceed the error-correction capability of the Hamming code. However, the example shows that when the Hamming code is combined with an interleaver, this system is able to recover the original message despite the 6-bit burst of errors. The improvement in performance occurs because the interleaving effectively spreads the errors among different codewords so that the number of errors per codeword is within the error-correction capability of the code.

```
st1 = 27221; st2 = 4831; % States for random number generator
n = 7; k = 4; % Parameters for Hamming code
msg = randint(k*500,1,2,st1); % Data to encode
code = encode(msg,n,k,'hamming/binary'); % Encoded data
% Create a burst error that will corrupt two adjacent codewords.
errors = zeros(size(code)); errors(n-2:n+3) = [1 1 1 1 1 1];

% With Interleaving
%------------------
inter = randintrlv(code,st2); % Interleave.
inter_err = bitxor(inter,errors); % Include burst error.
deinter = randdeintrlv(inter_err,st2); % Deinterleave.
decoded = decode(deinter,n,k,'hamming/binary'); % Decode.
disp('Number of errors and error rate, with interleaving:');
[number_with,rate_with] = biterr(msg,decoded) % Error statistics

% Without Interleaving
%---------------------
code_err = bitxor(code,errors); % Include burst error.
decoded = decode(code_err,n,k,'hamming/binary'); % Decode.
disp('Number of errors and error rate, without interleaving:');
[number_without,rate_without] = biterr(msg,decoded) % Error statistics
```

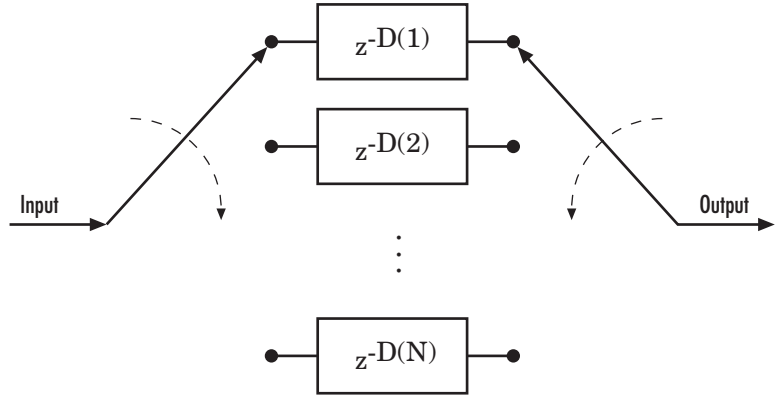
The output from the example is below.

```
Number of errors and error rate, with interleaving:

number_with =

     0


rate_with =

     0

Number of errors and error rate, without interleaving:

number_without =

     4


rate_without =

    0.0020
```

Convolutional Interleavers

A convolutional interleaver consists of a set of shift registers, each with a fixed delay. In a typical convolutional interleaver, the delays are nonnegative integer multiples of a fixed integer (although a general multiplexed interleaver allows unrestricted delay values). Each new symbol from an input vector feeds into the next shift register and the oldest symbol in that register becomes part of the output vector. A convolutional interleaver has memory; that is, its operation depends not only on current symbols but also on previous symbols.

The schematic below depicts the structure of a general convolutional interleaver by showing the set of shift registers and their delay values D(1), D(2),..., D(N). The kth shift register holds D(k) symbols, where k = 1,2,...,N. The convolutional interleaving functions in this toolbox have input arguments that indicate the number of shift registers and the delay for each shift register.



This section discusses

- The types of convolutional interleavers included in the toolbox
- An example that uses a convolutional interleaver
- The delay between the original sequence and the restored sequence

Convolutional Interleaving Features of the Toolbox

The set of convolutional interleavers in this toolbox includes a general interleaver/deinterleaver pair as well as several special cases. Each

special-case function uses the same computational code that its more general counterpart uses, but provides a syntax that is more suitable for the special case. The special cases are described below.

Type of Interleaver	Interleaving Function	Description
General multiplexed interleaver	`muxintrlv`	Allows unrestricted delay values for the set of shift registers.
Convolutional interleaver	`convintrlv`	The delay values for the set of shift registers are nonnegative integer multiples of a fixed integer that you specify.
Helical interleaver	`helintrlv`	Fills an array with input symbols in a helical fashion and empties the array row by row.

The `helscanintrlv` function and the `helintrlv` function both use a helical array for internal computations. However, the two functions have some important differences:

- `helintrlv` uses an unlimited-row array, arranges input symbols in the array along columns, outputs some symbols that are not from the current input, and leaves some input symbols in the array without placing them in the output.
- `helscanintrlv` uses a fixed-size matrix, arranges input symbols in the array across rows, and outputs all the input symbols without using any default values or values from a previous call.

Example: Convolutional Interleavers

The example below illustrates convolutional interleaving and deinterleaving using a sequence of consecutive integers. It also illustrates the inherent delay of the interleaver/deinterleaver pair.

```
x = [1:10]'; % Original data
delay = [0 1 2]; % Set delays of three shift registers.
[y,state_y] = muxintrlv(x,delay) % Interleave.
z = muxdeintrlv(y,delay) % Deinterleave.
```

In this example, the `muxintrlv` function initializes the three shift registers to the values `[]`, `[0]`, and `[0 0]`, respectively. Then the function processes the input data `[1:10]'`, performing internal computations as indicated in the table below.

Current Input	Current Shift Register	Current Output	Contents of Shift Registers
1	1	1	[] [0] [0 0]
2	2	0	[] [2] [0 0]
3	3	0	[] [2] [0 3]
4	1	4	[] [2] [0 3]
5	2	2	[] [5] [0 3]
6	3	0	[] [5] [3 6]

Current Input	Current Shift Register	Current Output	Contents of Shift Registers
7	1	7	[] [5] [3 6]
8	2	5	[] [8] [3 6]
9	3	3	[] [8] [6 9]
10	1	10	[] [8] [6 9]

The output from the example is below.

```
y =

     1
     0
     0
     4
     2
     0
     7
     5
     3
    10


state_y =

    value: {3x1 cell}
```

```
    index: 2


z =

     0
     0
     0
     0
     0
     0
     1
     2
     3
     4
```

Notice that the "Current Output" column of the table above agrees with the values in the vector `y`. Also, the last row of the table above indicates that the last shift register processed for the given data set is the first shift register. This agrees with the value of `2` for `state_y.index`, which indicates that any additional input data would be directed to the second shift register. You can optionally check that the state values listed in `state_y.value` match the "Contents of Shift Registers" entry in the last row of the table by typing `state_y.value{:}` in the Command Window after executing the example.

Another feature to notice about the example output is that `z` contains six zeros at the beginning before containing any of the symbols from the original data set. The six zeros illustrate that the delay of this convolutional interleaver/deinterleaver pair is `length(delay)*max(delay) = 3*2 = 6`. For more information about delays, see "Delays of Convolutional Interleavers" on page 7-9.

Delays of Convolutional Interleavers

After a sequence of symbols passes through a convolutional interleaver and a corresponding convolutional deinterleaver, the restored sequence lags behind the original sequence. The delay, measured in symbols, between the original and restored sequences is indicated in the table below. The variable names in the second column (`delay`, `nrows`, `slope`, `col`, `ngrp`, and `stp`) refer to the inputs named on each function's reference page.

Delays of Interleaver/Deinterleaver Pairs

Interleaver/Deinterleaver Pair	Delay Between Original and Restored Sequences
`muxintrlv`, `muxdeintrlv`	`length(delay)*max(delay)`
`convintrlv`, `convdeintrlv`	`nrows*(nrows-1)*slope`
`helintrlv`, `heldeintrlv`	`col*ngrp*ceil(stp*(col-1)/ngrp)`

Effect of Delays on Recovery of Convolutionally Interleaved Data

If you use a convolutional interleaver followed by a corresponding convolutional deinterleaver, then a nonzero delay means that the recovered data (that is, the output from the deinterleaver) is not the same as the original data (that is, the input to the interleaver). If you compare the two data sets directly, then you must take the delay into account by using appropriate truncating or padding operations.

Here are some typical ways to compensate for a delay of D in an interleaver/deinterleaver pair:

- Interleave a version of the original data that is padded with D extra symbols at the end. Before comparing the original data with the recovered data, omit the first D symbols of the recovered data. In this approach, all the original symbols appear in the recovered data.
- Before comparing the original data with the recovered data, omit the last D symbols of the original data and the first D symbols of the recovered data. In this approach, some of the original symbols are left in the deinterleaver's shift registers and do not appear in the recovered data.

The code below illustrates these approaches by computing a symbol error rate for the interleaving/deinterleaving operation.

```
x = randint(20,1,64); % Original data
nrows = 3; slope = 2; % Interleaver parameters
D = nrows*(nrows-1)*slope; % Delay of interleaver/deinterleaver pair

% First approach.
x_padded = [x; zeros(D,1)]; % Pad x at the end before interleaving.
a1 = convintrlv(x_padded,nrows,slope); % Interleave padded data.
b1 = convdeintrlv(a1,nrows,slope)
b1_trunc = b1(D+1:end); % Remove first D symbols.
ser1 = symerr(x,b1_trunc) % Compare original data with truncation.

% Second approach.
a2 = convintrlv(x,nrows,slope); % Interleave original data.
b2 = convdeintrlv(a2,nrows,slope)
x_trunc = x(1:end-D); % Remove last D symbols.
b2_trunc = b2(D+1:end); % Remove first D symbols.
ser2 = symerr(x_trunc,b2_trunc) % Compare the two truncations.
```

The output is shown below. The zero values of `ser1` and `ser2` indicate that the script correctly aligned the original and recovered data before computing the symbol error rates. However, notice from the lengths of `b1` and `b2` that the two approaches to alignment result in different amounts of deinterleaved data.

```
b1 =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
    59
    42
     1
```

```
    28
    52
    54
    43
     8
    56
     5
    35
    37
    48
    17
    28
    62
    10
    31
    61
    39


ser1 =

     0


b2 =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
    59
    42
```

```
     1
    28
    52
    54
    43
     8


ser2 =

     0
```

Combining Interleaving Delays and Other Delays

If you use convolutional interleavers in a script that incurs an additional delay, d, between the interleaver output and the deinterleaver input (for example, a delay from a filter), then the restored sequence lags behind the original sequence by the sum of d and the amount from the table Delays of Interleaver/Deinterleaver Pairs. In this case, d must be an integer multiple of the number of shift registers, or else the convolutional deinterleaver cannot recover the original symbols properly. If d is not naturally an integer multiple of the number of shift registers, then you can adjust the delay manually by padding the vector that forms the input to the deinterleaver.

Selected Bibliography for Interleaving

[1] Berlekamp, E. R. and P. Tong, "Improved Interleavers for Algebraic Block Codes," U. S. Patent 4559625, Dec. 17, 1985.

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

[3] Forney, G. D. Jr., "Burst-Correcting Codes for the Classic Bursty Channel," *IEEE Transactions on Communications*, vol. COM-19, October 1971, pp. 772-781.

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

[5] Ramsey, J. L, "Realization of Optimum Interleavers," *IEEE Transactions on Information Theory*, IT-16 (3), May 1970, pp. 338-345.

[6] 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. pp. 419.

8

Modulation

In most media for communication, only a fixed range of frequencies is available for transmission. One way to communicate a message signal whose frequency spectrum does not fall within that fixed frequency range, or one that is otherwise unsuitable for the channel, is to alter a transmittable signal according to the information in your message signal. This alteration is called *modulation*, and it is the modulated signal that you transmit. The receiver then recovers the original signal through a process called *demodulation*.

The sections of this chapter are as follows.

Modulation Features of the Toolbox

The available methods of modulation depend on whether the input signal is analog or digital. The tables below show the modulation techniques that the Communications Toolbox supports for analog and digital signals, respectively.

Analog Modulation Method	Acronym
Amplitude modulation (suppressed or transmitted carrier)	AM
Frequency modulation	FM
Phase modulation	PM
Single sideband amplitude modulation	SSB

Digital Modulation Method	Acronym
Differential phase shift keying modulation	DPSK
Frequency shift keying modulation	FSK
Minimum shift keying modulation	MSK
Offset quadrature phase shift keying modulation	OQPSK
Phase shift keying modulation	PSK
Pulse amplitude modulation	PAM
Quadrature amplitude modulation	QAM

Baseband Versus Passband Simulation

For a given modulation technique, two ways to simulate modulation techniques are called *baseband* and *passband*. Baseband simulation, also known as the *lowpass equivalent method*, requires less computation. This toolbox supports baseband simulation for digital modulation and passband simulation for analog modulation.

Modulation Terminology

Modulation is a process by which a *carrier signal* is altered according to information in a *message signal*. The *carrier frequency*, denoted `Fc`, is the frequency of the carrier signal. The *sampling rate* is the rate at which the message signal is sampled during the simulation.

The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. The Nyquist sampling theorem requires that the simulation sampling rate `Fs` be greater than two times the sum of the carrier frequency and the highest frequency of the modulated signal, in order for the demodulator to recover the message correctly.

Analog Modulation

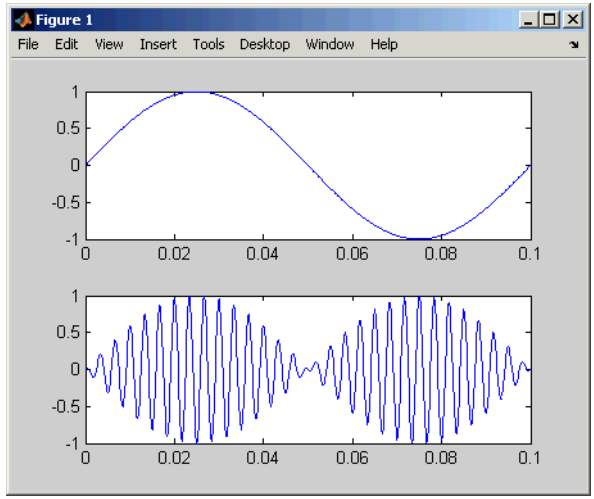
This section describes how to represent analog signals using vectors or matrices. It provides examples of using the analog modulation and demodulation functions.

Representing Analog Signals

To modulate an analog signal using this toolbox, start with a real message signal and a sampling rate `Fs` in hertz. Represent the signal using a vector `x`, the entries of which give the signal's values in time increments of 1/`Fs`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

For example, if `t` measures time in seconds, then the vector `x` below is the result of sampling a sine wave 8000 times per second for 0.1 seconds. The vector `y` represents the modulated signal.

```
Fs = 8000; % Sampling rate is 8000 samples per second.
Fc = 300; % Carrier frequency in Hz
t = [0:.1*Fs]'/Fs; % Sampling times for .1 second
x = sin(20*pi*t); % Representation of the signal
y = ammod(x,Fc,Fs); % Modulate x to produce y.
figure;
subplot(2,1,1); plot(t,x); % Plot x on top.
subplot(2,1,2); plot(t,y)% Plot y below.
```



As a multichannel example, the code below defines a two-channel signal in which one channel is a sinusoid with zero initial phase and the second channel is a sinusoid with an initial phase of `pi/8`.

```
Fs = 8000;
t = [0:.1*Fs]'/Fs;
x = [sin(20*pi*t), sin(20*pi*t+pi/8)];
```

Analog Modulation Example

This example illustrates the basic format of the analog modulation and demodulation functions. Although the example uses phase modulation, most elements of this example apply to other analog modulation techniques as well.

The example samples an analog signal and modulates it. Then it simulates an additive white Gaussian noise (AWGN) channel, demodulates the received signal, and plots the original and demodulated signals.

```
% Prepare to sample a signal for two seconds,
% at a rate of 100 samples per second.
Fs = 100; % Sampling rate
t = [0:2*Fs+1]'/Fs; % Time points for sampling

% Create the signal, a sum of sinusoids.
```

```
x = sin(2*pi*t) + sin(4*pi*t);

Fc = 10; % Carrier frequency in modulation
phasedev = pi/2; % Phase deviation for phase modulation

y = pmmod(x,Fc,Fs,phasedev); % Modulate.
y = awgn(y,10,'measured',103); % Add noise.
z = pmdemod(y,Fc,Fs,phasedev); % Demodulate.

% Plot the original and recovered signals.
figure; plot(t,x,'k-',t,z,'g-');
legend('Original signal','Recovered signal');
```

Other examples using analog modulation functions appear in the online reference pages for `ammod`, `amdemod`, `ssbdemod`, and `fmmod`.

Digital Modulation

Like analog modulation, digital modulation alters a transmittable signal according to the information in a message signal. However, in this case, the message signal is restricted to a finite set. Using this toolbox, you can modulate or demodulate signals using various digital modulation techniques, listed in “Modulation Features of the Toolbox” on page 8-2. You can also plot signal constellations.

Note The modulation and demodulation functions do not perform pulse shaping or filtering. See Chapter 9, “Special Filters” or “Combining Pulse Shaping and Filtering with Modulation” on page 8-10 for more information about filtering.

The topics in this section are as follows:

Representing Digital Signals

To modulate a signal using digital modulation with an alphabet having M symbols, start with a real message signal whose values are integers between 0 and M. Represent the signal by listing its values in a vector, `x`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

For example, if the modulation uses an alphabet with 8 symbols, then the vector `[2 3 7 1 0 5 5 2 6]'` is a valid single-channel input to the modulator. As a multichannel example, the two-column matrix

```
[2 3;
 3 3;
 7 3;
 0 3;]
```

defines a two-channel signal in which the second channel has a constant value of 3.

Baseband Modulated Signals Defined

If you use baseband modulation to produce the complex envelope y of the modulation of a message signal x, then y is a *complex-valued* signal that is related to the output of a passband modulator. If the modulated signal has the waveform

$$Y_1(t)\cos(2\pi f_c t + \theta) - Y_2(t)\sin(2\pi f_c t + \theta)$$

where f_c is the carrier frequency and θ is the carrier signal's initial phase, then a baseband simulation recognizes that this equals the real part of

$$[(Y_1(t) + jY_2(t))e^{j\theta}]\exp(j2\pi f_c t)$$

and models only the part inside the square brackets. Here j is the square root of -1. The complex vector y is a sampling of the complex signal

$$(Y_1(t) + jY_2(t))e^{j\theta}$$

If you prefer to work with passband signals instead of baseband signals, then you can build functions that convert between the two. Be aware that passband modulation tends to be more computationally intensive than baseband modulation because the carrier signal typically needs to be sampled at a high rate.

Examples of Digital Modulation and Demodulation

This section contains examples that illustrate how to use the digital modulation and demodulation functions.

Computing the Symbol Error Rate

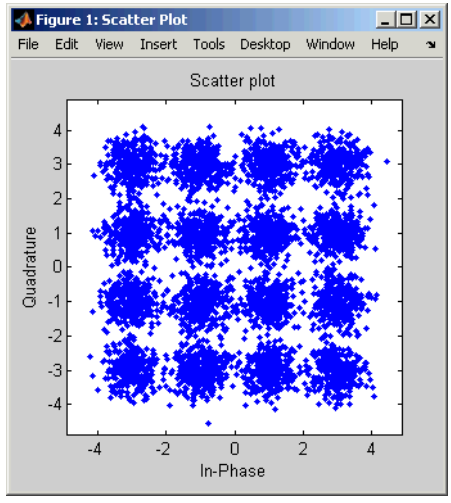
The example generates a random digital signal, modulates it, and adds noise. Then it creates a scatter plot, demodulates the noisy signal, and computes

the symbol error rate. For a more elaborate example that is similar to this one, see “Modulating a Random Signal” on page 1-4.

```
% Create a random digital message
M = 16; % Alphabet size
x = randint(5000,1,M); % Message signal

% Use 16-QAM modulation.
y = qammod(x,M);

% Transmit signal through an AWGN channel.
ynoisy = awgn(y,15,'measured');

% Create scatter plot from noisy data.
scatterplot(ynoisy);

% Demodulate to recover the message.
z = qamdemod(ynoisy,M);

% Check symbol error rate.
[num,rt]= symerr(x,z)
```

The output and scatter plot are below. Your numerical results and plot might vary, because the example uses random numbers.

```
num =

    83


rt =

    0.0166
```



Notice that the scatter plot does not look exactly like a signal constellation. Whereas the signal constellation would have 16 precisely located points, the noise causes the scatter plot to have a small cluster of points approximately where each constellation point would be.

Combining Pulse Shaping and Filtering with Modulation

Modulation is often followed by pulse shaping, and demodulation is often preceded by a filtering or an integrate-and-dump operation. This section presents an example involving rectangular pulse shaping. For an example that uses raised cosine pulse shaping, see “Pulse Shaping Using a Raised Cosine Filter” on page 1-14.

Rectangular Pulse Shaping. Rectangular pulse shaping repeats each output from the modulator a fixed number of times to create an upsampled signal. Rectangular pulse shaping can be a first step or an exploratory step in algorithm development, though it is less realistic than other kinds of pulse shaping. If the transmitter upsamples the modulated signal, then the receiver should downsample the received signal before demodulating. The “integrate and dump” operation is one way to downsample the received signal.

The code below uses the `rectpulse` function for rectangular pulse shaping at the transmitter and the `intdump` function for downsampling at the receiver.

```
M = 16; % Alphabet size
x = randint(5000,1,M); % Message signal
Nsamp = 4; % Oversampling rate

% Use 16-QAM modulation.
y = qammod(x,M);

% Follow with rectangular pulse shaping.
ypulse = rectpulse(y,Nsamp);

% Transmit signal through an AWGN channel.
ynoisy = awgn(ypulse,15,'measured');

% Downsample at the receiver.
ydownsamp = intdump(ynoisy,Nsamp);

% Demodulate to recover the message.
z = qamdemod(ydownsamp,M);
```

Plotting Signal Constellations

To plot the signal constellation associated with a modulation process, follow these steps:

1. If the alphabet size for the modulation process is M, then create the signal `[0:M-1]`. This signal represents all possible inputs to the modulator.
2. Use the appropriate modulation function to modulate this signal. If desired, scale the output. The result is the set of all points of the signal constellation.
3. Apply the `scatterplot` function to the modulated output to create a plot.

Examples of Signal Constellation Plots

The following examples produce plots of signal constellations:

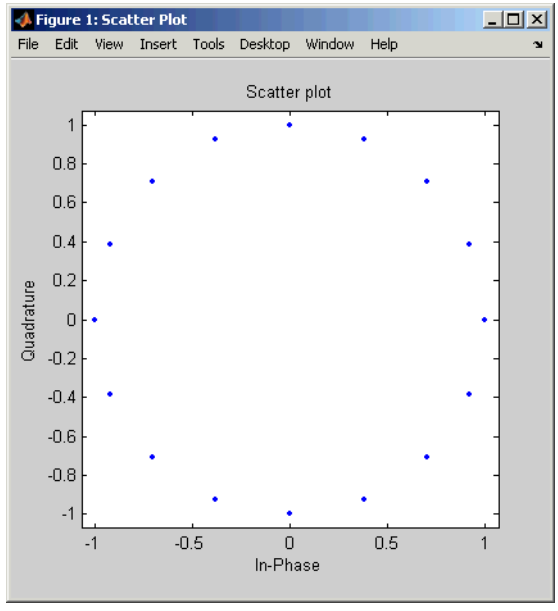
- "Constellation for 16-PSK" on page 8-12
- "Constellation for 32-QAM" on page 8-12
- "Gray-Coded Signal Constellation" on page 8-13

- "Customized Constellation for QAM" on page 8-14

The reference entries for the `modnorm` and `genqammod` functions provide additional examples.

Constellation for 16-PSK. The code below plots a PSK constellation having 16 points.

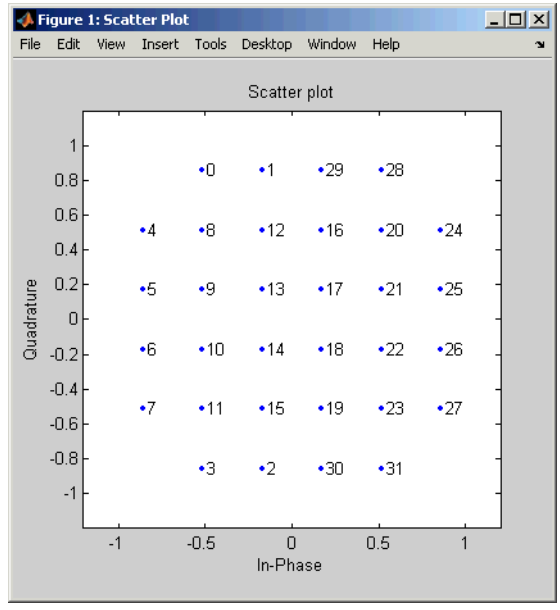
```
M = 16;
x = [0:M-1];
scatterplot(pskmod(x,M));
```



Constellation for 32-QAM. The code below plots a QAM constellation having 32 points and a peak power of 1 watt. The example also illustrates how to label the plot with the numbers that form the input to the modulator.

```
M = 32;
x = [0:M-1];
y = qammod(x,M);
scale = modnorm(y,'peakpow',1);
y = scale*y; % Scale the constellation.
```

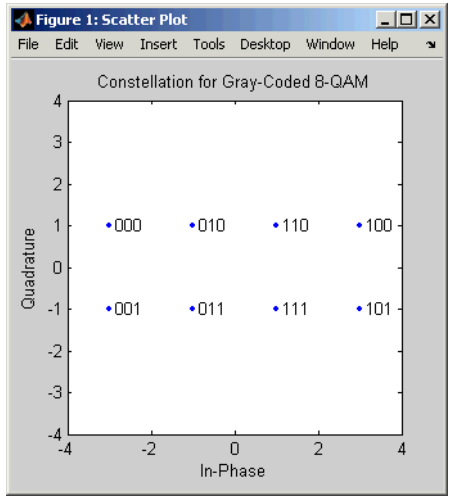
```
scatterplot(y); % Plot the scaled constellation.

% Include text annotations that number the points.
hold on; % Make sure the annotations go in the same figure.
for jj=1:length(y)
   text(real(y(jj)),imag(y(jj)),[' ' num2str(jj-1)]);
end
hold off;
```



Gray-Coded Signal Constellation. The example below plots an 8–QAM signal Gray-coded constellation, labeling the points using binary numbers so that you can verify visually that the constellation uses Gray coding.

```
M = 8;
x = [0:M-1];
y = qammod(x,M,[],'gray');

% Plot the Gray-coded constellation.
scatterplot(y,1,0,'b.'); % Dots for points.
% Include text annotations that number the points in binary.
```

```
hold on; % Make sure the annotations go in the same figure.
annot = dec2bin([0:length(y)-1],log2(M));
text(real(y)+0.15,imag(y),annot);
axis([-4 4 -4 4]);
title('Constellation for Gray-Coded 8-QAM');
hold off;
```



Customized Constellation for QAM. The code below describes and plots a constellation with a customized structure.

```
% Describe constellation.
inphase = [1/2 -1/2 1 0 3/2 -3/2 1 -1];
quadr = [1 1 0 2 1 1 2 2];
inphase = [inphase; -inphase]; inphase = inphase(:);
quadr = [quadr; -quadr]; quadr = quadr(:);
const = inphase + j*quadr;

% Plot constellation.
scatterplot(const,1,0,'*');
hold on;
axis([-3 3 -3 3]);
title('Customized Constellation for QAM');
hold off;
```

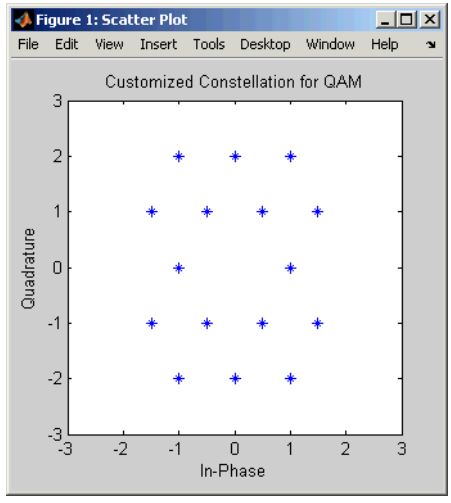


Figure 1: Scatter Plot
File Edit View Insert Tools Desktop Window Help
Customized Constellation for QAM
Quadrature
In-Phase
3
2
1
0
-1
-2
-3
-3 -2 -1 0 1 2 3

Selected Bibliography for Modulation

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[2] Proakis, John G., *Digital Communications*, 3rd ed., New York, McGraw-Hill, 1995.

[3] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

9

Special Filters

The Communications Toolbox includes several functions that can help you design and use filters. Other filtering capabilities are in the Signal Processing Toolbox. The sections of this chapter are as follows.

For a demonstration involving raised cosine filters, type `playshow rcosdemo`.

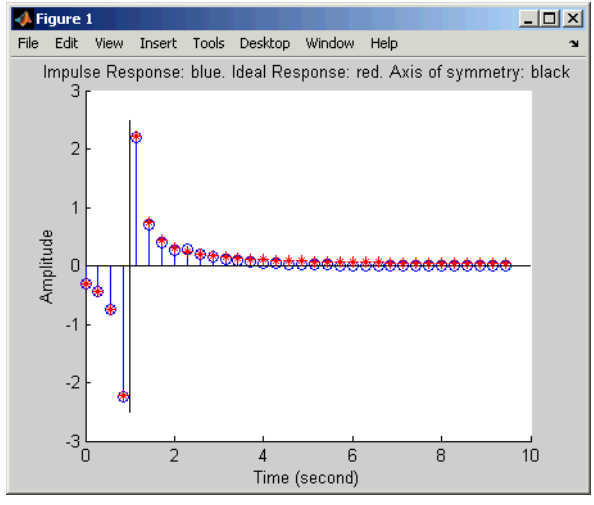
Noncausality and the Group Delay Parameter

Without propagation delays, both Hilbert filters and raised cosine filters are noncausal. This means that the current output depends on the system's future input. In order to design only *realizable* filters, the `hilbiir`, `rcosine`, and `rcosflt` functions delay the input signal before producing an output. This delay, known as the filter's *group delay*, is the time between the filter's initial response and its peak response. The group delay is defined as

$$-\frac{d}{d\omega}\theta(\omega)$$

where θ is the phase of the filter and ω is the frequency in radians. This delay is set so that the impulse response before time zero is negligible and can safely be ignored by the function.

For example, the Hilbert filter whose impulse is shown below uses a group delay of 1 second. Notice in the figure that the impulse response near time 0 is small and that the large impulse response values occur near time 1.



Example: Compensating for Group Delays When Analyzing Data

Comparing filtered with unfiltered data might be easier if you delay the unfiltered signal by the filter's group delay. For example, suppose you use the code below to filter `x` and produce `y`.

```
tx = 0:4; % Times for data samples
x = [0 1 1 1 1]'; % Binary data samples
% Filter the data and use a delay of 2 seconds.
delay = 2;
[y,ty] = rcosflt(x,1,8,'fir',.3,delay);
```

Here, the elements of `tx` and `ty` represent the times of each sample of `x` and `y`, respectively. However, `y` is delayed relative to `x`, so corresponding elements of `x` and `y` do not have the same time values. Plotting `y` against `ty` and `x` against `tx` is less useful than plotting `y` against `ty` and `x` against a *delayed version* of `tx`.

```
% Top plot
subplot(2,1,1), plot(tx,x,'*',ty,y);
legend('Data','Filtered data');
title('Data with No Added Delay');
% Bottom plot delays tx.
subplot(2,1,2), plot(tx+delay,x,'*',ty,y);
legend('Data','Filtered data');
title('Data with an Added Delay');
```

For another example of compensating for group delay, see the raised-cosine filter demo by typing `playshow rcosdemo`.

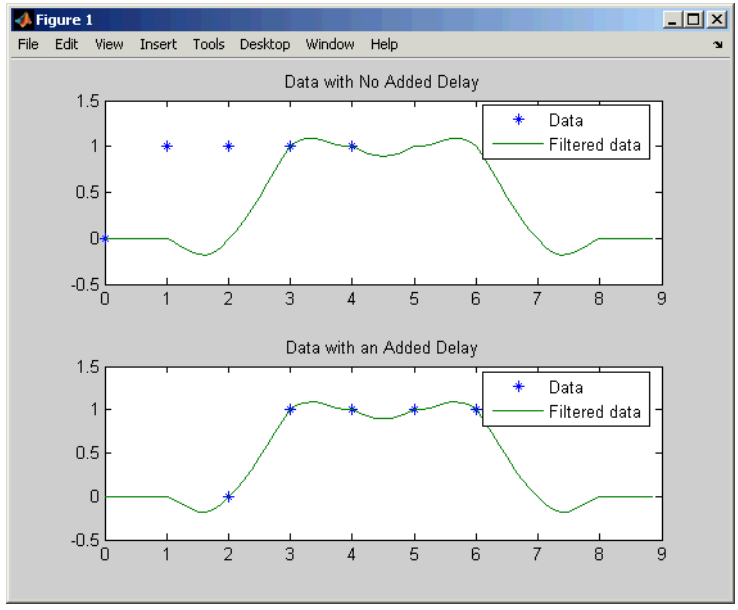


Figure 1
File Edit View Insert Tools Desktop Window Help
Data with No Added Delay
Data
Filtered data
1.5
1
0.5
0
-0.5
0 1 2 3 4 5 6 7 8 9
Data with an Added Delay
Data
Filtered data
1.5
1
0.5
0
-0.5
0 1 2 3 4 5 6 7 8 9

Designing Hilbert Transform Filters

The `hilbiir` function designs a Hilbert transform filter and produces either

- A plot of the filter's impulse response
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Example with Default Parameters

For example, typing simply

```
hilbiir
```

plots the impulse response of a fourth-order digital Hilbert transform filter having a 1-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 having a 1-second group delay. The plot is in the figure in "Noncausality and the Group Delay Parameter" on page 9-2.

To compute this filter's transfer function, use the command below.

```
[num,den] = hilbiir

num =

   -0.3183   -0.3041   -0.5160   -1.8453    3.3105


den =

    1.0000   -0.4459   -0.1012   -0.0479   -0.0372
```

Here, the vectors `num` and `den` contain the coefficients of the numerator and denominator, respectively, of the transfer function in ascending order of powers of z^{-1}.

The commands in this section use the function's default parameters. You can also control the filter design by specifying the sample time, group delay,

bandwidth, and tolerance index. The reference entry for `hilbiir` explains these parameters. The group delay is also mentioned above in “Noncausality and the Group Delay Parameter” on page 9-2.

Filtering with Raised Cosine Filters

The `rcosflt` function applies a raised cosine filter to data. Because `rcosflt` is a versatile function, you can

- Use `rcosflt` to both design and implement the filter
- Specify a raised cosine filter and use `rcosflt` only to filter the data
- Design and implement either raised cosine filters or square-root raised cosine filters
- Specify the rolloff factor and/or group delay of the filter, if `rcosflt` designs the filter
- Design and implement either FIR or IIR filters

This section discusses the use of sampling rates in filtering and then covers these options. For an additional example, type `playshow rcosdemo` in the MATLAB Command Window.

Sampling Rates

The basic `rcosflt` syntax

```
y = rcosflt(x,Fd,Fs...) % Basic syntax
```

assumes by default that you want to apply the filter to a digital signal `x` whose sampling rate is `Fd`. The filter's sampling rate is `Fs`. The ratio of `Fs` to `Fd` must be an integer. By default, the function upsamples the input data by a factor of `Fs/Fd` before filtering. It upsamples by inserting `Fs/Fd-1` zeros between consecutive input data samples. The upsampled data consists of `Fs/Fd` samples per symbol and has sampling rate `Fs`.

An example using this syntax is below. The output sampling rate is four times the input sampling rate.

```
y1 = rcosflt([1;0;0],1,4,'fir'); % Upsample by factor of 4/1.
```

Maintaining the Input Sampling Rate

You can also override the default upsampling behavior. In this case, the function assumes that the input signal already has sampling rate `Fs` and consists of `Fs/Fd` samples per symbol. You might want to maintain the sampling rate in a receiver's filter if the corresponding transmitter's filter has already upsampled sufficiently.

To maintain the sampling rate, modify the fourth input argument in `rcosflt` to include the string `Fs`. For example, in the first command below, `rcosflt` uses its default upsampling behavior and the output sampling rate is four times the input sampling rate. By contrast, the second command below uses `Fs` in the string argument and thus maintains the sampling rate throughout.

```
y1 = rcosflt([1;0;0],1,4,'fir'); % Upsample by factor of 4/1.
y2 = rcosflt([1;0;0],1,4,'fir/Fs'); % Maintain sampling rate.
```

The second command assumes that the sampling rate of the input signal is 4, and that the input signal contains 4/1 samples per symbol.

An example that uses the `'Fs'` option at the receiver is in "Combining Two Square-Root Raised Cosine Filters" on page 9-11.

Designing Filters Automatically

The simplest syntax of `rcosflt` assumes that the function should both design and implement the raised cosine filter. For example, the command below designs an FIR raised cosine filter and then filters the input vector `[1;0;0]` with it. The second and third input arguments indicate that the function should upsample the data by a factor of 8 (that is, 8/1) during the filtering process.

```
y = rcosflt([1;0;0],1,8);
```

Types of Raised Cosine Filters

You can have `rcosflt` design other types of raised cosine filters by using a fourth input argument. Variations on the previous example are below.

```
y = rcosflt([1;0;0],1,8,'fir'); % Same as original example
y = rcosflt([1;0;0],1,8,'fir/sqrt'); % FIR square-root RC filter
y = rcosflt([1;0;0],1,8,'iir'); % IIR raised cosine filter
```

```
y = rcosflt([1;0;0],1,8,'iir/sqrt'); % IIR square-root RC filter
```

Specifying Filters Using Input Arguments

If you have a transfer function for a raised cosine filter, then you can provide it as an input to `rcosflt` so that `rcosflt` does not design its own filter. This is useful if you want to use `rcosine` to design the filter once and then use the filter many times. For example, the `rcosflt` command below uses the `'filter'` flag to indicate that the transfer function is an input argument. The input `num` is a vector that represents the FIR transfer function by listing its coefficients.

```
num = rcosine(1,8); y = rcosflt([1;0;0],1,8,'filter',num);
```

This syntax for `rcosflt` works whether num represents the transfer function for a square-root raised cosine FIR filter or an ordinary raised cosine FIR filter. For example, the code below uses a square-root raised cosine FIR filter. Only the definition of `num` is different.

```
num = rcosine(1,8,'sqrt'); y = rcosflt([1;0;0],1,8,'filter',num);
```

You can also use a raised cosine IIR filter. To do this, modify the fourth input argument of the `rcosflt` command above so that it contains the string `'iir'` and provide a denominator argument. An example is below.

```
delay = 8;
[num,den] = rcosine(1,8,'iir',.5,delay);
y = rcosflt([1;0;0],1,8,'iir/filter',num,den,delay);
```

Controlling the Rolloff Factor

If `rcosflt` designs the filter automatically, then you can control the rolloff factor of the filter, as described below. If you specify your own filter, then `rcosflt` does not need to know its rolloff factor.

The rolloff factor determines the excess bandwidth of the filter. For example, a rolloff factor of .5 means that the bandwidth of the filter is 1.5 times the input sampling frequency, `Fd`. This also means that the transition band of the filter extends from `.5 * Fd` to `1.5 * Fd`.

The default rolloff factor is .5, but if you want to use a value of .2, then you can use a command such as the one below. Typical values for the rolloff factor are between .2 and .5.

```
y = rcosflt([1;0;0],1,8,'fir',.2); % Rolloff factor is .2.
```

Controlling the Group Delay

If `rcosflt` designs the filter automatically, then you can control the group delay of the filter, as described below. If you specify your own FIR filter, then `rcosflt` does not need to know its group delay.

The filter's group delay is the time between the filter's initial response and its peak response. The default group delay in the implementation is three input samples. To specify a different value, measure it in input symbol periods and provide it as the sixth input argument. For example, the command below specifies a group delay of six input samples, which is equivalent to 6 *8 /1 output samples.

```
y = rcosflt([1;0;0],1,8,'fir',.2,6); % Delay is 6 input samples.
```

The group delay influences the size of the output, as well as the order of the filter if `rcosflt` designs the filter automatically. See the reference page for `rcosflt` for details that relate to the syntax you want to use.

Example: Raised Cosine Filter Delays

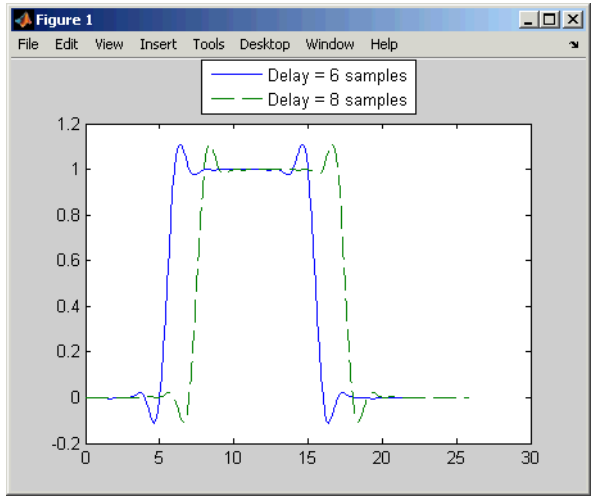
The code below filters a signal using two different group delays. A larger delay results in a smaller error in the frequency response of the filter. The plot shows how the two filtered signals differ, and the output `pt` indicates that the first peak occurs at different times for the two filtered signals. In the plot, the solid line corresponds to a delay of six samples, while the dashed line corresponds to a delay of eight samples.

```
[y,t] = rcosflt(ones(10,1),1,8,'fir',.5,6); % Delay = 6 samples
[y1,t1] = rcosflt(ones(10,1),1,8,'fir',.5,8); % Delay = 8 samples
plot(t,y,t1,y1,'--') % Two curves indicate the different delays.
legend('Delay = 6 samples','Delay = 8 samples','Location','NorthOutside');
peak = t(find(y == max(y))); % Times where first curve peaks
peak1 = t1(find(y1 == max(y1))); % Times where second curve peaks
pt = [min(peak), min(peak1)] % First peak time for both curves
```

The output is below.

```
pt =

   14.6250   16.6250
```

If `Fs/Fd` is at least 4, then a group delay value of at least 8 works well in many cases. In the examples of this section, `Fs/Fd` is 8.



Delays of Six Samples (Solid) and Eight Samples (Dashed)

Combining Two Square-Root Raised Cosine Filters

If you want to split the filtering equally between the transmitter's filter and the receiver's filter, then you can use a pair of square-root raised cosine filters. In theory, the combination of two square-root raised cosine filters is equivalent to a single normal raised cosine filter. However, the limited impulse response of practical square-root raised cosine filters causes a slight difference between the response of two successive square-root raised cosine filters and the response of one raised cosine filter.

Using rcosine and rcosflt to Implement Square-Root Raised Cosine Filters

One way to implement the pair of square-root raised cosine filters is to follow these steps:

1. Use `rcosine` with the `'sqrt'` flag to design a square-root raised cosine filter.
2. Use `rcosflt` in the transmitter section of code to upsample and filter the data.
3. Use `rcosflt` in the receiver section of code to filter the received data *without upsampling* it. Use the `'Fs'` flag to avoid upsampling.

An example of this approach is below. Notice that the syntaxes for `rcosflt` use the `'filter'` flag to indicate that you are providing the filter's transfer function as an input.

```
% First approach
x = randint(100,1,2,1234); % Data
num = rcosine(1,8,'sqrt'); % Transfer function of filter
y1 = rcosflt(x,1,8,'filter',num); % Filter the data.
z1 = rcosflt(y1,1,8,'Fs/filter',num); % Filter the received data
% but do not upsample it.
```

Using rcosflt Alone

Another way to implement the pair of square-root raised cosine filters is to have `rcosflt` both design and use the square-root raised cosine filter. This approach avoids using `rcosine`. The corresponding example code is below. Notice that the syntaxes for `rcosflt` use the `'sqrt'` flag to indicate that you want it to design a square-root raised cosine filter.

```
% Second approach
x = randint(100,1,2,1234); % Data (again)
y2 = rcosflt(x,1,8,'sqrt'); % Design and use a filter.
z2 = rcosflt(y2,1,8,'sqrt/Fs'); % Design and use a filter
% but do not upsample the data.
```

Because these two approaches are equivalent, `y1` is the same as `y2` and `z1` is the same as `z2`.

Designing Raised Cosine Filters

The `rcosine` function designs (but does not apply) filters of these types:

- Finite impulse response (FIR) raised cosine filter
- Infinite impulse response (IIR) raised cosine filter
- FIR square-root raised cosine filter
- IIR square-root raised cosine filter

The function returns the transfer function as output. To learn about applying raised cosine filters, see "Filtering with Raised Cosine Filters" on page 9-7.

Sampling Rates

The `rcosine` function assumes that you want to apply the filter to a digital signal whose sampling rate is `Fd`. The function also requires you to provide the filter's sampling rate, `Fs`. The ratio of `Fs` to `Fd` must be an integer.

Example Designing a Square-Root Raised Cosine Filter

For example, the command below designs a square-root raised cosine FIR filter with a sampling rate of 2, for use with a digital signal whose sampling rate is 1.

```
num = rcosine(1,2,'fir/sqrt')
num =

  Columns 1 through 7

    0.0021   -0.0106    0.0300   -0.0531   -0.0750    0.4092    0.8037

  Columns 8 through 13

    0.4092   -0.0750   -0.0531    0.0300   -0.0106    0.0021
```

Here, the vector `num` contains the coefficients of the filter, in ascending order of powers of z^{-1}.

Other Options in Filter Design

You can also control the filter design by specifying the rolloff factor, group delay, and (for IIR filters) tolerance index explicitly, instead of having `rcosine` use its default values. The reference entry for `rcosine` explains these parameters. The group delay is also mentioned above in “Noncausality and the Group Delay Parameter” on page 9-2.

Selected Bibliography for Special Filters

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

[2] Oppenheim, Alan V., and Ronald W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, N.J., Prentice Hall, 1989.

[3] Proakis, John G., *Digital Communications*, 3rd ed., New York, McGraw-Hill, 1995.

Channels

Communication channels introduce noise, fading, interference, and other distortions into the signals that they transmit. Simulating a communication system involves modeling a channel based on mathematical descriptions of the channel. Different transmission media have different properties and are modeled differently. This chapter describes the channel features of the Communications Toolbox, in the sections listed below.

Channel Features of the Toolbox

This toolbox supports these types of channels:

- Additive white Gaussian noise (AWGN) channel
- Fading channel
- Binary symmetric channel, for binary signals

Many applications use a channel model that combines fading with AWGN. In such cases, you should use the fading channel function first, followed by the AWGN function.

AWGN Channel

An AWGN channel adds white Gaussian noise to the signal that passes through it. To model an AWGN channel, use the `awgn` function. Several examples that illustrate the use of `awgn` are in Chapter 1, "Getting Started". The following demos also use `awgn`: `basicsimdemo`, `vitsimdemo`, and `scattereyedemo`.

Describing the Noise Level of an AWGN Channel

The relative power of noise in an AWGN channel is typically described by quantities such as

- Signal-to-noise ratio (SNR) per sample. This is the actual input parameter to the `awgn` function.
- Ratio of bit energy to noise power spectral density (E_b/N_0). This quantity is used by BERTool and performance evaluation functions in this toolbox.
- Ratio of symbol energy to noise power spectral density (E_s/N_0)

Relationship Between E_s/N_0 and E_b/N_0

The relationship between E_s/N_0 and E_b/N_0, both expressed in dB, is as follows:

$$E_s / N_0 \text{ (dB)} = E_b / N_0 \text{ (dB)} + 10\log_{10}(k)$$

where k is the number of information bits per symbol.

In a communication system, k might be influenced by the size of the modulation alphabet or the code rate of an error-control code. For example, if a system uses a rate-1/2 code and 8-PSK modulation, then the number of information bits per symbol (k) is the product of the code rate and the number of coded bits per modulated symbol: $(1/2)\log_2(8) = 3/2$. In such a system, three information bits correspond to six coded bits, which in turn correspond to two 8-PSK symbols.

Relationship Between E_s/N_0 and SNR

The relationship between E_s/N_0 and SNR, both expressed in dB, is as follows:

$$E_s / N_0 \text{ (dB)} = 10 \log_{10}\left(T_{sym} / T_{samp}\right) + SNR \text{ (dB)} \quad \text{for complex input signals}$$

$$E_s / N_0 \text{ (dB)} = 10 \log_{10}\left(2T_{sym} / T_{samp}\right) + SNR \text{ (dB)} \quad \text{for real input signals}$$

where T_{sym} is the signal's symbol period and T_{samp} is the signal's sampling period.

For example, if a complex baseband signal is oversampled by a factor of 4, then E_s/N_0 exceeds the corresponding SNR by $10 \log_{10}(4)$.

Derivation for Complex Input Signals. You can derive the relationship between E_s/N_0 and SNR for complex input signals as follows:

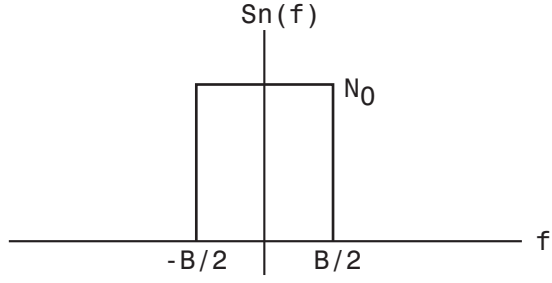
$$\begin{aligned} E_s / N_0 \text{ (dB)} &= 10 \log_{10}\left((S \cdot T_{sym}) / (N / B_n)\right) \\ &= 10 \log_{10}\left((T_{sym} F_s) \cdot (S / N)\right) \\ &= 10 \log_{10}\left(T_{sym} / T_{samp}\right) + SNR \text{ (dB)} \end{aligned}$$

where

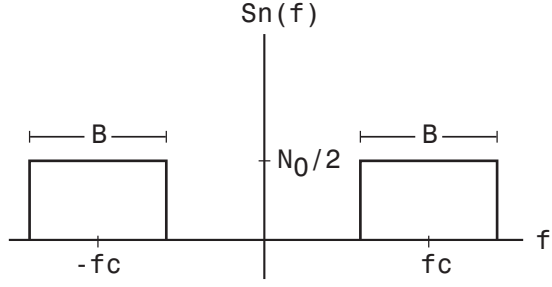
- S = Input signal power, in watts
- N = Noise power, in watts
- B_n = Noise bandwidth, in Hz
- F_s = Sampling frequency, in Hz.

Note that $B_n = F_s = 1/T_{samp}$.

Behavior for Real and Complex Input Signals. The following figures illustrate the difference between the real and complex cases by showing the noise power spectral densities $S_n(f)$ of a real bandpass white noise process and its complex lowpass equivalent.



Complex Lowpass Noise Power Spectral Density



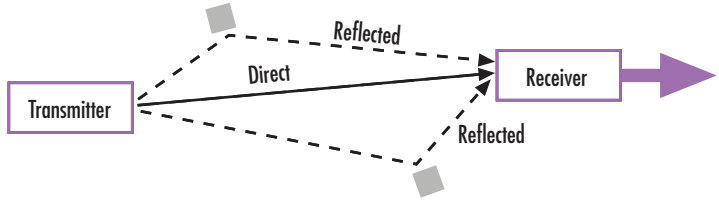
Real Bandpass Noise Power Spectral Density

Fading Channels

Rayleigh and Rician fading channels are useful models of real-world phenomena in wireless communications. These phenomena include multipath scattering effects, time dispersion, and Doppler shifts that arise from relative motion between the transmitter and receiver. This section gives a brief overview of fading channels and describes how to implement them using the toolbox. The topics are as follows:

Overview of Fading Channels

The figure below depicts direct and major reflected paths between a stationary radio transmitter and a moving receiver. The shaded shapes represent reflectors such as buildings.



The major paths result in the arrival of delayed versions of the signal at the receiver. In addition, the radio signal undergoes scattering on a *local* scale for each major path. Such local scattering is typically characterized by a large number of reflections by objects near the mobile. These irresolvable components combine at the receiver and give rise to the phenomenon known as multipath fading. Due to this phenomenon, each major path behaves as a discrete fading path. Typically, the fading process is characterized by a

Rayleigh distribution for a non-line-of-sight path and a Rician distribution for a line-of-sight path.

The relative motion between the transmitter and receiver causes Doppler shifts. Local scattering typically comes from many angles around the mobile. This scenario causes a range of Doppler shifts, known as the Doppler spectrum. The *maximum* Doppler shift corresponds to the local scattering components whose direction exactly opposes the mobile's trajectory.

Fading Channel Features of the Toolbox

The toolbox implements a baseband channel model for multipath propagation scenarios that include

- Local scattering from all angles, with uniform power distribution, around the mobile. This scenario corresponds to what is known as the Jakes Doppler spectrum. The toolbox lets you specify the maximum Doppler shift of the Jakes Doppler spectrum. You can also omit a Doppler shift to model a static channel.
- N discrete fading paths, each with its own delay and average power gain. A channel for which N = 1 is called a frequency-flat fading channel. A channel for which N > 1 is experienced as a frequency-selective fading channel by a signal of sufficiently wide bandwidth.
- A Rayleigh or Rician model for the first major path. Any subsequent paths use a Rayleigh model.

Some additional information about typical values for delays and gains is in "Choosing Realistic Channel Property Values" on page 10-12.

Specifying Fading Channels

This toolbox models a fading channel as a linear FIR filter. Filtering a signal using a fading channel involves these steps:

1. Create a channel object that describes the channel that you want to use. A channel object is a type of MATLAB variable that contains information about the channel, such as the maximum Doppler shift.

2 Adjust properties of the channel object, if necessary, to tailor it to your needs. For example, you can change the path delays or average path gains.

3 Apply the channel object to your signal using the `filter` function.

This section describes how to define, inspect, and manipulate channel objects. The topics are:

Creating Channel Objects

The `rayleighchan` and `ricianchan` functions create fading channel objects. The table below indicates the situations in which each function is suitable.

Function	Object	Situation Modeled
`rayleighchan`	Rayleigh fading channel object	One or more major reflected paths
`ricianchan`	Rician fading channel object	One direct line-of-sight path, possibly combined with one or more major reflected paths

For example, the command below creates a channel object representing a Rayleigh fading channel that acts on a signal sampled at 100,000 Hz. The maximum Doppler shift of the channel is 130 Hz.

```
c1 = rayleighchan(1/100000,130); % Rayleigh fading channel object
```

The object `c1` is a valid input argument for the `filter` function. To learn how to use the `filter` function to filter a signal using a channel object, see "Using Fading Channels" on page 10-14.

Duplicating and Copying Objects. Another way to create an object is to duplicate an existing object and then adjust the properties of the new object, if necessary. If you do this, it is important that you use a `copy` command such as

```
c2 = copy(c1); % Copy c1 to create an independent c2.
```

instead of `c2 = c1`. The `copy` command creates a copy of `c1` that is independent of `c1`. By contrast, the command `c2 = c1` creates `c2` as merely a reference to `c1`, so that `c1` and `c2` always have indistinguishable content.

Viewing Object Properties

A channel object has numerous properties that record information about the channel model, about the state of a channel that has already filtered a signal, and about the channel's operation on a future signal. You can view the properties in these ways:

- To view all properties of a channel object, enter the object's name in the Command Window.
- To view a specific property of a channel object or to assign the property's value to a variable, enter the object's name followed by a dot (period), followed by the name of the property.

In the example below, entering `c1` causes MATLAB to display all properties of the channel object `c1`. Some of the properties have values from the `rayleighchan` command that created `c1`, while other properties have default values.

```
c1 = rayleighchan(1/100000,130); % Create object.
c1 % View all properties of c1.
g = c1.PathGains % Retrieve the PathGains property of c1.
```

The output is

```
c1 =

            ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-005
         MaxDopplerShift: 130
              PathDelays: 0
          AvgPathGaindB: 0
```

```
      NormalizePathGains: 1
            StoreHistory: 0
               PathGains: 0.2104 - 0.6197i
      ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0

g =

   0.2104 - 0.6197i
```

A Rician fading channel object has an additional property that does not appear above, namely, a scalar `KFactor` property.

For more information about what each channel property means, see the reference page for the `rayleighchan` or `ricianchan` function.

Changing Object Properties

To change the value of a writeable property of a channel object, issue an assignment statement that uses dot notation on the channel object. More specifically, dot notation means an expression that consists of the object's name, followed by a dot, followed by the name of the property.

The example below illustrates how to change the `ResetBeforeFiltering` property, indicating that you do not want to reset the channel before each filtering operation.

```
c1 = rayleighchan(1/100000,130) % Create object.
c1.ResetBeforeFiltering = 0 % Do not reset before filtering.
```

The output below displays all the properties of the channel object before and after the change in the value of the `ResetBeforeFiltering` property. Notice that in the second listing of properties, the `ResetBeforeFiltering` property has the value `0`.

```
c1 =

            ChannelType: 'Rayleigh'
      InputSamplePeriod: 1.0000e-005
        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
       MaxDopplerShift: 130
            PathDelays: 0
          AvgPathGaindB: 0
     NormalizePathGains: 1
           StoreHistory: 0
              PathGains: 0.2104 - 0.6197i
     ChannelFilterDelay: 0
  ResetBeforeFiltering: 0
   NumSamplesProcessed: 0
```

Note Some properties of a channel object are read-only. For example, you cannot assign a new value to the `NumSamplesProcessed` property because the channel automatically counts the number of sample it has processed since the last reset.

Linked Properties of Channel Objects

Some properties of an channel object are related to each other such that when one property's value changes, another property's value must change in some corresponding way to keep the channel object consistent. For example, if you change the vector length of `PathDelays`, then the value of `AvgPathGaindB` must change so that its vector length equals that of the new value of `PathDelays`. This is because the length of each of the two vectors equals the number of discrete paths of the channel. For details about linked properties and an example, see the reference page for `rayleighchan` or `ricianchan`.

Configuring Channel Objects

Before you filter a signal using a channel object, you must ensure that the properties of the channel have suitable values for the situation you want to model. This section offers some guidelines to help you choose realistic values that are appropriate for your modeling needs. The topics are

- "Choosing Realistic Channel Property Values" on page 10-12
- "Configuring Channel Objects Based on Simulation Needs" on page 10-13

The syntaxes for viewing and changing values of properties of channel objects are described in "Specifying Fading Channels" on page 10-7.

Choosing Realistic Channel Property Values

Here are some tips for choosing property values that describe realistic channels:

Path Delays

- By convention, the first delay is typically set to zero. The first delay corresponds to the first arriving path.
- For indoor environments, path delays after the first are typically between 1 ns and 100 ns (that is, between 1e-9 s and 1e-7 s).
- For outdoor environments, path delays after the first are typically between 100 ns and 10 μs (that is, between 1e-7 s and 1e-5 s). Very large delays in this range might correspond, for example, to an area surrounded by mountains.
- The ability of a signal to resolve discrete paths is related to its bandwidth. If the difference between the largest and smallest path delays is less than about 1% of the symbol period, then the signal experiences the channel as if it had only one discrete path.

Average Path Gains

- The average path gains in the channel object indicate the average power gain of each fading path. In practice, an average path gain value is a large negative dB value. However, computer models typically use average path gains between -20 dB and 0 dB.

- The dB values in a vector of average path gains often decay roughly linearly as a function of delay, but the specific delay profile depends on the propagation environment.
- To ensure that the expected value of the path gains' total power is 1, you can normalize path gains via the channel object's `NormalizePathGains` property.

Maximum Doppler Shifts

- Some wireless applications, such as standard GSM (Global System for Mobile Communication) systems, prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed v (m/s), then the maximum Doppler shift is given below, where f is the transmission carrier frequency in Hz and c is the speed of light (3e8 m/s).

$$f_d = \frac{vf}{c}$$

- Based on the formula above in terms of the speed of the mobile, a signal from a moving car on a freeway might experience a maximum Doppler shift of about 80 Hz, while a signal from a moving pedestrian might experience a maximum Doppler shift of about 4 Hz. These figures assume a transmission carrier frequency of 900 MHz.
- A maximum Doppler shift of 0 corresponds to a static channel that comes from a Rayleigh or Rician distribution.

K-Factor for Rician Fading Channels

- The Rician K-factor specifies the ratio of specular-to-diffuse power for a direct line-of-sight path. The ratio is expressed linearly, not in dB.
- For Rician fading, the K-factor is typically between 1 and 10.
- A K-factor of 0 corresponds to Rayleigh fading.

Configuring Channel Objects Based on Simulation Needs

Here are some tips for configuring a channel object to customize the filtering process:

- If your data is partitioned into a series of vectors (that you process within a loop, for example), then you can invoke the `filter` function multiple times while automatically saving the channel's state information for use in a subsequent invocation. The state information is visible to you in the channel object's `PathGains` and `NumSamplesProcessed` properties, but also involves properties that are internal rather than visible.

  Note To maintain continuity from one invocation to the next, you must set the `ResetBeforeFiltering` property of the channel object to `0`.

- If you set the `ResetBeforeFiltering` property of the channel object to `0` and want the randomness to be repeatable, then use the `reset` function before filtering any signals, to reset both the channel and the state of the internal random number generator.
- If you want to reset the channel before a filtering operation so that it does not use any previously stored state information, then either use the `reset` function or set the `ResetBeforeFiltering` property of the channel object to `1`. The former method resets the channel object once, while the latter method causes the `filter` function to reset the channel object each time you invoke it.
- If you want to normalize the fading process so that the expected value of the path gains' total power is 1, then set the `NormalizePathGains` property of the channel object to `1`.

Using Fading Channels

After you have created a channel object as described in "Specifying Fading Channels" on page 10-7, you can use the `filter` function to pass a signal through the channel. The arguments to `filter` are the channel object and the signal. At the end of the filtering operation, the channel object retains its state so that you can find out the final path gains or the total number of samples that the channel has processed since it was created or reset. If you configured the channel to avoid resetting its state before each new filtering operation (that is, `ResetBeforeFiltering` is `0`), then the retention of state information is important for maintaining continuity between successive filtering operations.

For an example that illustrates the basic syntax and state retention, see “Power of a Faded Signal” on page 10-16.

If you want to use the channel visualization tool to plot the characteristics of a channel object, you need to set the `StateHistory` property of the channel object to `1`, so that it will get populated with plot information. See for details.

Compensating for Fading

A communication system involving a fading channel usually requires component(s) that compensate for the fading. Here are some typical approaches:

- Differential modulation or a one-tap equalizer can help compensate for a frequency-flat fading channel.
- An equalizer with multiple taps can help compensate for a frequency-selective fading channel.

See Chapter 11, “Equalizers” to learn how to implement equalizers in this toolbox. See the `dpskmod` reference page or the example in “Comparing Empirical with Theoretical Results” on page 10-17 to learn how to implement differential modulation.

Examples Using Fading Channels

The following examples use fading channels:

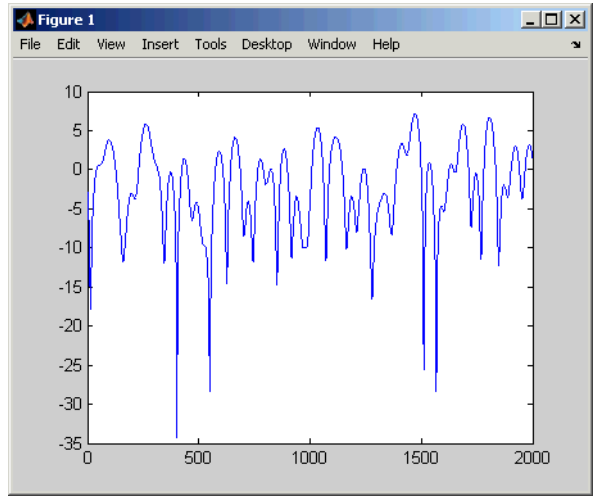
Power of a Faded Signal

The code below plots a faded signal's power (versus sample number). The code also illustrates the syntax of the `filter` and `rayleighchan` functions and the state retention of the channel object. Notice from the output that `NumSamplesProcessed` equals the number of elements in `sig`, the signal.

```
c = rayleighchan(1/10000,100);
sig = j*ones(2000,1); % Signal
y = filter(c,sig); % Pass signal through channel.
c % Display all properties of the channel object.

% Plot power of faded signal, versus sample number.
plot(20*log10(abs(y)))
```

Below are the output and the plot.

```
c =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-004
         MaxDopplerShift: 100
              PathDelays: 0
          AvgPathGaindB: 0
      NormalizePathGains: 1
            StoreHistory: 0
               PathGains: -1.1700 + 0.1288i
     ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 2000
```



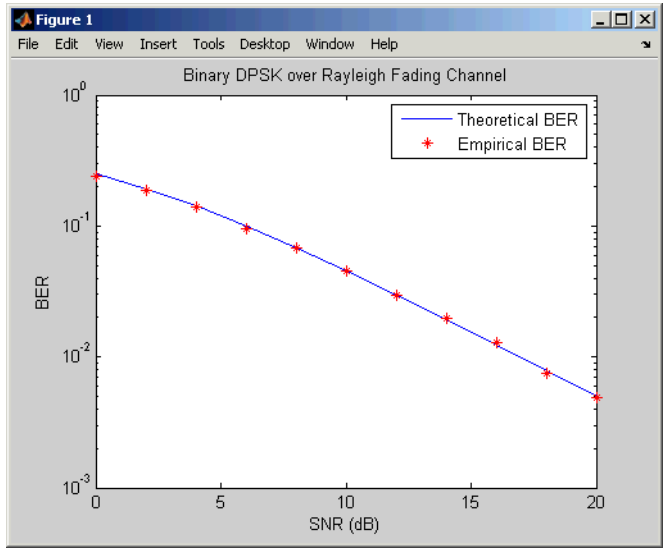
Comparing Empirical with Theoretical Results

The code below creates a frequency-flat Rayleigh fading channel object and uses it to process a DBPSK signal consisting of a single vector. The example continues by computing the bit error rate of the system for different values of the signal-to-noise ratio. Notice that the example uses `filter` before `awgn`; this is the recommended sequence to use when you combine fading with AWGN.

```
% Create Rayleigh fading channel object.
chan = rayleighchan(1/10000,100);

% Generate data and apply fading channel.
M = 2; % DBPSK modulation order
tx = randint(50000,1,M);  % Random bit stream
dpskSig = dpskmod(tx,M);  % DPSK signal
fadedSig = filter(chan,dpskSig); % Effect of channel

% Compute error rate for different values of SNR.
SNR = 0:2:20; % Range of SNR values, in dB.
for n = 1:length(SNR)
   rxSig = awgn(fadedSig,SNR(n)); % Add Gaussian noise.
   rx = dpskdemod(rxSig,M); % Demodulate.
   % Compute error rate.
```

```
    % Ignore first sample because of DPSK initial condition.
    [nErrors, BER(n)] = biterr(tx(2:end),rx(2:end));
end

% Compute theoretical performance results, for comparison.
BERtheory = berfading(SNR,'dpsk',M,1);

% Plot BER results.
semilogy(SNR,BERtheory,'b-',SNR,BER,'r*');
legend('Theoretical BER','Empirical BER');
xlabel('SNR (dB)'); ylabel('BER');
title('Binary DPSK over Rayleigh Fading Channel');
```

The resulting plot shows that the simulation results are close to the theoretical results computed by `berfading`.



Working with Delays

The value of a channel object's `ChannelFilterDelay` property is the number of samples by which the output of the channel lags the input. If you compare the input and output data sets directly, then you must take the delay into account by using appropriate truncating or padding operations.

The example illustrates one way to account for the delay before computing a bit error rate.

```
M = 2; % DQPSK modulation order
bitRate = 50000;

% Create Rayleigh fading channel object.
ch = rayleighchan(1/bitRate,4,[0 0.5/bitRate],[0 -10]);
delay = ch.ChannelFilterDelay;

tx = randint(50000,1,M); % Random bit stream
dpskSig = dpskmod(tx,M); % DPSK signal
fadedSig = filter(ch,dpskSig); % Effect of channel
rx = dpskdemod(fadedSig,M); % Demodulated signal

% Compute bit error rate, taking delay into account.
% Remove first sample because of DPSK initial condition.
tx = tx(2:end); rx = rx(2:end);
% Truncate to account for channel delay.
tx_trunc = tx(1:end-delay); rx_trunc = rx(delay+1:end);
[num,ber] = biterr(tx_trunc,rx_trunc) % Bit error rate
```

The output below shows that the error rate is small. If the example had not compensated for the channel delay, then the error rate would have been close to 1/2.

```
num =

   845


ber =

    0.0169
```

More Information About Working with Delays. The discussion in "Effect of Delays on Recovery of Convolutionally Interleaved Data" on page 7-10 describes two typical ways to compensate for delays. Although the discussion there is about interleaving operations instead of channel modeling, the techniques involving truncating and padding data are equally applicable to channel modeling.

Quasi-Static Channel Modeling

Typically, a path gain in a fading channel changes insignificantly over a period of $1/(100f_d)$ seconds, where f_d is the maximum Doppler shift. Because this period corresponds to a very large number of bits in many modern wireless data applications, assessing performance over a statistically significant range of fading would entail simulating a prohibitively large amount of data. Quasi-static channel modeling provides a more tractable approach, which you can implement using these steps:

1. Generate a random channel realization using a maximum Doppler shift of 0.
2. Process some large number of bits.
3. Compute error statistics.
4. Repeat the steps above many times to produce a distribution of the performance metric.

The example below illustrates the quasi-static channel modeling approach.

```
M = 4; % DQPSK modulation order
numBits = 10000; % Each trial uses 10000 bits.
numTrials = 20; % Number of BER computations

% Note: In reality, numTrials would be a large number
% to get an accurate estimate of outage probabilities
% or packet error rate.
% Use 20 here just to make the example run more quickly.

% Create Rician channel object.
chan = ricianchan; % Static channel
chan.KFactor = 3; % Rician K-factor
```

```
% Because chan.ResetBeforeFiltering is 1 by default,
% FILTER resets the channel in each trial below.

% Compute error rate once for each independent trial.
for n = 1:numTrials
   tx = randint(numBits,1,M);  % Random bit stream
   dpskSig = dpskmod(tx,M);  % DPSK signal
   fadedSig = filter(chan, dpskSig); % Effect of channel
   rxSig = awgn(fadedSig,20);  % Add Gaussian noise.
   rx = dpskdemod(rxSig,M);  % Demodulate.

   % Compute number of symbol errors.
   % Ignore first sample because of DPSK initial condition.
   nErrors(n) = symerr(tx(2:end),rx(2:end))
end
per = mean(nErrors > 0) % Proportion of packets that had errors
```

While the example runs, the Command Window displays the growing list of symbol error counts in the vector `nErrors`. It also displays the packet error rate at the end. The sample output below shows a final value of `nErrors` and omits intermediate values. Your results might vary because of randomness in the example.

```
nErrors =

  Columns 1 through 9

     0     0     0     0     0     0     0     0     0

  Columns 10 through 18

     0     0     0     0     7     0     0     0     0

  Columns 19 through 20

     0   216


per =
```

```
    0.1000
```

More About the Quasi-Static Technique. As an example to show how the quasi-static channel modeling approach can save computation, consider a wireless local area network (LAN) in which the carrier frequency is 2.4 GHz, mobile speed is 1 m/s, and bit rate is 10 Mb/s. The following expression shows that the channel changes insignificantly over 12,500 bits:

$$\begin{aligned}\left(\frac{1}{100 f_d}\ \text{s}\right)(10\ \text{Mb/s}) &= \left(\frac{c}{100 v f}\ \text{s}\right)(10\ \text{Mb/s}) \\ &= \frac{3\times 10^8\ \text{m/s}}{100(1\ \text{m/s})(2.4\ \text{GHz})}(10\ \text{Mb/s}) \\ &= 12{,}500\ \text{b}\end{aligned}$$

A traditional Monte Carlo approach for computing the error rate of this system would entail simulating thousands of times the number of bits shown above, perhaps tens of millions of bits. By contrast, a quasi-static channel modeling approach would simulate a few packets at each of about 100 locations to arrive at a spatial distribution of error rates. From this distribution one could determine, for example, how reliable the communication link is for a random location within the indoor space. If each simulation contains 5,000 bits, then 100 simulations would process half a million bits in total. This is substantially fewer bits compared to the traditional Monte Carlo approach.

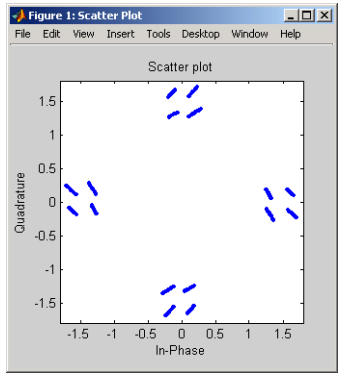
Filtering Using a Loop

The section "Configuring Channel Objects Based on Simulation Needs" on page 10-13 indicates how to invoke the `filter` function multiple times while maintaining continuity from one invocation to the next. The example below invokes `filter` within a loop and uses the small data sets from successive iterations to create an animated effect. The particular channel in this example is a Rayleigh fading channel with two discrete major paths.

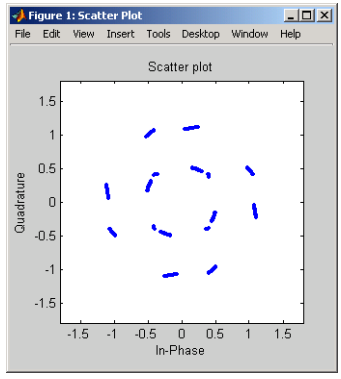
```
% Set up parameters.
M = 4; % QPSK modulation order
bitRate = 50000; % Data rate is 50 kb/s.
numTrials = 125; % Number of iterations of loop

% Create Rayleigh fading channel object.
```

```
ch = rayleighchan(1/bitRate,4,[0 2e-5],[0 -9]);
% Indicate that FILTER should not reset the channel
% in each iteration below.
ch.ResetBeforeFiltering = 0;

% Initialize scatter plot.
h = scatterplot(0);

% Apply channel in a loop, maintaining continuity.
% Plot only the current data in each iteration.
for n = 1:numTrials
   tx = randint(500,1,M); % Random bit stream
   pskSig = pskmod(tx,M); % PSK signal
   fadedSig = filter(ch, pskSig); % Effect of channel

   % Plot the new data from this iteration.
   h = scatterplot(fadedSig,1,0,'b.',h);
   axis([-1.8 1.8 -1.8 1.8]) % Adjust axis limits.
   drawnow; % Refresh the image.
end
```

The scatter plot changes with each iteration of the loop, and the exact content varies because the fading process involves random numbers. Below are some snapshots of typical images that the example can produce.



Sample Scatter Plot (a)



Sample Scatter Plot (b)

Storing Channel State History

By default, the `PathGains` property of a channel object stores the current complex path gain vector.

Setting the `StoreHistory` property of a channel to `true` will make it store the last N path gain vectors, where N is the length of the vector processed through the channel. The following code illustrates this property:

```
h = rayleighchan(1/100000, 130);  % Rayleigh channel
tx = randint(10, 1, 2);           % Random bit stream
dpskSig = dpskmod(tx, 2);         % DPSK signal
h.StoreHistory = true;            % Allow states to be stored
```

```
y = filter(h, dpskSig);          % Run signal through channel
h.PathGains                      % Display the stored path gains data

h.PathGains =

  -0.0460 - 1.1873i
  -0.0439 - 1.1881i
  -0.0418 - 1.1889i
  -0.0397 - 1.1898i
  -0.0376 - 1.1904i
  -0.0355 - 1.1912i
  -0.0334 - 1.1920i
  -0.0313 - 1.1928i
  -0.0296 - 1.1933i
  -0.0278 - 1.1938i
```

The last element is the current path gain of the channel.

Note that setting `StoreHistory` to `true` will significantly slow down the execution speed of the channel's `filter` function.

Using the Channel Visualization Tool

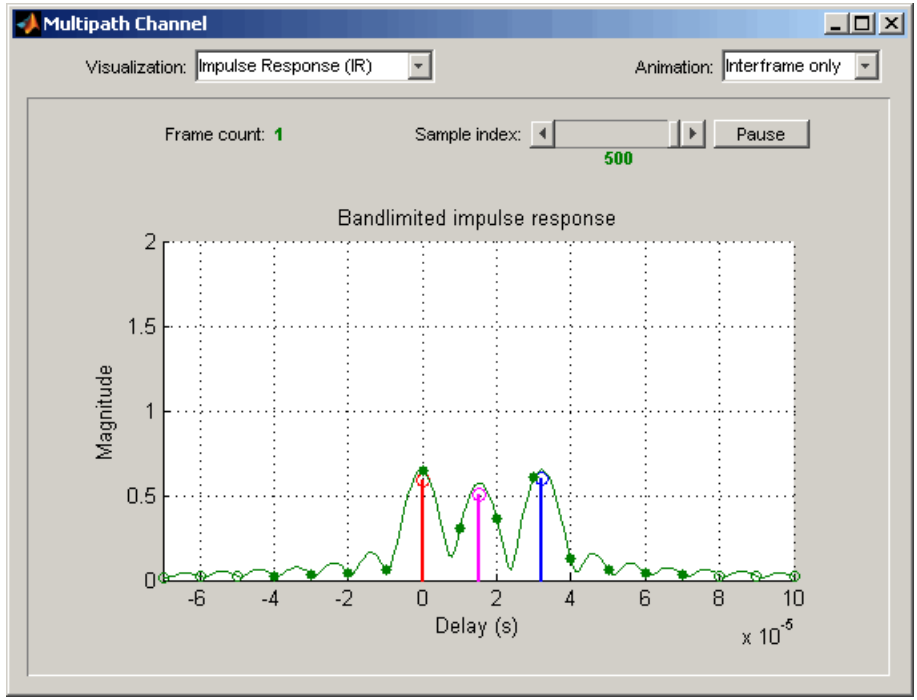
The Communications Toolbox provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See "Fading Channels" on page 10-6 for a description of fading channels and objects.

To open the channel visualization tool, type `plot(h)` at the command line, where `h` is a channel object that contains plot information. To populate a channel object with plot information, run a signal through it after setting its `StoreHistory` property to `true`.

For example, the following code opens the channel visualization tool showing a three-path Rayleigh channel through which a random signal is passed:

```
% Three-Path Rayleigh channel
h = rayleighchan(1/100000, 130, [0 1.5e-5 3.2e-5], [0, -3, -3]);
tx = randint(500, 1, 2);         % Random bit stream
dpskSig = dpskmod(tx, 2);        % DPSK signal
h.StoreHistory = true;           % Allow states to be stored
```

```
y = filter(h, dpskSig);          % Run signal through channel
plot(h);                         % Call Channel Visualization Tool
```



See “Examples of Using the Channel Visualization Tool” on page 10-35 for the basic usage cases of the channel visualization tool.

Note that this tool can also be accessed from the Communications Blockset.

Parts of the GUI

The **Visualization** pull-down menu allows you to choose the visualization method. See “Visualization Options” on page 10-27 for details.

The **Frame count** counter shows the index of the current frame. It shows the number of frames processed by the filter method since the channel object was constructed or reset. A “frame” is a vector of M elements, interpreted to be M

successive samples that are uniformly spaced in time, with a sample period equal to that specified for the channel.

The **Sample index** slider control indicates which channel snapshot is currently being displayed, while the **Pause** button pauses a running animation until you click it again. The slider control and **Pause** button apply to all visualizations except the `Doppler Spectrum`.

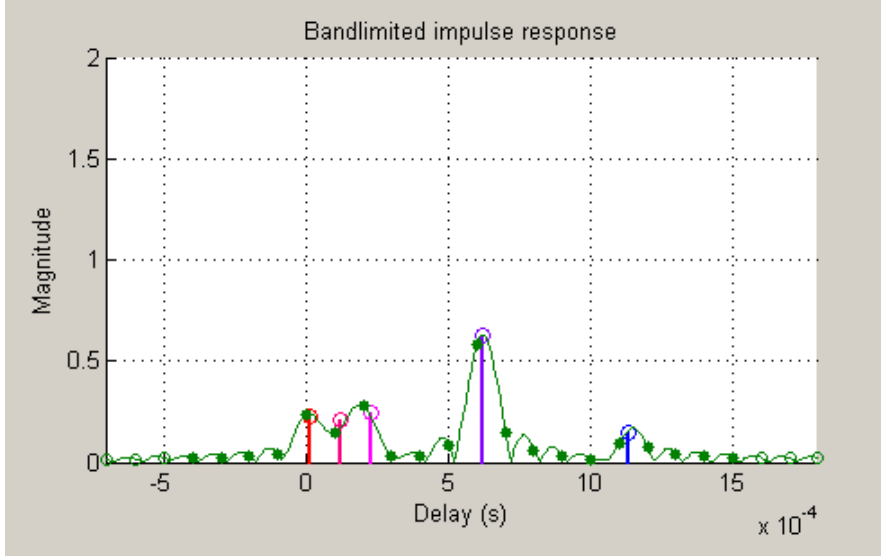
The **Animation** pull-down menu allows you to select how you want to display the channel snapshots within each frame. Setting this to `Slow` makes the tool show channel snapshots in succession, starting at the sample set by the **Sample index** slider control. Selecting `Medium` or `Fast` makes the tool show fewer uniformly spaced snapshots, allowing you to go through the channel snapshots more rapidly. Selecting `Interframe only` (the default selection) prevents automatic animation of snapshots within the same frame. The **Animation** menu applies to all visualizations except the `Doppler Spectrum`.

Visualization Options

The channel visualization tool plots the characteristics of a filter in various ways. Simply choose the visualization method from the **Visualization** menu, and the plot will update itself automatically.

The following visualization methods are currently available:

Impulse Response (IR). This plot shows the magnitudes of two impulse responses: the multipath response (infinite bandwidth) and the bandlimited channel response.



The multipath response is represented by stems, each corresponding to one multipath component. The component with the smallest delay value is shown in red, while the component with the largest delay value is shown in blue. Components with intermediate delay values are shades between red and blue, becoming more blue for larger delays.

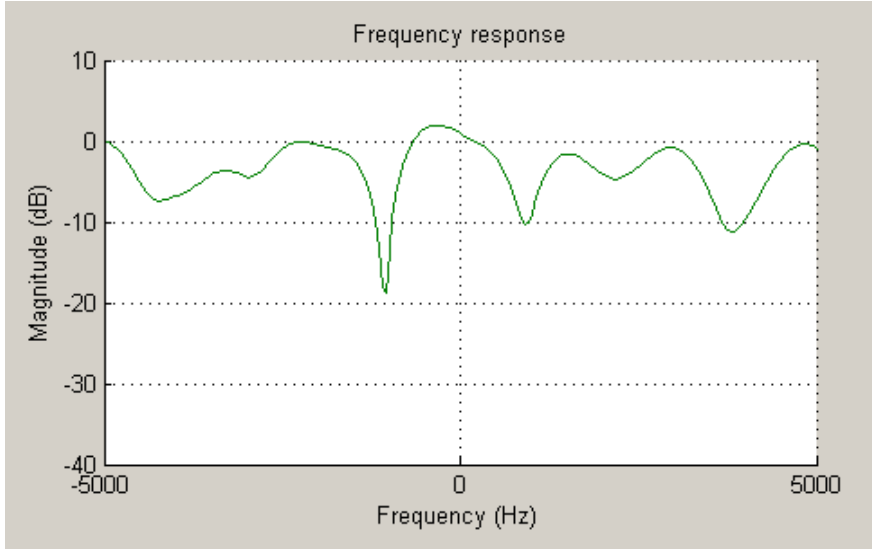
The bandlimited channel response is represented by the green curve. This response is the result of convolving the multipath impulse response, described above, with a sinc pulse of period, T, equal to the input signal's sample period.

The solid green circles represent the channel filter response sampled at rate 1/T. The output of the channel filter is the convolution of the input signal (sampled at rate 1/T) with this discrete-time FIR channel filter response. For computational speed, the response is truncated.

The hollow green circles represent sample values not captured in the channel filter response that is used for processing the input signal.

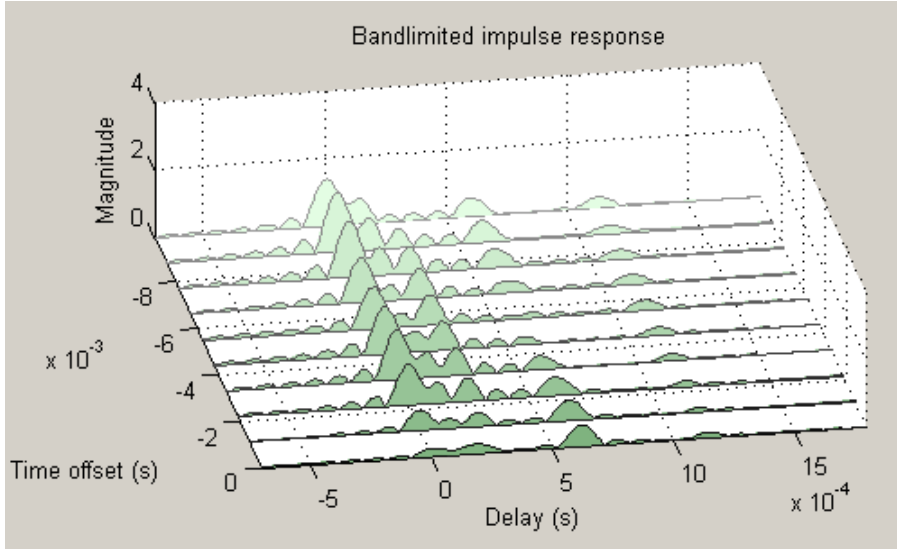
Note that these impulse responses vary over time. You can use the slider to visualize how the impulse response changes over time for the current frame (i.e., input signal vector over time).

Frequency Response (FR). This plot shows the magnitude (in dB) of the frequency response of the multipath channel over the signal bandwidth.



As with the impulse response visualization, you can visualize how this frequency response changes over time.

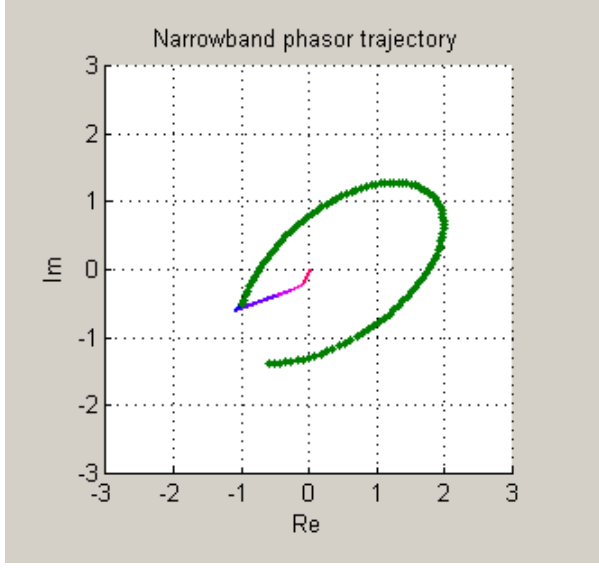
IR Waterfall. This plot shows the evolution of the magnitude impulse response over time.



It shows ten snapshots of the bandlimited channel impulse response within the last frame, with the darkest green curve showing the current response.

The time offset is the time of the channel snapshot relative to the current response time.

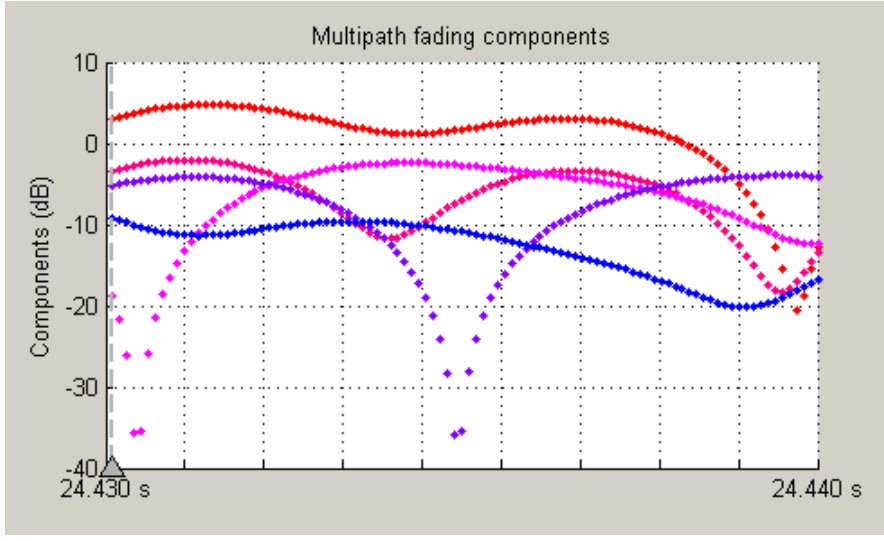
Phasor Trajectory. This plot shows phasors (vectors representing magnitude and phase) for each multipath component, using the same color code that was used for the impulse response plot.



The phasors are connected end-to-end in order of path delay, and the trajectory of the resultant phasor is plotted as a green line. This resultant phasor is referred to as the narrowband phasor.

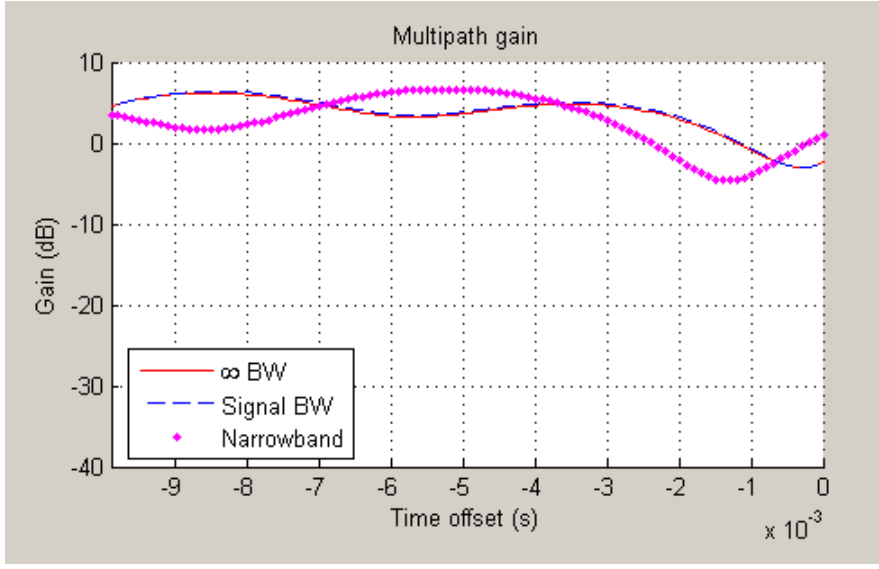
This plot can be used to determine the impact of the multipath channel on a narrowband signal. A narrowband signal is defined here as having a sample period much greater than the span of delays of the multipath channel (alternatively, a signal bandwidth much smaller than the coherence bandwidth of the channel). Thus, the multipath channel can be represented by a single complex gain, which is the sum of all the multipath component gains. When the narrowband phasor trajectory passes through or near the origin, it corresponds to a deep narrowband fade.

Multipath Components. This plot shows the magnitudes of the multipath gains over time, using the same color code as that used for the multipath impulse response.



The triangle marker and vertical dashed line represent the start of the current frame. If a frame has been processed previously, its multipath gains may also be displayed.

Multipath Gain. This plot shows the collective gains for the multipath channel for three signal bandwidths.



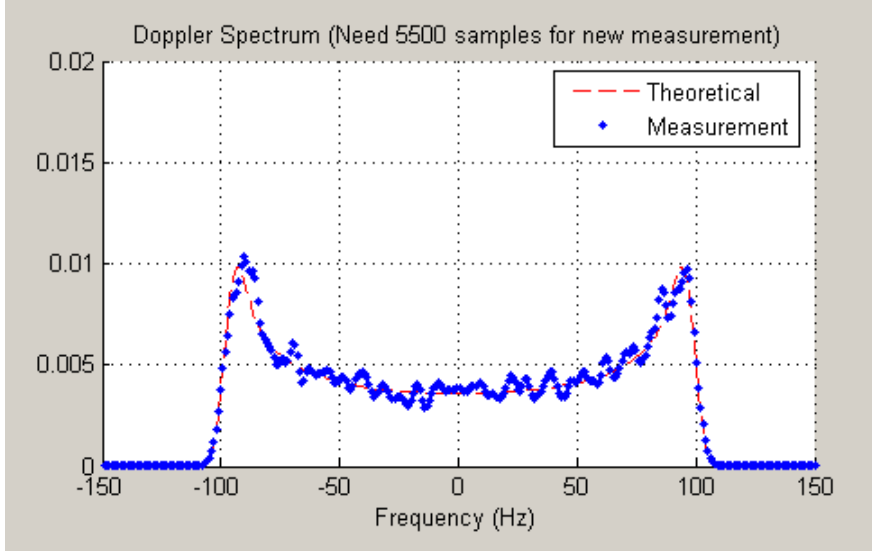
A collective gain is the sum of component magnitudes, as explained in the following:

- Narrowband (magenta dots): This is the magnitude of the narrowband phasor in the above trajectory plot. This curve is sometimes referred to as the "narrowband fading envelope."
- Current signal bandwidth (dashed blue line): This is the sum of the magnitudes of the channel filter impulse response samples (the solid green dots in the impulse response plot). This curve represents the maximum signal energy that can be captured using a RAKE receiver. Its value (or metrics, such as theoretical BER, derived from it) is sometimes referred to as the matched filter bound.
- Infinite bandwidth (solid red line): This is the sum of the magnitudes of the multipath component gains.

In general, the variability of this multipath gain, or of the signal "fading," decreases as signal bandwidth is increased, because multipath components

become more resolvable. If the signal bandwidth curve roughly follows the narrowband curve, you might describe the signal as narrowband. If the signal bandwidth curve roughly follows the infinite bandwidth curve, you might describe the signal as wideband. With the right receiver, a wideband signal exploits the path diversity inherent in a multipath channel.

Doppler Spectrum. This plot shows up to two Doppler spectra.



The first Doppler spectrum, represented by the dashed red line, is a theoretical spectrum based on the Doppler filter response used in the multipath channel model. The theoretical Doppler spectrum used for the multipath channel model is known as the "Jakes" spectrum. This Doppler spectrum is used to determine a Doppler filter response. For practical purposes, the Doppler filter response is truncated, which has the effect of modifying the Doppler spectrum, as shown in the plot.

The second Doppler spectrum, represented by the blue dots, is determined by measuring the power spectrum of the multipath fading channel as the model generates path gains. This measurement is meaningful only after enough path gains have been generated. The title above the plot reports how many

samples will need to be processed through the channel before either the first Doppler spectrum or an updated spectrum can be plotted.

If the measured Doppler spectrum is a good approximation of the theoretical Doppler spectrum, then the multipath channel model has generated enough fading gains to yield a reasonable representation of the channel statistics. For instance, if you want to determine the average BER of a communications link with a multipath channel and you want a statistically accurate measure of this average, you may want to ensure that the channel has processed enough samples to yield at least one Doppler spectrum measurement.

Composite Plots. Several composite plots are also available. These are chosen by selecting the following from the **Visualization** pull-down menu:

- `IR and FR` for impulse response and frequency response plots.
- `Components and Gain` for multipath components and multipath gain plots.
- `Components and IR` for multipath components and impulse response plots.
- `Components, IR, and Phasor` for multipath components, impulse response, and phasor trajectory plots.

Examples of Using the Channel Visualization Tool

Here are two examples that show how you might interact with the GUI.

- "Visualizing Samples Within a Frame" on page 10-35
- "Animating Snapshots Across Frames" on page 10-36

Visualizing Samples Within a Frame. This example shows how to visualize samples within a frame through animation. The following lines of code create a Rayleigh channel and open the channel visualization tool:

```
% Create a fast fading channel
h = rayleighchan(1e-4, 100, [0 1.1e-4], [0 0]);

h.StoreHistory = true;       % Allow states to be stored
y = filter(h, ones(100,1));  % Process samples through channel
plot(h);                     % Open channel visualization tool
```

After selecting a visualization option and a speed in the **Animation** menu, move the **Sample index** slider control all the way to the left and press the **Resume** button. Notice how the slider control moves by itself during animation. The sample index increments automatically to show which snapshot you are visualizing.

You can also move the slider control and glance through the samples of the frame as you like.

Animating Snapshots Across Frames. This example shows how to animate snapshots across frames. The following lines of code call the filter and plot methods within a loop to accomplish this:

```
Ts = 1e-4;   % Sample period (s)
fd = 100;    % Maximum Doppler shift

% Path delay and gains
tau = [0.1 1.2 2.3 6.2 11.3]*Ts;
PdB = linspace(0, -10, length(tau)) - length(tau)/20;

nTrials = 10000;   % Number of trials
N = 100;           % Number of samples per frame

h = rayleighchan(Ts, fd, tau, PdB);  % Create channel object
h.NormalizePathGains = false;
h.ResetBeforeFiltering = false;
h.StoreHistory = true;
h % Show channel object

% Channel fading simulation
for trial = 1:nTrials
    x = randint(10000, 1, 4);
    dpskSig = dpskmod(x, 4);
    y = filter(h, dpskSig);
    plot(h);
    % The line below returns control to the command line in case
    % the GUI is closed while this program is still running
    if isempty(findobj('name', 'Multipath Channel')), break; end;
end
```

While the animation is running, you can move the slider control and change the sample index (which also makes the animation pause). After pressing the **Resume** button, the plot will continue to animate.

Note that the property `ResetBeforeFiltering` needs to be set to false so that the state information in the channel would not get reset after the processing of each frame.

Binary Symmetric Channel

A binary symmetric channel corrupts a binary signal by reversing each bit with a fixed probability. Such a channel can be useful for testing error-control coding.

To model a binary symmetric channel, use the `bsc` function. The two input arguments are the binary signal and the probability, `p`.

If you want to model a binary channel whose statistical description involves the number of errors per codeword, then see the description of `randerr` in "Random Bit Error Patterns" on page 2-5.

Example: Introducing Noise in a Convolutional Code

The example below introduces bit errors in a convolutional code with probability 0.01.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Trellis
msg = ones(10000,1); % Data to encode
code = convenc(ones(10000,1),t); % Encode using convolutional code.
[ncode,err] = bsc(code,.01); % Introduce errors in code.
numchanerrs = sum(sum(err)) % Number of channel errors
dcode = vitdec(ncode,t,2,'trunc','hard'); % Decode.
[numsyserrs,ber] = biterr(dcode,msg) % Errors after decoding
```

The output below shows that the decoder corrects some, but not all, of the errors that `bsc` introduced into the code. Your results might vary because the channel errors are random.

```
numchanerrs =

   132


numsyserrs =

    27
```

```
ber =

    0.0027
```

Selected Bibliography for Channels

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

[2] Jakes, William C., ed. *Microwave Mobile Communications*, New York, IEEE Press, 1974.

[3] Lee, William C. Y., *Mobile Communications Design Fundamentals*, Second Edition, New York, Wiley, 1993.

11

Equalizers

Time-dispersive channels can cause intersymbol interference (ISI). For example, in a multipath scattering environment, the receiver sees delayed versions of a symbol transmission, which can interfere with other symbol transmissions. An equalizer attempts to mitigate ISI and thus improve the receiver's performance. This chapter describes the equalizer features of the Communications Toolbox, in the sections listed below.

Equalizer Features of the Toolbox

This toolbox supports these distinct classes of equalizers, each with a different overall structure:

- Linear equalizers, a class that is further divided into these categories:
  - Symbol-spaced equalizers
  - Fractionally spaced equalizers (FSE)
- Decision-feedback equalizers (DFE)
- MLSE (Maximum-Likelihood Sequence Estimation) equalizer that uses the Viterbi algorithm. To learn how to use the MLSE equalizer capabilities, see "Using MLSE Equalizers" on page 11-28.

Linear and decision-feedback equalizers are adaptive equalizers that use an adaptive algorithm when operating. For each of the adaptive equalizer classes listed above, this toolbox supports these adaptive algorithms:

- Least mean square (LMS)
- Signed LMS, including these types: sign LMS, signed regressor LMS, and sign-sign LMS
- Normalized LMS
- Variable-step-size LMS
- Recursive least squares (RLS)
- Constant modulus algorithm (CMA)

To learn how to use the adaptive equalizer capabilities, start with "Using Adaptive Equalizer Functions and Objects" on page 11-8. For brief background material on the supported adaptive equalizer types, see "Overview of Adaptive Equalizer Classes" on page 11-3. For more detailed background material, see the works listed in "Selected Bibliography for Equalizers" on page 11-35.

Overview of Adaptive Equalizer Classes

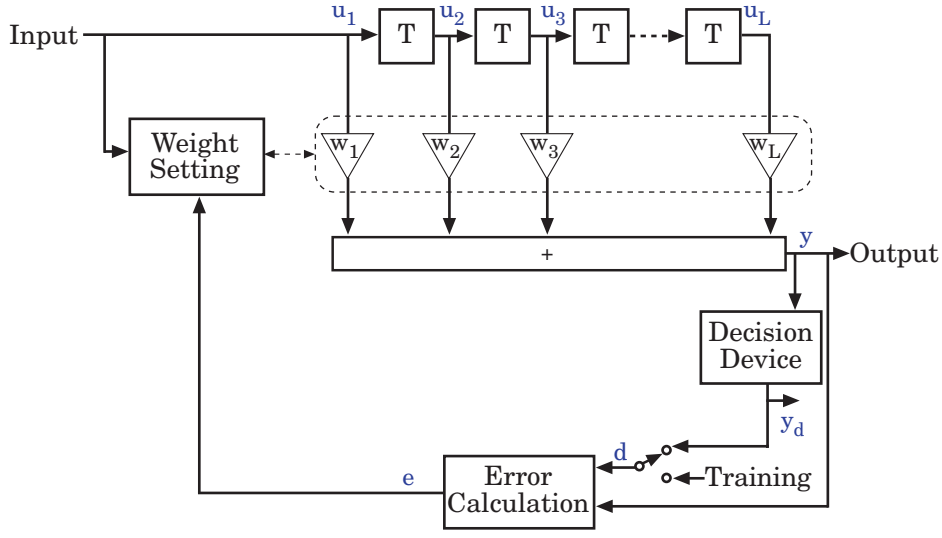
These topics give some background information about the supported classes of adaptive equalizers:

For more detailed background material, see the works listed in "Selected Bibliography for Equalizers" on page 11-35. For more information about particular adaptive algorithms, see the reference pages for the corresponding functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`.

Symbol-Spaced Equalizers

A symbol-spaced linear equalizer consists of a tapped delay line that stores samples from the input signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period. This class of equalizer is called "symbol-spaced" because the sample rates of the input and output are equal.

Below is a schematic of a symbol-spaced linear equalizer with N weights, where the symbol period is T.



Updating the Set of Weights

The algorithms for the Weight Setting and Error Calculation blocks in the schematic are determined by the adaptive algorithm chosen from the list in "Equalizer Features of the Toolbox" on page 11-2. The new set of weights depends on these quantities:

- The current set of weights
- The input signal
- The output signal
- For adaptive algorithms other than CMA, a reference signal, d, whose characteristics depend on the operation mode of the equalizer

Reference Signal and Operation Modes

The table below briefly describes the nature of the reference signal for each of the two operation modes.

Operation Mode of Equalizer	Reference Signal
Training mode	Preset known transmitted sequence
Decision-directed mode	Detected version of the output signal, denoted by y_d in the schematic

In typical applications, the equalizer begins in training mode to gather information about the channel, and later switches to decision-directed mode.

Error Calculation

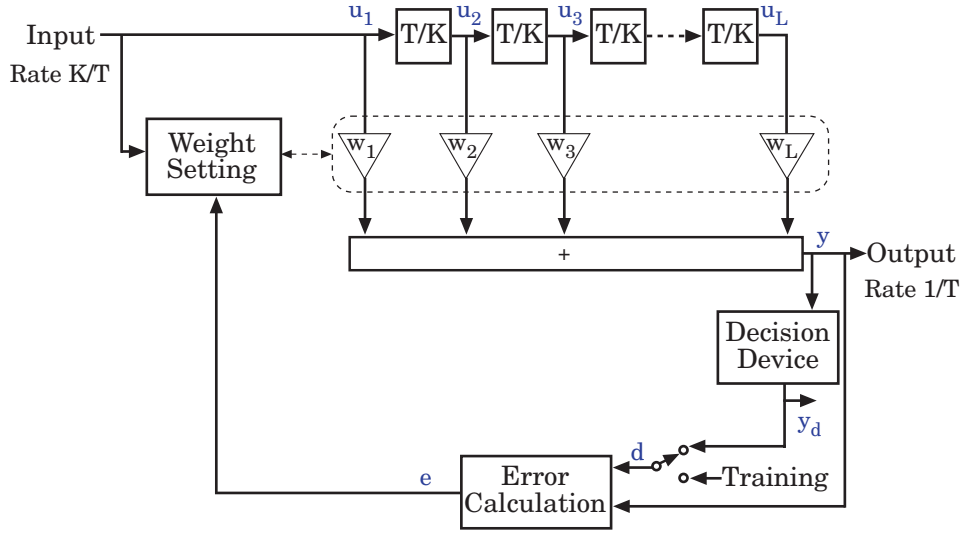
The error calculation operation produces a signal given by the expression below, where R is a constant related to the signal constellation.

$$e = \begin{cases} d - y & \text{Algorithms other than CMA} \\ y(R - |y|^2) & \text{CMA} \end{cases}$$

Fractionally Spaced Equalizers

A fractionally spaced equalizer is a linear equalizer that is similar to a symbol-spaced linear equalizer, as described in “Symbol-Spaced Equalizers” on page 11-3. By contrast, however, a fractionally spaced equalizer receives K input samples before it produces one output sample and updates the weights, where K is an integer. In many applications, K is 2. The output sample rate is 1/T, while the input sample rate is K/T. The weight-updating occurs at the output rate, which is the slower rate.

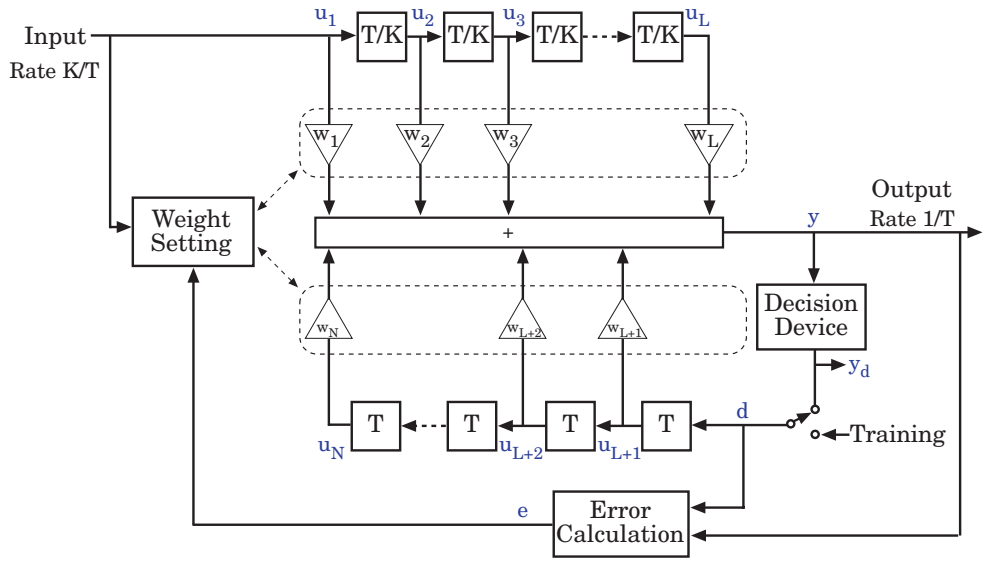
Below is a schematic of a fractionally spaced equalizer.



Decision-Feedback Equalizers

A decision-feedback equalizer is a nonlinear equalizer that contains a forward filter and a feedback filter. The forward filter is similar to the linear equalizer described in “Symbol-Spaced Equalizers” on page 11-3, while the feedback filter contains a tapped delay line whose inputs are the decisions made on the equalized signal. The purpose of a DFE is to cancel intersymbol interference while minimizing noise enhancement. By contrast, noise enhancement is a typical problem with the linear equalizers described earlier.

Below is a schematic of a fractionally spaced DFE with L forward weights and N-L feedback weights. The forward filter is at the top and the feedback filter is at the bottom. If K is 1, then the result is a symbol-spaced DFE instead of a fractionally spaced DFE.



In each symbol period, the equalizer receives K input samples at the forward filter, as well as one decision or training sample at the feedback filter. The equalizer then outputs a weighted sum of the values in the forward and feedback delay lines, and updates the weights to prepare for the next symbol period.

Note The algorithm for the Weight Setting block in the schematic *jointly* optimizes the forward and feedback weights. Joint optimization is especially important for the RLS algorithm.

Using Adaptive Equalizer Functions and Objects

This section gives an overview of the process you typically use in MATLAB to take advantage of the adaptive equalizer capabilities. The MLSE equalizer has a different interface, described in "Using MLSE Equalizers" on page 11-28.

Basic Procedure for Equalizing a Signal

Equalizing a signal using the Communications Toolbox involves these steps:

1. Create an equalizer object that describes the equalizer class and the adaptive algorithm that you want to use. An equalizer object is a type of MATLAB variable that contains information about the equalizer, such as the name of the equalizer class, the name of the adaptive algorithm, and the values of the weights.

2. Adjust properties of the equalizer object, if necessary, to tailor it to your needs. For example, you can change the number of weights or the values of the weights.

3. Apply the equalizer object to the signal that you want to equalize, using the `equalize` function.

Example Illustrating the Basic Procedure

This code briefly illustrates the steps in the basic procedure above.

```
% Build a set of test data.
x = pskmod(randint(1000,1),2); % BPSK symbols
rxsig = conv(x,[1 0.8 0.3]);  % Received signal
% Create an equalizer object.
eqlms = lineareq(8,lms(0.03));
% Change the reference tap index in the equalizer.
eqlms.RefTap = 4;
% Apply the equalizer object to a signal.
y = equalize(eqlms,rxsig,x(1:200));
```

In this example, `eqlms` is an equalizer object that describes a linear LMS equalizer having 8 weights and a step size of 0.03. At first, the reference tap index in the equalizer has a default value, but assigning a new value to the property `eqlms.RefTap` changes this index. Finally, the `equalize` command

uses the `eqlms` object to equalize the signal `rxsig` using the training sequence `x(1:200)`.

Learning More About Adaptive Equalizer Functions

Keeping the basic procedure in mind, you can read other portions of this chapter to learn more details about

- How to create objects that represent different classes of adaptive equalizers and different adaptive algorithms
- How to adjust properties of an adaptive equalizer or properties of an adaptive algorithm
- How to equalize signals using an adaptive equalizer object

Specifying an Adaptive Algorithm

Configuring an equalizer involves choosing an adaptive algorithm and indicating your choice when creating an equalizer object in MATLAB. This section includes information that might help you choose an adaptive algorithm. It then describes how to indicate your choice and how to access properties of an adaptive algorithm that you have chosen.

Choosing an Adaptive Algorithm

Although the best choice of adaptive algorithm might depend on your individual situation, here are some generalizations that might influence your choice:

- The LMS algorithm executes quickly but converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, but its complexity grows with the square of the number of weights, roughly speaking. This algorithm can also be unstable when the number of weights is large.
- The various types of signed LMS algorithms simplify hardware implementation.
- The normalized LMS and variable-step-size LMS algorithms are more robust to variability of the input signal's statistics (such as power).
- The Constant modulus algorithm is useful when no training signal is available, and works best for constant-modulus modulations such as PSK.

  However, if CMA has no additional side information, it can introduce phase ambiguity. For example, CMA might find weights that produce a perfect QPSK constellation but might introduce a phase rotation of 90, 180, or 270 degrees. Alternatively, differential modulation can be used to avoid phase ambiguity.

Details about the adaptive algorithms are in the references listed in "Selected Bibliography for Equalizers" on page 11-35.

Indicating a Choice of Adaptive Algorithm

After you have chosen the adaptive algorithm you want to use, you must indicate your choice when creating the equalizer object mentioned in "Basic Procedure for Equalizing a Signal" on page 11-8. The functions listed in the table below provide a way to indicate your choice of adaptive algorithm.

Adaptive Algorithm Function	Type of Adaptive Algorithm
`lms`	Least mean square (LMS)
`signlms`	Signed LMS, signed regressor LMS, sign-sign LMS
`normlms`	Normalized LMS
`varlms`	Variable-step-size LMS
`rls`	Recursive least squares (RLS)
`cma`	Constant modulus algorithm (CMA)

Two typical ways to use a function from the table are as follows:

- Use the function in an inline expression when creating the equalizer object.

  For example, the code below uses the `lms` function inline when creating an equalizer object.

  ```
  eqlms = lineareq(10,lms(0.003));
  ```

- Use the function to create a variable in the MATLAB workspace and then use that variable when creating the equalizer object. The variable is called an adaptive algorithm object.

  For example, the code below creates an adaptive algorithm object named `alg` that represents the adaptive algorithm, and then uses `alg` when creating an equalizer object.

  ```
  alg = lms(0.003);
  eqlms = lineareq(10,alg);
  ```

Note If you want to create an adaptive algorithm object by duplicating an existing one and then changing its properties, then see the important note in "Duplicating and Copying Objects" on page 11-14 about the use of `copy` versus the = operator.

In practice, the two ways are equivalent when your goal is to create an equalizer object or to equalize a signal.

Accessing Properties of an Adaptive Algorithm

The adaptive algorithm functions not only provide a way to indicate your choice of adaptive algorithm, but also let you specify certain properties of the algorithm. For information about what each property of an adaptive algorithm object means, see the reference page for the `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma` function.

To view or change any properties of an adaptive algorithm, use the syntax described for channel objects in "Viewing Object Properties" on page 10-9 and "Changing Object Properties" on page 10-10.

Specifying an Adaptive Equalizer

As mentioned earlier in "Basic Procedure for Equalizing a Signal" on page 11-8, you must create an equalizer object before you can equalize a signal. This section describes how to define an equalizer object and how to access its properties.

Defining an Equalizer Object

To create an equalizer object, use one of the functions listed in the table below.

Function	Type of Equalizer
`lineareq`	Linear equalizer (symbol-spaced or fractionally spaced)
`dfe`	Decision-feedback equalizer

For example, the code below creates three equalizer objects: one representing a symbol-spaced linear RLS equalizer having 10 weights, one representing a fractionally spaced linear RLS equalizer having 10 weights and two samples per symbol, and one representing a decision-feedback RLS equalizer having 3 weights in the feedforward filter and 2 weights in the feedback filter.

```
% Create equalizer objects of different types.
eqlin = lineareq(10,rls(0.3)); % Symbol-spaced linear
eqfrac = lineareq(10,rls(0.3),[-1 1],2); % Fractionally spaced linear
eqdfe = dfe(3,2,rls(0.3)); % DFE
```

Although the `lineareq` and `dfe` functions have different syntaxes, they both require an input argument that represents an adaptive algorithm. To learn how to represent an adaptive algorithm or how to vary properties of the adaptive algorithm, see "Specifying an Adaptive Algorithm" on page 11-10.

Each of the equalizer objects created above is a valid input argument for the `equalize` function. To learn how to use the `equalize` function to equalize a signal, see "Using Adaptive Equalizers" on page 11-17.

Duplicating and Copying Objects

Another way to create an object is to duplicate an existing object and then adjust the properties of the new object, if necessary. If you do this, it is important that you use a `copy` command such as

```
c2 = copy(c1); % Copy c1 to create an independent c2.
```

instead of `c2 = c1`. The `copy` command creates a copy of `c1` that is independent of `c1`. By contrast, the command `c2 = c1` creates `c2` as merely a reference to `c1`, so that `c1` and `c2` always have indistinguishable content.

Accessing Properties of an Equalizer

An equalizer object has numerous properties that record information about the equalizer. Properties can be related to

- The structure of the equalizer (for example, the number of weights).
- The adaptive algorithm that the equalizer uses (for example, the step size in the LMS algorithm). When you create the equalizer object using `lineareq` or `dfe`, the function copies certain properties from the algorithm object to the equalizer object. However, the equalizer object does not retain a connection to the algorithm object.
- Information about the equalizer's current state (for example, the values of the weights). The `equalize` function automatically updates these properties when it operates on a signal.
- Instructions for operating on a signal (for example, whether the equalizer should reset itself before starting the equalization process).

For information about what each equalizer property means, see the reference page for the `lineareq` or `dfe` function.

To view or change any properties of an equalizer object, use the syntax described for channel objects in "Viewing Object Properties" on page 10-9 and "Changing Object Properties" on page 10-10.

Linked Properties of an Equalizer Object

Some properties of an equalizer object are related to each other such that when one property's value changes, another property's value must adjust, or

else the equalizer object would fail to describe a valid equalizer. For example, in a linear equalizer, the `nWeights` property is the number of weights, while the `Weights` property is the value of the weights. If you change the value of `nWeights`, then the value of `Weights` must adjust so that its vector length is the new value of `nWeights`.

To find out which properties are related and how MATLAB compensates automatically when you make certain changes in property values, see the reference page for `lineareq` or `dfe`.

The example below illustrates that when you change the value of `nWeights`, MATLAB automatically changes the values of `Weights` and `WeightInputs` to make their vector lengths consistent with the new value of `nWeights`. Because the example uses the variable-step-size LMS algorithm, `StepSize` is a vector (not a scalar) and MATLAB changes its vector length to maintain consistency with the new value of `nWeights`.

```
eqlvar = lineareq(10,varlms(0.01,0.01,0,1)) % Create equalizer object.
eqlvar.nWeights = 8 % Change the number of weights from 10 to 8.
% MATLAB automatically changes the sizes of eqlvar.Weights and
% eqlvar.WeightInputs.
```

The output below displays all the properties of the equalizer object before and after the change in the value of the `nWeights` property. Notice that in the second listing of properties, the `nWeights`, `Weights`, `WeightInputs`, and `StepSize` properties all have different values compared to the first listing of properties.

```
eqlvar =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Variable Step Size LMS'
                nWeights: 10
              nSampPerSym: 1
                   RefTap: 1
                SigConst: [-1 1]
                InitStep: 0.0100
                 IncStep: 0.0100
                 MinStep: 0
                 MaxStep: 1
            LeakageFactor: 1
```

```
                StepSize: [1x10 double]
                 Weights: [0 0 0 0 0 0 0 0 0 0]
           WeightInputs: [0 0 0 0 0 0 0 0 0 0]
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0


eqlvar =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Variable Step Size LMS'
                nWeights: 8
              nSampPerSym: 1
                   RefTap: 1
                SigConst: [-1 1]
                InitStep: 0.0100
                 IncStep: 0.0100
                 MinStep: 0
                 MaxStep: 1
          LeakageFactor: 1
                StepSize: [1x8 double]
                 Weights: [0 0 0 0 0 0 0 0]
           WeightInputs: [0 0 0 0 0 0 0 0]
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```

Using Adaptive Equalizers

This section describes how to equalize a signal by using the `equalize` function to apply an adaptive equalizer object to the signal. The `equalize` function also updates the equalizer This section assumes that you have already created an adaptive equalizer object, as described in "Specifying an Adaptive Equalizer" on page 11-13. The topics in this section are as follows:

- "Equalizing Using a Training Sequence" on page 11-17
- "Equalizing in Decision-Directed Mode" on page 11-19
- "Delays from Equalization" on page 11-21
- "Equalizing Using a Loop" on page 11-22

For examples that complement those in this section, see the Adaptive Equalization Simulation demo (part I and part II).

Equalizing Using a Training Sequence

In typical applications, an equalizer begins by using a known sequence of transmitted symbols when adapting the equalizer weights. The known sequence, called a training sequence, enables the equalizer to gather information about the channel characteristics. After the equalizer finishes processing the training sequence, it adapts the equalizer weights in decision-directed mode using a detected version of the output signal. To use a training sequence when invoking the `equalize` function, include the symbols of the training sequence as an input vector.

Note As an exception, CMA equalizers do not use a training sequence. If an equalizer object is based on CMA, then you should not include a training sequence as an input vector.

The code below illustrates how to use `equalize` with a training sequence. The training sequence in this case is just the beginning of the transmitted message.

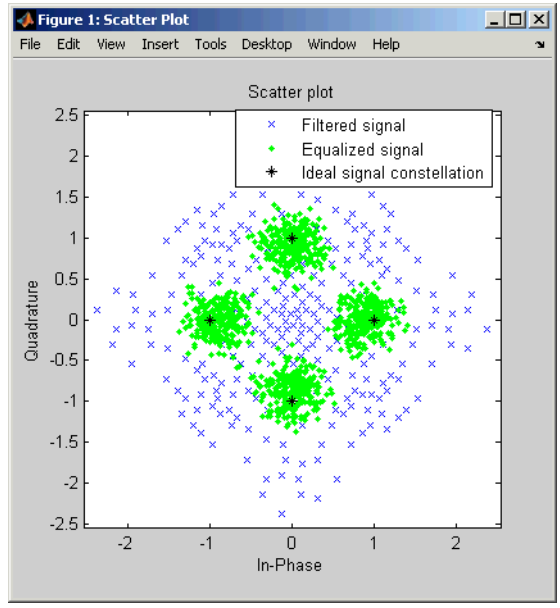
```
% Set up parameters and signals.
M = 4; % Alphabet size for modulation
msg = randint(1500,1,M); % Random message
modmsg = pskmod(msg,M); % Modulate using QPSK.
trainlen = 500; % Length of training sequence
chan = [.986; .845; .237; .123+.31i]; % Channel coefficients
filtmsg = filter(chan,1,modmsg); % Introduce channel distortion.

% Equalize the received signal.
eq1 = lineareq(8, lms(0.01)); % Create an equalizer object.
eq1.SigConst = pskmod([0:M-1],M); % Set signal constellation.
[symbolest,yd] = equalize(eq1,filtmsg,modmsg(1:trainlen)); % Equalize.

% Plot signals.
h = scatterplot(filtmsg,1,trainlen,'bx'); hold on;
scatterplot(symbolest,1,trainlen,'g.',h);
scatterplot(eq1.SigConst,1,0,'k*',h);
legend('Filtered signal','Equalized signal',...
   'Ideal signal constellation');
hold off;

% Compute error rates with and without equalization.
demodmsg_noeq = pskdemod(filtmsg,M); % Demodulate unequalized signal.
demodmsg = pskdemod(yd,M); % Demodulate detected signal from equalizer.
[nnoeq,rnoeq] = symerr(demodmsg_noeq(trainlen+1:end),...
   msg(trainlen+1:end));
[neq,req] = symerr(demodmsg(trainlen+1:end),...
   msg(trainlen+1:end));
disp('Symbol error rates with and without equalizer:')
disp([req rnoeq])
```

The example goes on to determine how many errors occur in trying to recover the modulated message with and without the equalizer. The symbol error rates, below, show that the equalizer improves the performance significantly.

```
Symbol error rates with and without equalizer:
         0    0.3410
```

The example also creates a scatter plot that shows the signal before and after equalization, as well as the signal constellation for QPSK modulation. Notice

from the plot that the points of the equalized signal are clustered more closely around the points of the signal constellation.



Equalizing in Decision-Directed Mode

Decision-directed mode means that the equalizer uses a detected version of its output signal when adapting the weights. Adaptive equalizers typically start with a training sequence (as mentioned in “Equalizing Using a Training Sequence” on page 11-17) and switch to decision-directed mode after exhausting all symbols in the training sequence. CMA equalizers are an exception, using neither training mode nor decision-directed mode. For non-CMA equalizers, the `equalize` function operates in decision-directed mode when one of these conditions is true:

- The syntax does not include a training sequence.
- The equalizer has exhausted all symbols in the training sequence and still has more input symbols to process.

The example in "Equalizing Using a Training Sequence" on page 11-17 uses training mode when processing the first `trainlen` symbols of the input signal, and decision-directed mode thereafter. The example below discusses another scenario.

Example: Equalizing Multiple Times, Varying the Mode

If you invoke `equalize` multiple times with the same equalizer object to equalize a series of signal vectors, then you might use a training sequence the first time you call the function and omit the training sequence in subsequent calls. Each iteration of the `equalize` function after the first one operates completely in decision-directed mode. However, because the `ResetBeforeFiltering` property of the equalizer object is set to `0`, the `equalize` function uses the existing state information in the equalizer object when starting each iteration's equalization operation. As a result, the training affects all equalization operations, not just the first.

The code below illustrates this approach. Notice that the first call to `equalize` uses a training sequence as an input argument, while the second call to `equalize` omits a training sequence.

```
M = 4; % Alphabet size for modulation
msg = randint(1500,1,M); % Random message
modmsg = pskmod(msg,M); % Modulate using QPSK.
trainlen = 500; % Length of training sequence
chan = [.986; .845; .237; .123+.31i]; % Channel coefficients
filtmsg = filter(chan,1,modmsg); % Introduce channel distortion.

% Set up equalizer.
eqlms = lineareq(8, lms(0.01)); % Create an equalizer object.
eqlms.SigConst = pskmod([0:M-1],M); % Set signal constellation.
% Maintain continuity between calls to equalize.
eqlms.ResetBeforeFiltering = 0;

% Equalize the received signal, in pieces.
% 1. Process the training sequence.
s1 = equalize(eqlms,filtmsg(1:trainlen),modmsg(1:trainlen));
% 2. Process some of the data in decision-directed mode.
s2 = equalize(eqlms,filtmsg(trainlen+1:800));
% 3. Process the rest of the data in decision-directed mode.
```

```
s3 = equalize(eqlms,filtmsg(801:end));
s = [s1; s2; s3]; % Full output of equalizer
```

Delays from Equalization

For proper equalization using adaptive algorithms other than CMA, you should set the reference tap so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay between the modulator output and the equalizer output is equal to

```
(RefTap-1)/nSampPerSym
```

symbols. Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap in a linear equalizer, or the center tap of the forward filter in a decision-feedback equalizer.

For CMA equalizers, the expression above does not apply because a CMA equalizer has no reference tap. If you need to know the delay, you can find it empirically after the equalizer weights have converged. Use the `xcorr` function to examine cross-correlations of the modulator output and the equalizer output.

Techniques for Working with Delays

Here are some typical ways to take a delay of D into account by padding or truncating data:

- Pad your original data with D extra symbols at the end. Before comparing the original data with the received data, omit the first D symbols of the received data. In this approach, all the original data (not including the padding) is accounted for in the received data.
- Before comparing the original data with the received data, omit the last D symbols of the original data and the first D symbols of the received data. In this approach, some of the original symbols are not accounted for in the received data.

The example below illustrates the latter approach. For an example that illustrates both approaches in the context of interleavers, see "Delays of Convolutional Interleavers" on page 7-9.

```
M = 2; % Use BPSK modulation for this example.
msg = randint(1000,1,M); % Random data
modmsg = pskmod(msg,M); % Modulate.
trainlen = 100; % Length of training sequence
trainsig = modmsg(1:trainlen); % Training sequence

% Define an equalizer and equalize the received signal.
eqlin = lineareq(3,normlms(.0005,.0001),pskmod(0:M-1,M));
eqlin.RefTap = 2; % Set reference tap of equalizer.
[eqsig,detsym] = equalize(eqlin,modmsg,trainsig); % Equalize.

detmsg = pskdemod(detsym,M); % Demodulate the detected signal.

% Compensate for delay introduced by RefTap.
D = (eqlin.RefTap -1)/eqlin.nSampPerSym;
trunc_detmsg = detmsg(D+1:end); % Omit first D symbols of equalized data.
trunc_msg = msg(1:end-D); % Omit last D symbols.

% Compute bit error rate, ignoring training sequence.
[numerrs,ber] = biterr(trunc_msg(trainlen+1:end),...
   trunc_detmsg(trainlen+1:end))
```

The output is below.

```
numerrs =

     0


ber =

     0
```

Equalizing Using a Loop

If your data is partitioned into a series of vectors (that you process within a loop, for example), then you can invoke the `equalize` function multiple times, saving the equalizer's internal state information for use in a subsequent invocation. In particular, the final values of the `WeightInputs` and `Weights` properties in one equalization operation should be the initial values in the

next equalization operation. This section gives an example, followed by more general procedures for equalizing within a loop.

Example: Adaptive Equalization Within a Loop

The example below illustrates how to use `equalize` within a loop, varying the equalizer between iterations. Because the example is long, this discussion presents it in these steps:

- "Initializing Variables" on page 11-23
- "Simulating the System Using a Loop" on page 11-24

If you want to equalize iteratively while potentially changing equalizers between iterations, then the procedure in "Changing the Equalizer Between Iterations" on page 11-26 should help you generalize from this example to other cases.

Initializing Variables. The beginning of the example defines parameters and creates three equalizer objects:

- An RLS equalizer object.
- An LMS equalizer object.
- A variable, `eq_current`, that points to the equalizer object to use in the current iteration of the loop. Initially, this points to the RLS equalizer object. After the second iteration of the loop, `eq_current` is redefined to point to the LMS equalizer object.

```
% Set up parameters.
M = 16; % Alphabet size for modulation
sigconst = qammod(0:M-1,M); % Signal constellation for 16-QAM
chan = [1 0.45 0.3+0.2i]; % Channel coefficients

% Set up equalizers.
eqrls = lineareq(6, rls(0.99,0.1)); % Create an RLS equalizer object.
eqrls.SigConst = sigconst; % Set signal constellation.
eqrls.ResetBeforeFiltering = 0; % Maintain continuity between iterations.
eqlms = lineareq(6, lms(0.003)); % Create an LMS equalizer object.
eqlms.SigConst = sigconst; % Set signal constellation.
eqlms.ResetBeforeFiltering = 0; % Maintain continuity between iterations.
```

```
eq_current = eqrls; % Point to RLS for first iteration.
```

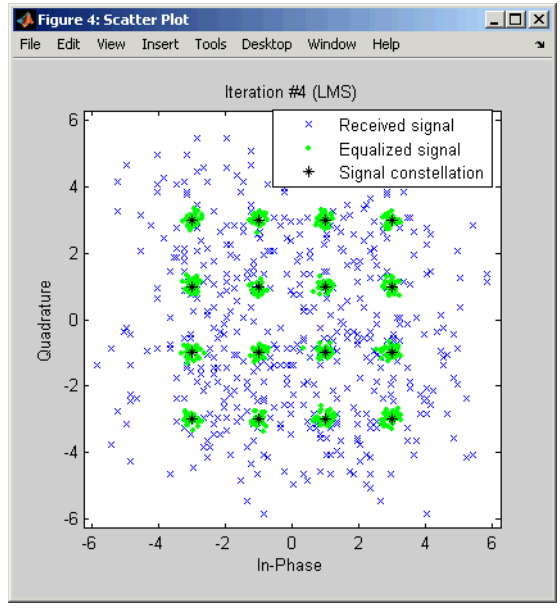
Simulating the System Using a Loop. The next portion of the example is a loop that

- Generates a signal to transmit and selects a portion to use as a training sequence in the first iteration of the loop
- Introduces channel distortion
- Equalizes the distorted signal using the chosen equalizer for this iteration, retaining the final state and weights for later use
- Plots the distorted and equalized signals, for comparison
- Switches to an LMS equalizer between the second and third iterations

```
% Main loop
for jj = 1:4
   msg = randint(500,1,M); % Random message
   modmsg = qammod(msg,M); % Modulate using 8-QAM.

   % Set up training sequence for first iteration.
   if jj == 1
      ltr = 200; trainsig = modmsg(1:ltr);
   else
      % Use decision-directed mode after first iteration.
      ltr = 0; trainsig = [];
   end

   % Introduce channel distortion.
   filtmsg = filter(chan,1,modmsg);

   % Equalize the received signal.
   s = equalize(eq_current,filtmsg,trainsig);

   % Plot signals.
   h = scatterplot(filtmsg(ltr+1:end),1,0,'bx'); hold on;
   scatterplot(s(ltr+1:end),1,0,'g.',h);
   scatterplot(sigconst,1,0,'k*',h);
   legend('Received signal','Equalized signal','Signal constellation');
   title(['Iteration #' num2str(jj) ' (' eq_current.AlgType ')']);
```

```
    hold off;

    % Switch from RLS to LMS after second iteration.
    if jj == 2
       eqlms.WeightInputs = eq_current.WeightInputs; % Copy final inputs.
       eqlms.Weights = eq_current.Weights; % Copy final weights.
       eq_current = eqlms; % Make eq_current point to eqlms.
    end
end
```

The example produces one scatter plot for each iteration, indicating the iteration number and the adaptive algorithm in the title. A sample plot is below. Your plot might look different because the example uses random numbers.



Procedures for Equalizing Within a Loop

This section describes two procedures for equalizing within a loop. The first procedure uses the same equalizer in each iteration, while the second is useful if you want to change the equalizer between iterations.

Using the Same Equalizer in Each Iteration. The typical procedure for using `equalize` within a loop is as follows:

1. Before the loop starts, create the equalizer object that you want to use in the first iteration of the loop.
2. Set the equalizer object's `ResetBeforeFiltering` property to `0` to maintain continuity between successive invocations of `equalize`.
3. Inside the loop, invoke `equalize` using a syntax like one of these:

   ```
   y = equalize(eqz,x,trainsig);
   y = equalize(eqz,x);
   ```

   The `equalize` function updates the state and weights of the equalizer at the end of the current iteration. In the next iteration, the function continues from where it finished in the previous iteration because `ResetBeforeFiltering` is set to `0`.

This procedure is similar to the one used in "Example: Equalizing Multiple Times, Varying the Mode" on page 11-20. That example uses `equalize` multiple times but not within a loop.

Changing the Equalizer Between Iterations. In some applications, you might want to modify the adaptive algorithm between iterations. For example, you might use a CMA equalizer for the first iteration and an LMS or RLS equalizer in subsequent iterations. The procedure below gives one way to accomplish this, roughly following the example in "Example: Adaptive Equalization Within a Loop" on page 11-23:

1. Before the loop starts, create the different kinds of equalizer objects that you want to use during various iterations of the loop.

   For example, create one CMA equalizer object, `eqcma`, and one LMS equalizer object, `eqlms`.
2. For each equalizer object, set the `ResetBeforeFiltering` property to `0` to maintain continuity between successive invocations of `equalize`.

3. Create a variable `eq_current` that points to the equalizer object that you want to use for the first iteration. Use = to establish the connection so that the two objects get updated together, as below.

   ```
   eq_current = eqcma; % Point to eqcma.
   ```

   The purpose of `eq_current` is to represent the equalizer used in each iteration, where you can switch equalizers from one iteration to the next by using a command like `eq_current = eqlms`. The example illustrates this approach near the end of its loop.

4. Inside the loop, perform these steps:

   a. Invoke `equalize` using a syntax like one of these:

      ```
      y = equalize(eq_current,x,trainsig);
      y = equalize(eq_current,x);
      ```

   b. Copy the values of the `WeightInputs` and `Weights` properties from `eq_current` to the equalizer object that you want to use for the next iteration. Use dot notation. For example,

      ```
      eqlms.WeightInputs = eq_current.WeightInputs;
      eqlms.Weights = eq_current.Weights;
      ```

   c. Redefine `eq_current` to point to the equalizer object that you want to use for the next iteration, using =. Now `eq_current` is set up for the next iteration, because it represents the new kind of equalizer but retains the old values for the state and weights.

The reason for creating multiple equalizer objects and then copying the state and weights, instead of simply changing the equalizer class or adaptive algorithm in a single equalizer object, is that the class and adaptive algorithm properties of an equalizer object are fixed.

Using MLSE Equalizers

The `mlseeq` function uses the Viterbi algorithm to equalize a signal through a dispersive channel. The function receives a baseband linearly modulated input signal and outputs the maximum likelihood sequence estimate of the signal, using an estimate of the channel modeled as a finite input response (FIR) filter.

The function decodes the received signal using these steps:

1. Applies the FIR filter, corresponding to the channel estimate, to the symbols in the input signal.
2. Uses the Viterbi algorithm to compute the traceback paths and the state metric, which are the numbers assigned to the symbols at each step of the Viterbi algorithm. The metrics are based on Euclidean distance.
3. Outputs the maximum likelihood sequence estimate of the signal, as a sequence of complex numbers corresponding to the constellation points of the modulated signal.

An MLSE equalizer yields the best possible performance, in theory, but is computationally intensive. These topics describe how to use the MLSE equalizer capabilities in this toolbox:

- "Equalizing a Vector Signal" on page 11-28
- "Equalizing in Continuous Operation Mode" on page 11-29
- "Using a Preamble or Postamble" on page 11-33

For background material about MLSE equalizers, see the works listed in "Selected Bibliography for Equalizers" on page 11-35.

Equalizing a Vector Signal

In its simplest form, the `mlseeq` function equalizes a vector of modulated data when you specify the estimated coefficients of the channel (modeled as an FIR filter), the signal constellation for the modulation type, and the traceback depth that you want the Viterbi algorithm to use. Larger values for

the traceback depth can improve the results from the equalizer but increase the computation time.

An example of the basic syntax for `mlseeq` is below.

```
M = 4; const = pskmod([0:M-1],M); % 4-PSK constellation
msg = pskmod([1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]',M); % Modulated message
chcoeffs = [.986; .845; .237; .12345+.31i]; % Channel coefficients
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
tblen =  10; % Traceback depth for equalizer
chanest = chcoeffs; % Assume the channel is known exactly.
msgEq = mlseeq(filtmsg,chanest,const,tblen,'rst'); % Equalize.
```

The `mlseeq` function has two operation modes:

- Continuous operation mode enables you to process a series of vectors using repeated calls to `mlseeq`, where the function saves its internal state information from one call to the next. To learn more, see “Equalizing in Continuous Operation Mode” on page 11-29.
- Reset operation mode enables you to specify a preamble and postamble that accompany your data. To learn more, see “Using a Preamble or Postamble” on page 11-33.

If you are not processing a series of vectors and do not need to specify a preamble or postamble, then the operation modes are nearly identical. However, they differ in that continuous operation mode incurs a delay, while reset operation mode does not. The example above could have used either mode, except that substituting continuous operation mode would have produced a delay in the equalized output. To learn more about the delay in continuous operation mode, see “Delays in Continuous Operation Mode” on page 11-30.

Equalizing in Continuous Operation Mode

If your data is partitioned into a series of vectors (that you process within a loop, for example), then continuous operation mode is an appropriate way to use the `mlseeq` function. In continuous operation mode, `mlseeq` can save its internal state information for use in a subsequent invocation and can initialize using previously stored state information. To choose continuous operation mode, use `'cont'` as an input argument when invoking `mlseeq`.

Note Continuous operation mode incurs a delay, as described in "Delays in Continuous Operation Mode" on page 11-30. Also, continuous operation mode cannot accommodate a preamble or postamble.

Procedure for Continuous Operation Mode

The typical procedure for using continuous mode within a loop is as follows:

1. Before the loop starts, create three empty matrix variables (for example, `sm`, `ts`, `ti`) that will eventually store the state metrics, traceback states, and traceback inputs for the equalizer.

2. Inside the loop, invoke `mlseeq` using a syntax like

   ```
   [y,sm,ts,ti] = mlseeq(x,chcoeffs,const,tblen,'cont',nsamp,sm,ts,ti);
   ```

   Using `sm`, `ts`, and `ti` as input arguments causes `mlseeq` to continue from where it finished in the previous iteration. Using `sm`, `ts`, and `ti` as output arguments causes `mlseeq` to update the state information at the end of the current iteration. In the first iteration, `sm`, `ts`, and `ti` start as empty matrices, so the first invocation of the `mlseeq` function initializes the metrics of all states to 0.

Delays in Continuous Operation Mode

Continuous operation mode with a traceback depth of `tblen` incurs an output delay of `tblen` symbols. This means that the first `tblen` output symbols are unrelated to the input signal, while the last `tblen` input symbols are unrelated to the output signal. For example, the command below uses a traceback depth of 3, and the first 3 output symbols are unrelated to the input signal of `ones(1,10)`.

```
y = mlseeq(ones(1,10),1,[-7:2:7],3,'cont')
y =

    -7    -7    -7     1     1     1     1     1     1     1
```

Keeping track of delays from different portions of a communication system is important, especially if you compare signals to compute error rates. The

example in "Example: Continuous Operation Mode" on page 11-31 illustrates how to take the delay into account when computing an error rate.

Example: Continuous Operation Mode

The example below illustrates the procedure for using continuous operation mode within a loop. Because the example is long, this discussion presents it in multiple steps:

- "Initializing Variables" on page 11-31
- "Simulating the System Using a Loop" on page 11-31
- "Computing an Error Rate and Plotting Results" on page 11-32

Initializing Variables. The beginning of the example defines parameters, initializes the state variables `sm`, `ts`, and `ti`, and initializes variables that accumulate results from each iteration of the loop.

```
n = 200; % Number of symbols in each iteration
numiter = 25; % Number of iterations
M = 4; % Use 4-PSK modulation.
const = pskmod(0:M-1,M); % PSK constellation
chcoeffs = [1 ; 0.25]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
tblen = 10; % Traceback depth for equalizer
nsamp = 1; % Number of input samples per symbol
sm = []; ts = []; ti = []; % Initialize equalizer data.
% Initialize cumulative results.
fullmodmsg = []; fullfiltmsg = []; fullrx = [];
```

Simulating the System Using a Loop. The middle portion of the example is a loop that generates random data, modulates it using baseband PSK modulation, and filters it. Finally, `mlseeq` equalizes the filtered data. The loop also updates the variables that accumulate results from each iteration of the loop.

```
for jj = 1:numiter
   msg = randint(n,1,M); % Random signal vector
   modmsg = pskmod(msg,M); % PSK-modulated signal
   filtmsg = filter(chcoeffs,1,modmsg); % Filtered signal
```

```
    % Equalize, initializing from where the last iteration
    % finished, and remembering final data for the next iteration.
    [rx sm ts ti] = mlseeq(filtmsg,chanest,const,tblen,...
       'cont',nsamp,sm,ts,ti);

    % Update vectors with cumulative results.
    fullmodmsg = [fullmodmsg; modmsg];
    fullfiltmsg = [fullfiltmsg; filtmsg];
    fullrx = [fullrx; rx];
end
```

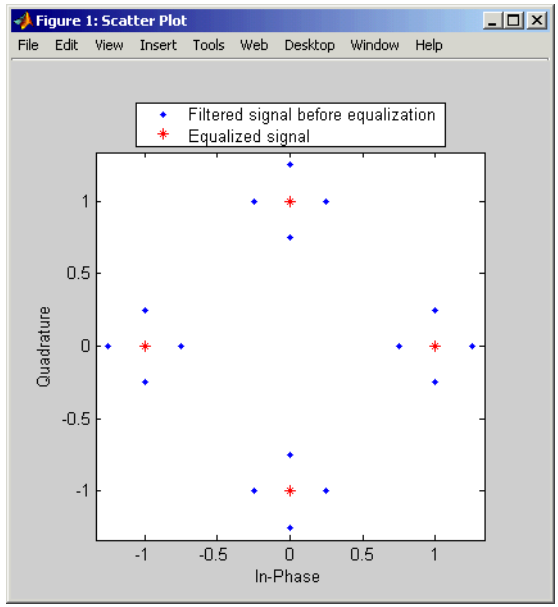
Computing an Error Rate and Plotting Results. The last portion of the example computes the symbol error rate from all iterations of the loop. Notice that the `symerr` function compares selected portions of the received and transmitted signals, not the entire signals. Because continuous operation mode incurs a delay whose length in samples is the traceback depth (`tblen`) of the equalizer, it is necessary to exclude the first `tblen` samples from the received signal and the last `tblen` samples from the transmitted signal. Excluding samples that represent the delay of the equalizer ensures that the symbol error rate calculation compares samples from the received and transmitted signals that are meaningful and that truly correspond to each other.

The example also plots the signal before and after equalization in a scatter plot. The points in the equalized signal coincide with the points of the ideal signal constellation for 4-PSK.

```
% Compute total number of symbol errors. Take the delay into account.
numsymerrs = symerr(fullrx(tblen+1:end),fullmodmsg(1:end-tblen))

% Plot signal before and after equalization.
h = scatterplot(fullfiltmsg); hold on;
scatterplot(fullrx,1,0,'r*',h);
legend('Filtered signal before equalization','Equalized signal',...
   'Location','NorthOutside');
hold off;
```

The output and plot are below.

```
numsymerrs =

     0
```



Using a Preamble or Postamble

Some systems include a sequence of known symbols at the beginning or end of a set of data. The known sequence at the beginning or end is called a preamble or postamble, respectively. The `mlseeq` function can accommodate a preamble and postamble that are already incorporated into its input signal. When you invoke the function, you specify the preamble and postamble as integer vectors that represent the sequence of known symbols by indexing into the signal constellation vector. For example, a preamble vector of `[1 4 4]` and a 4-PSK signal constellation of `[1 j -1 -j]` indicate that the modulated signal begins with `[1 -j -j]`.

If your system uses a preamble without a postamble, then use a postamble vector of `[]` when invoking `mlseeq`. Similarly, if your system uses a postamble without a preamble, then use a preamble vector of `[]`.

Example: Using a Preamble

The example below illustrates how to accommodate a preamble when using `mlseeq`. Notice that the same preamble symbols appear at the beginning of the message vector and in the syntax for `mlseeq`. If you wanted to use a postamble, then you could append it to the message vector and also include it as the last input argument for `mlseeq`. In this example, however, the postamble input in the `mlseeq` syntax is an empty vector because the system uses no postamble.

```
M = 4; % Use 4-PSK modulation.
const = pskmod(0:3,4); % PSK constellation
tblen = 16; % Traceback depth for equalizer

preamble = [3; 1]; % Expected preamble, as integers
msgIdx = randint(98,1,M); % Random symbols
msgIdx = [preamble; msgIdx]; % Include preamble at the beginning.
msg = pskmod(msgIdx,M); % Modulated message
chcoeffs = [.623; .489+.234i; .398i; .21]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
d = mlseeq(filtmsg,chanest,const,tblen,'rst',1,preamble,[]);

[nsymerrs ser] = symerr(msg,d) % Symbol error rate
```

The output is below.

```
nsymerrs =

     0


ser =

     0
```

Selected Bibliography for Equalizers

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

[5] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, Wiley, 1996.

Galois Field Computations

A *Galois field* is an algebraic field that has a finite number of members. Galois fields having 2^m members are used in error-control coding and are denoted GF(2^m). This chapter describes how to work with fields that have 2^m members, where m is an integer between 1 and 16. The sections in this chapter are as follows.

"Speed and Nondefault Primitive Polynomials" (p. 12-38)	Accelerating computations involving Galois field elements expressed relative to a nondefault primitive polynomial
"Selected Bibliography for Galois Fields" (p. 12-40)	Works containing background information about Galois fields or their use in error-control coding

If you need to use Galois fields having an odd number of elements, see "Galois Fields of Odd Characteristic" in the online documentation for the Communications Toolbox.

For more details about specific functions that process arrays of Galois field elements, see the online reference entries in the documentation for MATLAB or for the Communications Toolbox. MATLAB functions whose generalization to Galois fields is straightforward to describe do not have reference entries in this manual because the entries would be identical to those in the MATLAB manual.

Galois Field Terminology

The discussion of Galois fields in this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [4]:

- A *primitive element* of $GF(2^m)$ is a cyclic generator of the group of nonzero elements of $GF(2^m)$. This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power.
- A *primitive polynomial* for $GF(2^m)$ is the minimal polynomial of some primitive element of $GF(2^m)$. That is, it is the binary-coefficient polynomial of smallest nonzero degree having a certain primitive element as a root in $GF(2^m)$. As a consequence, a primitive polynomial has degree m and is irreducible.

The definitions imply that a primitive element is a root of a corresponding primitive polynomial.

Representing Elements of Galois Fields

This section describes how to create a *Galois array*, which is a MATLAB expression that represents elements of a Galois field. This section also describes how MATLAB interprets the numbers that you use in the representation, and includes several examples. The topics are

Creating a Galois Array

To begin working with data from a Galois field GF(`2^m`), you must set the context by associating the data with crucial information about the field. The `gf` function performs this association and creates a Galois array in MATLAB. This function accepts as inputs

- The Galois field data, `x`, which is a MATLAB array whose elements are integers between 0 and `2^m-1`.
- (*Optional*) An integer, `m`, that indicates that `x` is in the field GF(`2^m`). Valid values of `m` are between 1 and 16. The default is 1, which means that the field is GF(2).
- (*Optional*) A positive integer that indicates which primitive polynomial for GF(`2^m`) you are using in the representations in `x`. If you omit this input argument, then `gf` uses a default primitive polynomial for GF(`2^m`). For information about this argument, see "Specifying the Primitive Polynomial" on page 12-9.

The output of the `gf` function is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate the variable, MATLAB works within the Galois field you have specified. For example, if you apply the `log` function to a Galois array, then

MATLAB computes the logarithm in the Galois field and *not* in the field of real or complex numbers.

When MATLAB Implicitly Creates a Galois Array

Some operations on Galois arrays require multiple arguments. If you specify one argument that is a Galois array and another that is an ordinary MATLAB array, then MATLAB interprets both as Galois arrays in the same field. That is, it implicitly invokes the `gf` function on the ordinary MATLAB array. This implicit invocation simplifies your syntax because you can omit some references to the `gf` function. For an example of the simplification, see "Example: Addition and Subtraction" on page 12-14.

Example: Creating Galois Field Variables

The code below creates a row vector whose entries are in the field GF(4), and then adds the row to itself.

```
x = 0:3; % A row vector containing integers
m = 2; % Work in the field GF(2^2), or, GF(4).
a = gf(x,m) % Create a Galois array in GF(2^m).

b = a + a % Add a to itself, creating b.
```

The output is

```
a = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     1     2     3


b = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     0     0     0
```

The output shows the values of the Galois arrays named `a` and `b`. Notice that each output section indicates

- The field containing the variable, namely, GF(2^2) = GF(4).
- The primitive polynomial for the field. In this case, it is the toolbox's default primitive polynomial for GF(4).
- The array of Galois field values that the variable contains. In particular, the array elements in `a` are exactly the elements of the vector `x`, while the array elements in `b` are four instances of the zero element in GF(4).

The command that creates `b` shows how, having defined the variable `a` as a Galois array, you can add `a` to itself by using the ordinary + operator. MATLAB performs the vectorized addition operation in the field GF(4). Notice from the output that

- Compared to `a`, `b` is in the same field and uses the same primitive polynomial. It is not necessary to indicate the field when defining the sum, `b`, because MATLAB remembers that information from the definition of the addends, `a`.
- The array elements of `b` are zeros because the sum of any value with itself, in a Galois field *of characteristic two*, is zero. This result differs from the sum `x + x`, which represents an addition operation in the infinite field of integers.

Example: Representing Elements of GF(8)

To illustrate what the array elements in a Galois array mean, the table below lists the elements of the field GF(8) as integers and as polynomials in a primitive element, A. The table should help you interpret a Galois array like

```
gf8 = gf([0:7],3); % Galois vector in GF(2^3)
```

Integer Representation	Binary Representation	Element of GF(8)
0	000	0
1	001	1
2	010	A
3	011	A + 1

Integer Representation	Binary Representation	Element of GF(8)
4	100	A^2
5	101	$A^2 + 1$
6	110	$A^2 + A$
7	111	$A^2 + A + 1$

How Integers Correspond to Galois Field Elements

Building on the GF(8) example above, this section explains the interpretation of array elements in a Galois array in greater generality. The field GF(`2^m`) has `2^m` distinct elements, which this toolbox labels as 0, 1, 2,..., `2^m-1`. These integer labels correspond to elements of the Galois field via a polynomial expression involving a primitive element of the field. More specifically, each integer between 0 and `2^m-1` has a binary representation in `m` bits. Using the bits in the binary representation as coefficients in a polynomial, where the least significant bit is the constant term, leads to a binary polynomial whose order is at most `m-1`. Evaluating the binary polynomial at a primitive element of GF(`2^m`) leads to an element of the field.

Conversely, any element of GF(`2^m`) can be expressed as a binary polynomial of order at most `m-1`, evaluated at a primitive element of the field. The `m`-tuple of coefficients of the polynomial corresponds to the binary representation of an integer between 0 and `2^m`.

Below is a symbolic illustration of the correspondence of an integer X, representable in binary form, with a Galois field element. Each b_k is either zero or one, while A is a primitive element.

$$X = b_{m-1} \cdot 2^{m-1} + \cdots + b_2 \cdot 4 + b_1 \cdot 2 + b_0$$
$$\leftrightarrow b_{m-1} \cdot A^{m-1} + \cdots + b_2 \cdot A^2 + b_1 \cdot A + b_0$$

Example: Representing a Primitive Element

The code below defines a variable `alph` that represents a primitive element of the field GF(2^4).

```
m = 4; % Or choose any positive integer value of m.
alph = gf(2,m) % Primitive element in GF(2^m)
```

The output is

```
alph = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     2
```

The Galois array `alph` represents a primitive element because of the correspondence between

- The integer 2, specified in the `gf` syntax
- The binary representation of 2, which is 10 (or 0010 using four bits)
- The polynomial A + 0, where A is a primitive element in this field (or $0A^3 + 0A^2 + A + 0$ using the four lowest powers of A)

Primitive Polynomials and Element Representations

This section builds on the discussion in "Creating a Galois Array" on page 12-4 by describing how to specify your own primitive polynomial when you create a Galois array. The topics are

- "Specifying the Primitive Polynomial" on page 12-9
- "Finding Primitive Polynomials" on page 12-10
- "Effect of Nondefault Primitive Polynomials on Numerical Results" on page 12-11

If you perform many computations using a nondefault primitive polynomial, then see "Speed and Nondefault Primitive Polynomials" on page 12-38 as well.

Specifying the Primitive Polynomial

The discussion in "How Integers Correspond to Galois Field Elements" on page 12-7 refers to a primitive element, which is a root of a primitive polynomial of the field. When you use the `gf` function to create a Galois array, the function interprets the integers in the array with respect to a specific default primitive polynomial for that field, unless you explicitly provide a different primitive polynomial. A list of the default primitive polynomials is on the reference page for the `gf` function.

To specify your own primitive polynomial when creating a Galois array, use a syntax like

```
c = gf(5,4,25) % 25 indicates the primitive polynomial for GF(16).
```

instead of

```
c1= gf(5,4); % Use default primitive polynomial for GF(16).
```

The extra input argument, `25` in this case, specifies the primitive polynomial for the field GF(`2^m`) in a way similar to the representation described in "How Integers Correspond to Galois Field Elements" on page 12-7. In this case, the integer 25 corresponds to a binary representation of 11001, which in turn corresponds to the polynomial $D^4 + D^3 + 1$.

Note When you specify the primitive polynomial, the input argument must have a binary representation using exactly `m+1` bits, not including unnecessary leading zeros. In other words, a primitive polynomial for GF(`2^m`) always has order `m`.

When you use an input argument to specify the primitive polynomial, the output reflects your choice by showing the integer value as well as the polynomial representation.

```
d = gf([1 2 3],4,25)
```

```
d = GF(2^4) array. Primitive polynomial = D^4+D^3+1 (25 decimal)

Array elements =

     1     2     3
```

Note After you have defined a Galois array, you cannot change the primitive polynomial with respect to which MATLAB interprets the array elements.

Finding Primitive Polynomials

You can use the `primpoly` function to find primitive polynomials for GF(`2^m`) and the `isprimitive` function to determine whether a polynomial is primitive for GF(`2^m`). The code below illustrates.

```
m = 4;
defaultprimpoly = primpoly(m) % Default primitive poly for GF(16)
allprimpolys = primpoly(m,'all') % All primitive polys for GF(16)
i1 = isprimitive(25) % Can 25 be the prim_poly input in gf(...)?
i2 = isprimitive(21) % Can 21 be the prim_poly input in gf(...)?
```

The output is below.

```
Primitive polynomial(s) =

D^4+D^1+1

defaultprimpoly =

    19

Primitive polynomial(s) =

D^4+D^1+1
D^4+D^3+1
```

```
allprimpolys =

    19
    25

i1 =

     1

i2 =

     0
```

Effect of Nondefault Primitive Polynomials on Numerical Results

Most fields offer multiple choices for the primitive polynomial that helps define the representation of members of the field. When you use the `gf` function, changing the primitive polynomial changes the interpretation of the array elements and, in turn, changes the results of some subsequent operations on the Galois array. For example, exponentiation of a primitive element makes it easy to see how the primitive polynomial affects the representations of field elements.

```
a11 = gf(2,3); % Use default primitive polynomial of 11.
a13 = gf(2,3,13); % Use D^3+D^2+1 as the primitive polynomial.
z = a13.^3 + a13.^2 + 1 % 0 because a13 satisfies the equation
nz = a11.^3 + a11.^2 + 1 % Nonzero. a11 does not satisfy equation.
```

The output below shows that when the primitive polynomial has integer representation 13, the Galois array satisfies a certain equation. By contrast, when the primitive polynomial has integer representation 11, the Galois array fails to satisfy the equation.

```
z = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     0
```

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

Array elements =

     6
```

The output when you try this example might also include a warning about lookup tables. This is normal if you did not use the `gftable` function to optimize computations involving a nondefault primitive polynomial of 13.

Arithmetic in Galois Fields

You can perform arithmetic operations on Galois arrays by using familiar MATLAB operators, listed in the table below. Whenever you operate on a pair of Galois arrays, both arrays must be in the same Galois field.

Operation	Operator
Addition	`+`
Subtraction	`-`
Elementwise multiplication	`.*`
Matrix multiplication	`*`
Elementwise left division	`./`
Elementwise right division	`.\`
Matrix left division	`/`
Matrix right division	`\`
Elementwise exponentiation	`.^`
Elementwise logarithm	`log()`
Exponentiation of a square Galois matrix by a scalar integer	`^`

Note For multiplication and division of polynomials over a Galois field, see "Addition and Subtraction of Polynomials" on page 12-30.

Examples of these operations are in the sections that follow:

• "Example: Elementwise Logarithm" on page 12-18

Example: Addition and Subtraction

The code below adds two Galois arrays to create an addition table for GF(8). Addition uses the ordinary + operator. The code below also shows how to index into the array `addtb` to find the result of adding 1 to the elements of GF(8).

```
m = 3;
e = repmat([0:2^m-1],2^m,1);
f = gf(e,m); % Create a Galois array.
addtb = f + f' % Add f to its own matrix transpose.

addone = addtb(2,:); % Assign 2nd row to the Galois vector addone.
```

The output is below.

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

Array elements =

     0     1     2     3     4     5     6     7
     1     0     3     2     5     4     7     6
     2     3     0     1     6     7     4     5
     3     2     1     0     7     6     5     4
     4     5     6     7     0     1     2     3
     5     4     7     6     1     0     3     2
     6     7     4     5     2     3     0     1
     7     6     5     4     3     2     1     0
```

As an example of reading this addition table, the (7,4) entry in the `addtb` array shows that `gf(6,3)` plus `gf(3,3)` equals `gf(5,3)`. Equivalently, the element A^2+A plus the element A+1 equals the element A^2+1. The equivalence arises from the binary representation of 6 as 110, 3 as 011, and 5 as 101.

The subtraction table, which you can obtain by replacing + by -, would be the same as `addtb`. This is because subtraction and addition are identical operations in a field *of characteristic two*. In fact, the zeros along the main diagonal of `addtb` illustrate this fact for GF(8).

Simplifying the Syntax

The code below illustrates scalar expansion and the implicit creation of a Galois array from an ordinary MATLAB array. The Galois arrays `h` and `h1` are identical, but the creation of `h` uses a simpler syntax.

```
g = gf(ones(2,3),4); % Create a Galois array explicitly.
h = g + 5; % Add gf(5,4) to each element of g.
h1 = g + gf(5*ones(2,3),4) % Same as h.
```

The output is below.

```
h1 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     4     4     4
     4     4     4
```

Notice that 1+5 is reported as 4 in the Galois field. This is true because the 5 represents the polynomial expression A^2+1, and $1+(A^2+1)$ in GF(16) is A^2. Furthermore, the integer that represents the polynomial expression A^2 is 4.

Example: Multiplication

The example below multiplies individual elements in a Galois array using the `.*` operator. It then performs matrix multiplication using the `*` operator. The elementwise multiplication produces an array whose size matches that of the inputs. By contrast, the matrix multiplication produces a Galois scalar because it is the matrix product of a row vector with a column vector.

```
m = 5;
row1 = gf([1:2:9],m); row2 = gf([2:2:10],m);
col = row2'; % Transpose to create a column array.
ep = row1 .* row2; % Elementwise product.
mp = row1 * col; % Matrix product.
```

Multiplication Table for GF(8)

As another example, the code below multiplies two Galois vectors using matrix multiplication. The result is a multiplication table for GF(8).

```
m = 3;
els = gf([0:2^m-1]',m);
multb = els * els' % Multiply els by its own matrix transpose.
```

The output is below.

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

Array elements =

     0     0     0     0     0     0     0     0
     0     1     2     3     4     5     6     7
     0     2     4     6     3     1     7     5
     0     3     6     5     7     4     1     2
     0     4     3     7     6     2     5     1
     0     5     1     4     2     7     3     6
     0     6     7     1     5     3     2     4
     0     7     5     2     1     6     4     3
```

Example: Division

The examples below illustrate the four division operators in a Galois field by computing multiplicative inverses of individual elements and of an array. You can also compute inverses using `inv` or using exponentiation by -1.

Elementwise Division

This example divides 1 by each of the individual elements in a Galois array using the `./` and `.\` operators. These two operators differ only in their sequence of input arguments. Each quotient vector lists the multiplicative inverses of the nonzero elements of the field. In this example, MATLAB expands the scalar 1 to the size of `nz` before computing; alternatively, you can use as arguments two arrays of the same size.

```
m = 5;
nz = gf([1:2^m-1],m); % Nonzero elements of the field
inv1 = 1 ./ nz; % Divide 1 by each element.
inv2 = nz .\ 1; % Obtain same result using .\ operator.
```

Matrix Division

This example divides the identity array by the square Galois array `mat` using the `/` and `\` operators. Each quotient matrix is the multiplicative inverse of `mat`. Notice how the transpose operator (`'`) appears in the equivalent operation using `\`. For square matrices, the sequence of transpose operations is unnecessary, but for nonsquare matrices, it is necessary.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minv1 = eye(3) / mat; % Compute matrix inverse.
minv2 = (mat' \ eye(3)')'; % Obtain same result using \ operator.
```

Example: Exponentiation

The examples below illustrate how to compute integer powers of a Galois array. To perform matrix exponentiation on a Galois array, you must use a square Galois array as the base and an ordinary (not Galois) integer scalar as the exponent.

Elementwise Exponentiation

This example computes powers of a primitive element, A, of a Galois field. It then uses these separately computed powers to evaluate the default primitive polynomial at A. The answer of zero shows that A is a root of the primitive polynomial. Notice that the `.^` operator exponentiates each array element independently.

```
m = 3;
av = gf(2*ones(1,m+1),m); % Row containing primitive element
expa = av .^ [0:m]; % Raise element to different powers.
evp = expa(4)+expa(2)+expa(1) % Evaluate D^3 + D + 1.
```

The output is below.

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

Array elements =

     0
```

Matrix Exponentiation

This example computes the inverse of a square matrix by raising the matrix to the power -1. It also raises the square matrix to the powers 2 and -2.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minvs = mat ^ (-1); % Matrix inverse
matsq = mat^2; % Same as mat * mat
matinvssq = mat^(-2); % Same as minvs * minvs
```

Example: Elementwise Logarithm

The code below computes the logarithm of the elements of a Galois array. The output indicates how to express each *nonzero* element of GF(8) as a power of the primitive element. The logarithm of the zero element of the field is undefined.

```
gf8_nonzero = gf([1:7],3); % Vector of nonzero elements of GF(8)
expformat = log(gf8_nonzero) % Logarithm of each element
```

The output is

```
expformat =

     0     1     3     2     6     4     5
```

As an example of how to interpret the output, consider the last entry in each vector in this example. You can infer that the element `gf(7,3)` in GF(8) can be expressed as either

- A^5, using the last element of `expformat`
- A^2+A+1, using the binary representation of 7 as 111. See “Example: Representing Elements of GF(8)” on page 12-6 for more details.

Logical Operations in Galois Fields

You can apply logical tests to Galois arrays and obtain a logical array. Some important types of tests are testing for equality of two Galois arrays and testing for nonzero values in a Galois array.

Testing for Equality

To compare corresponding elements of two Galois arrays that have the same size, use the operators == and ~=. The result is a logical array, each element of which indicates the truth or falsity of the corresponding elementwise comparison. If you use the same operators to compare a scalar with a Galois array, then MATLAB compares the scalar with each element of the array, producing a logical array of the same size.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg1 = (r1 .* r2 == [1 1 1]) % Does each element equal one?
lg2 = (r1 .* r2 == 1) % Same as above, using scalar expansion
lg3 = (r1 ~= r2) % Does each element differ from its inverse?
```

The output is below.

```
lg1 =

     1     1     1


lg2 =

     1     1     1


lg3 =

     0     1     1
```

Comparison of isequal and ==

To compare entire arrays and obtain a logical *scalar* result rather than a logical array, you can use the built-in `isequal` function. Note, however, that

`isequal` uses strict rules for its comparison, and returns a value of `0` (false) if you compare

- A Galois array with an ordinary MATLAB array, even if the values of the underlying array elements match
- A scalar with a nonscalar array, even if all elements in the array match the scalar

The example below illustrates this difference between == and `isequal`.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg4 = isequal(r1 .* r2, [1 1 1]); % False
lg5 = isequal(r1 .* r2, gf(1,m)); % False
lg6 = isequal(r1 .* r2, gf([1 1 1],m)); % True
```

Testing for Nonzero Values

To test for nonzero values in a Galois vector, or in the columns of a Galois array that has more than one row, use the `any` or `all` function. These two functions behave just like the ordinary MATLAB functions `any` and `all`, except that they consider only the underlying array elements while ignoring information about which Galois field the elements are in. Examples are below.

```
m = 3; randels = gf(randint(6,1,2^m),m);
if all(randels) % If all elements are invertible
    invels = randels .\ 1; % Compute inverses of elements.
else
    disp('At least one element was not invertible.');
end
alph = gf(2,4);
poly = 1 + alph + alph^3;
if any(poly) % If poly contains a nonzero value
    disp('alph is not a root of 1 + D + D^3.');
end
code = rsenc(gf([0:4;3:7],3),7,5); % Each row is a codeword.
if all(code,2) % Is each row entirely nonzero?
    disp('Both codewords are entirely nonzero.');
else
    disp('At least one codeword contains a zero.');
end
```

Matrix Manipulation in Galois Fields

Some basic operations that you would perform on an ordinary MATLAB array are available for Galois arrays. This section illustrates how to perform basic manipulations and how to get basic information.

Basic Manipulations of Galois Arrays

Basic array operations on Galois arrays are in the table below. The functionality of these operations is analogous to the MATLAB operations having the same syntax.

Operation	Syntax
Index into array, possibly using colon operator instead of a vector of explicit indices	`a(vector)` or `a(vector,vector1)`, where `vector` and/or `vector1` can be `":"` instead of a vector
Transpose array	`a'`
Concatenate matrices	`[a,b]` or `[a;b]`
Create array having specified diagonal elements	`diag(vector)` or `diag(vector,k)`
Extract diagonal elements	`diag(a)` or `diag(a,k)`
Extract lower triangular part	`tril(a)` or `tril(a,k)`
Extract upper triangular part	`triu(a)` or `triu(a,k)`
Change shape of array	`reshape(a,k1,k2)`

The code below uses some of these syntaxes.

```
m = 4; a = gf([0:15],m);
a(1:2) = [13 13]; % Replace some elements of the vector a.
b = reshape(a,2,8); % Create 2-by-8 matrix.
c = [b([1 1 2],1:3); a(4:6)]; % Create 4-by-3 matrix.
d = [c, a(1:4)']; % Create 4-by-4 matrix.
dvec = diag(d); % Extract main diagonal of d.
dmat = diag(a(5:9)); % Create 5-by-5 diagonal matrix
dtril = tril(d); % Extract upper and lower triangular
```

```
dtriu = triu(d); % parts of d.
```

Basic Information About Galois Arrays

You can determine the length of a Galois vector or the size of any Galois array using the `length` and `size` functions. The functionality for Galois arrays is analogous to that of the MATLAB operations on ordinary arrays, except that the output arguments from `size` and `length` are always integers, not Galois arrays. The code below illustrates the use of these functions.

```
m = 4; e = gf([0:5],m); f = reshape(e,2,3);
lne = length(e); % Vector length of e
szf = size(f); % Size of f, returned as a two-element row
[nr,nc] = size(f); % Size of f, returned as two scalars
nc2 = size(f,2); % Another way to compute number of columns
```

Positions of Nonzero Elements

Another type of information you might want to determine from a Galois array is the positions of nonzero elements. For an ordinary MATLAB array, you might use the `find` function. However, for a Galois array you should use `find` in conjunction with the ~= operator, as illustrated.

```
x = [0 1 2 1 0 2]; m = 2; g = gf(x,m);
nzx = find(x); % Find nonzero values in the ordinary array x.
nzg = find(g~=0); % Find nonzero values in the Galois array g.
```

Linear Algebra in Galois Fields

You can do linear algebra in a Galois field using Galois arrays. Important categories of computations are inverting matrices, computing determinants, computing ranks, factoring square matrices, and solving linear equations.

Inverting Matrices and Computing Determinants

To invert a square Galois array, use the `inv` function. Related is the `det` function, which computes the determinant of a Galois array. Both `inv` and `det` behave like their ordinary MATLAB counterparts, except that they perform computations in the Galois field instead of in the field of complex numbers.

Note A Galois array is singular if and only if its determinant is exactly zero. It is not necessary to consider roundoff errors, as in the case of real and complex arrays.

The code below illustrates matrix inversion and determinant computation.

```
m = 4;
randommatrix = gf(randint(4,4,2^m),m);
gfid = gf(eye(4),m);
if det(randommatrix) ~= 0
    invmatrix = inv(randommatrix);
    check1 = invmatrix * randommatrix;
    check2 = randommatrix * invmatrix;
    if (isequal(check1,gfid) & isequal(check2,gfid))
        disp('inv found the correct matrix inverse.');
    end
else
    disp('The matrix is not invertible.');
end
```

The output from this example is either of these two messages, depending on whether the randomly generated matrix is nonsingular or singular.

```
inv found the correct matrix inverse.
The matrix is not invertible.
```

Computing Ranks

To compute the rank of a Galois array, use the `rank` function. It behaves like the ordinary MATLAB `rank` function when given exactly one input argument. The example below illustrates how to find the rank of square and nonsquare Galois arrays.

```
m = 3;
asquare = gf([4 7 6; 4 6 5; 0 6 1],m);
r1 = rank(asquare);
anonsquare = gf([4 7 6 3; 4 6 5 1; 0 6 1 1],m);
r2 = rank(anonsquare);
[r1 r2]
```

The output is

```
ans =

     2     3
```

The values of `r1` and `r2` indicate that `asquare` has less than full rank but that `anonsquare` has full rank.

Factoring Square Matrices

To express a square Galois array (or a permutation of it) as the product of a lower triangular Galois array and an upper triangular Galois array, use the `lu` function. This function accepts one input argument and produces exactly two or three output arguments. It behaves like the ordinary MATLAB `lu` function when given the same syntax. The example below illustrates how to factor using `lu`.

```
tofactor = gf([6 5 7 6; 5 6 2 5; 0 1 7 7; 1 0 5 1],3);
[L,U]=lu(tofactor); % lu with two output arguments
c1 = isequal(L*U, tofactor) % True
tofactor2 = gf([1 2 3 4;1 2 3 0;2 5 2 1; 0 5 0 0],3);
[L2,U2,P] = lu(tofactor2); % lu with three output arguments
c2 = isequal(L2*U2, P*tofactor2) % True
```

Solving Linear Equations

To find a particular solution of a linear equation in a Galois field, use the `\` or `/` operator on Galois arrays. The table below indicates the equation that each operator addresses, assuming that `A` and `B` are previously defined Galois arrays.

Operator	Linear Equation	Syntax	Equivalent Syntax Using \
Backslash (`\`)	`A * x = B`	`x = A \ B`	Not applicable
Slash (`/`)	`x * A = B`	`x = B / A`	`x = (A'\B')'`

The results of the syntax in the table depend on characteristics of the Galois array `A`:

- If `A` is square and nonsingular, then the output `x` is the unique solution to the linear equation.
- If `A` is square and singular, then the syntax in the table produces an error.
- If `A` is not square, then MATLAB attempts to find a particular solution. If `A'*A` or `A*A'` is a singular array, or if `A` is a tall matrix that represents an overdetermined system, then the attempt might fail.

Note An error message does not necessarily indicate that the linear equation has no solution. You might be able to find a solution by rephrasing the problem. For example, `gf([1 2; 0 0],3) \ gf([1; 0],3)` produces an error but the mathematically equivalent `gf([1 2],3) \ gf([1],3)` does not. The first syntax fails because `gf([1 2; 0 0],3)` is a singular square matrix.

Example: Solving Linear Equations

The examples below illustrate how to find particular solutions of linear equations over a Galois field.

```
m = 4;
A = gf(magic(3),m); % Square nonsingular matrix
Awide=[A, 2*A(:,3)]; % 3-by-4 matrix with redundancy on the right
Atall = Awide'; % 4-by-3 matrix with redundancy at the bottom
```

```
B = gf([0:2]',m);
C = [B; 2*B(3)];
D = [B; B(3)+1];
thesolution = A \ B; % Solution of A * x = B
thesolution2 = B' / A; % Solution of x * A = B'
ck1 = all(A * thesolution == B) % Check validity of solutions.
ck2 = all(thesolution2 * A == B')
% Awide * x = B has infinitely many solutions. Find one.
onesolution = Awide \ B;
ck3 = all(Awide * onesolution == B) % Check validity of solution.
% Atall * x = C has a solution.
asolution = Atall \ C;
ck4 = all(Atall * asolution == C) % Check validity of solution.
% Atall * x = D has no solution.
notasolution = Atall \ D;
ck5 = all(Atall * notasolution == D) % It is not a valid solution.
```

The output from this example indicates that the validity checks are all true (`1`), except for `ck5`, which is false (`0`).

Signal Processing Operations in Galois Fields

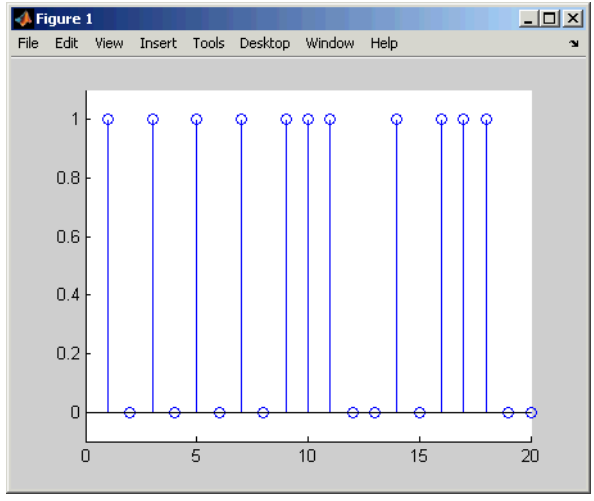
You can perform some signal-processing operations on Galois arrays, such as filtering, convolution, and the discrete Fourier transform. This section describes how to perform these operations. Other information about the corresponding operations for ordinary real vectors is in the Signal Processing Toolbox documentation.

Filtering

To filter a Galois vector, use the `filter` function. It behaves like the ordinary MATLAB `filter` function when given exactly three input arguments.

The code and diagram below give the impulse response of a particular filter over GF(2).

```
m = 1; % Work in GF(2).
b = gf([1 0 0 1 0 1 0 1],m); % Numerator
a = gf([1 0 1 1],m); % Denominator
x = gf([1,zeros(1,19)],m);
y = filter(b,a,x); % Filter x.
figure; stem(y.x); % Create stem plot.
axis([0 20 -.1 1.1])
```



Convolution

This toolbox offers two equivalent ways to convolve a pair of Galois vectors:

- Use the `conv` function, as described in "Multiplication and Division of Polynomials" on page 12-30. This works because convolving two vectors is equivalent to multiplying the two polynomials whose coefficients are the entries of the vectors.
- Use the `convmtx` function to compute the convolution matrix of one of the vectors, and then multiply that matrix by the other vector. This works because convolving two vectors is equivalent to filtering one of the vectors by the other. The equivalence permits the representation of a digital filter as a convolution matrix, which you can then multiply by any Galois vector of appropriate length.

Tip If you need to convolve large Galois vectors, then multiplying by the convolution matrix might be faster than using `conv`.

Example

The example below computes the convolution matrix for a vector `b` in GF(4), representing the numerator coefficients for a digital filter. It then illustrates the two equivalent ways to convolve `b` with `x` over the Galois field.

```
m = 2; b = gf([1 2 3]',m);
n = 3; x = gf(randint(n,1,2^m),m);
C = convmtx(b,n); % Compute convolution matrix.
v1 = conv(b,x); % Use conv to convolve b with x
v2 = C*x; % Use C to convolve b with x.
```

Discrete Fourier Transform

The discrete Fourier transform is an important tool in digital signal processing. This toolbox offers these tools to help you process discrete Fourier transforms:

- `fft`, which transforms a Galois vector
- `ifft`, which inverts the discrete Fourier transform on a Galois vector

- `dftmtx`, which returns a Galois array that you can use to perform or invert the discrete Fourier transform on a Galois vector

In all cases, the vector being transformed must be a Galois vector of length 2^m-1 in the field GF(2^m). The examples below illustrate the use of these functions. You can check, using the `isequal` function, that `y` equals `y1`, `z` equals `z1`, and `z` equals `x`.

```
m = 4;
x = gf(randint(2^m-1,1,2^m),m); % A vector to transform
alph = gf(2,m);
dm = dftmtx(alph);
idm = dftmtx(1/alph);
y = dm*x; % Transform x using the result of dftmtx.
y1 = fft(x); % Transform x using fft.
z = idm*y; % Recover x using the result of dftmtx(1/alph).
z1 = ifft(y1); % Recover x using ifft.
```

Tip If you have many vectors that you want to transform (in the same field), then it might be faster to use `dftmtx` once and matrix multiplication many times, instead of using `fft` many times.

Polynomials over Galois Fields

You can use Galois vectors to represent polynomials in an indeterminate quantity x, with coefficients in a Galois field. Form the representation by listing the coefficients of the polynomial in a vector in order of descending powers of x. For example, the vector

```
gf([2 1 0 3],4)
```

represents the polynomial $Ax^3 + 1x^2 + 0x + (A+1)$, where

- A is a primitive element in the field $GF(2^4)$.
- x is the indeterminate quantity in the polynomial.

You can then use such a Galois vector to perform arithmetic with, evaluate, and find roots of polynomials. You can also find minimal polynomials of elements of a Galois field.

Addition and Subtraction of Polynomials

To add and subtract polynomials, use + and - on equal-length Galois vectors that represent the polynomials. If one polynomial has lower degree than the other, then you must pad the shorter vector with zeros at the beginning so that the two vectors have the same length. The example below shows how to add a degree-one and a degree-two polynomial.

```
lin = gf([4 2],3); % A^2 x + A, which is linear in x
linpadded = gf([0 4 2],3); % The same polynomial, zero-padded
quadr = gf([1 4 2],3); % x^2 + A^2 x + A, which is quadratic in x
% Can't do lin + quadr because they have different vector lengths.
sumpoly = [0, lin] + quadr; % Sum of the two polynomials
sumpoly2 = linpadded + quadr; % The same sum
```

Multiplication and Division of Polynomials

To multiply and divide polynomials, use `conv` and `deconv` on Galois vectors that represent the polynomials. Multiplication and division of polynomials is equivalent to convolution and deconvolution of vectors. The `deconv` function returns the quotient of the two polynomials as well as the remainder polynomial. Examples are below.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
bpoly = gf([1 1],m); % x + 1
xpoly = gf([1 0],m); % x
% Product is A^2 x^3 + x^2 + (A^2 + A) x + (A + 1).
cpoly = conv(apoly,bpoly);
[a2,remd] = deconv(cpoly,bpoly); % a2==apoly. remd is zero.
[otherpol,remd2] = deconv(cpoly,xpoly); % remd is nonzero.
```

The multiplication and division operators in “Arithmetic in Galois Fields” on page 12-13 multiply elements or matrices, not polynomials.

Evaluating Polynomials

To evaluate a polynomial at an element of a Galois field, use `polyval`. It behaves like the ordinary MATLAB `polyval` function when given exactly two input arguments. The example below evaluates a polynomial at several elements in a field and checks the results using `.^` and `.*` in the field.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
x0 = gf([0 1 2],m); % Points at which to evaluate the polynomial
y = polyval(apoly,x0)

a = gf(2,m); % Primitive element of the field, corresponding to A.
y2 = a.^2.*x0.^2 + (a.^2+1).*x0 + (a+1) % Check the result.
```

The output is below.

```
y = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10


y2 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10
```

The first element of y evaluates the polynomial at 0 and, therefore, returns the polynomial's constant term of 3.

Roots of Polynomials

To find the roots of a polynomial in a Galois field, use the roots function on a Galois vector that represents the polynomial. This function finds roots that are in the same field that the Galois vector is in. The number of times an entry appears in the output vector from roots is exactly its multiplicity as a root of the polynomial.

Note If the Galois vector is in GF(2^m), then the polynomial it represents might have additional roots in some extension field GF($(2^m)^k$). However, roots does not find those additional roots or indicate their existence.

The examples below find roots of cubic polynomials in GF(8).

```
m = 3;
cubicpoly1 = gf([2 7 3 0],m); % A polynomial divisible by x
cubicpoly2 = gf([2 7 3 1],m);
cubicpoly3 = gf([2 7 3 2],m);
zeroandothers = roots(cubicpoly1); % Zero is among the roots.
multipleroots = roots(cubicpoly2); % One root has multiplicity 2.
oneroot = roots(cubicpoly3); % Only one root is in GF(2^m).
```

Roots of Binary Polynomials

In the special case of a polynomial having binary coefficients, it is also easy to find roots that exist in an extension field. This because the elements 0 and 1 have the same unambiguous representation in all fields of characteristic two. To find roots of a binary polynomial in an extension field, apply the roots function to a Galois vector in the extension field whose array elements are the binary coefficients of the polynomial.

The example below seeks roots of a binary polynomial in various fields.

```
gf2poly = gf([1 1 1],1); % x^2 + x + 1 in GF(2)
noroots = roots(gf2poly); % No roots in the ground field, GF(2)
```

```
gf4poly = gf([1 1 1],2); % x^2 + x + 1 in GF(4)
roots4 = roots(gf4poly); % The roots are A and A+1, in GF(4).
gf16poly = gf([1 1 1],4); % x^2 + x + 1 in GF(16)
roots16 = roots(gf16poly); % Roots in GF(16)
checkanswer4 = polyval(gf4poly,roots4); % Zero vector
checkanswer16 = polyval(gf16poly,roots16); % Zero vector
```

The roots of the polynomial do not exist in GF(2), so `noroots` is an empty array. However, the roots of the polynomial exist in GF(4) as well as in GF(16), so `roots4` and `roots16` are nonempty.

Notice that `roots4` and `roots16` are not equal to each other. They differ in these ways:

- `roots4` is a GF(4) array, while `roots16` is a GF(16) array. MATLAB keeps track of the underlying field of a Galois array.
- The array elements in `roots4` and `roots16` differ because they use representations with respect to different primitive polynomials. For example, `2` (which represents a primitive element) is an element of the vector `roots4` because the default primitive polynomial for GF(4) is the same polynomial that `gf4poly` represents. On the other hand, `2` is not an element of `roots16` because the primitive element of GF(16) is not a root of the polynomial that `gf16poly` represents.

Minimal Polynomials

The minimal polynomial of an element of GF(2^m) is the smallest-degree nonzero binary-coefficient polynomial having that element as a root in GF(2^m). To find the minimal polynomial of an element or a column vector of elements, use the `minpol` function.

The code below finds that the minimal polynomial of `gf(6,4)` is $D^2 + D + 1$ and then checks that `gf(6,4)` is indeed among the roots of that polynomial in the field GF(16).

```
m = 4;
e = gf(6,4);
em = minpol(e) % Find minimal polynomial of e. em is in GF(2).

emr = roots(gf([0 0 1 1 1],m)) % Roots of D^2+D+1 in GF(2^m)
```

The output is

```
em = GF(2) array.

Array elements =

     0     0     1     1     1


emr = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     6
     7
```

To find out which elements of a Galois field share the same minimal polynomial, use the `cosets` function.

Manipulating Galois Variables

This section describes techniques for manipulating Galois variables or for transferring information between Galois arrays and ordinary MATLAB arrays.

Note These techniques are particularly relevant if you write M-file functions that process Galois arrays. For an example of this type of usage, enter `edit gf/conv` in the Command Window and examine the first several lines of code in the editor window.

Determining Whether a Variable Is a Galois Array

To find out whether a variable is a Galois array rather than an ordinary MATLAB array, use the `isa` function. An illustration is below.

```
mlvar = eye(3);
gfvar = gf(mlvar,3);
no = isa(mlvar,'gf'); % False because mlvar is not a Galois array
yes = isa(gfvar,'gf'); % True because gfvar is a Galois array
```

Extracting Information from a Galois Array

To extract the array elements, field order, or primitive polynomial from a variable that is a Galois array, append a suffix to the name of the variable. The table below lists the exact suffixes, which are independent of the name of the variable.

Information	Suffix	Output Value
Array elements	`.x`	MATLAB array of type `uint16` that contains the data values from the Galois array
Field order	`.m`	Integer of type `double` that indicates that the Galois array is in GF(`2^m`)
Primitive polynomial	`.prim_poly`	Integer of type `uint32` that represents the primitive polynomial. The representation is similar to the description in "How Integers Correspond to Galois Field Elements" on page 12-7.

Note If the output value is an integer data type and you want to convert it to `double` for later manipulation, use the `double` function.

The code below illustrates the use of these suffixes. The definition of `empr` uses a vector of binary coefficients of a polynomial to create a Galois array in an extension field. Another part of the example retrieves the primitive polynomial for the field and converts it to a binary vector representation having the appropriate number of bits.

```
% Check that e solves its own minimal polynomial.
e = gf(6,4); % An element of GF(16)
emp = minpol(e); % The minimal polynomial, emp, is in GF(2).
empr = roots(gf(emp.x,e.m)); % Find roots of emp in GF(16).

% Check that the primitive element gf(2,m) is
% really a root of the primitive polynomial for the field.
primpoly_int = double(e.prim_poly);
```

```
mval = e.m;
primpoly_vect = gf(de2bi(primpoly_int,mval+1,'left-msb'),mval);
containstwo = roots(primpoly_vect); % Output vector includes 2.
```

Speed and Nondefault Primitive Polynomials

The section "Specifying the Primitive Polynomial" on page 12-9 described how you can represent elements of a Galois field with respect to a primitive polynomial of your choice. This section describes how you can increase the speed of computations involving a Galois array that uses a primitive polynomial other than the default primitive polynomial. The technique is recommended if you perform many such computations.

The mechanism for increasing the speed is a data file, `userGftable.mat`, that some computational functions use to avoid performing certain computations repeatedly. To take advantage of this mechanism for your combination of field order (`m`) and primitive polynomial (`prim_poly`):

1. Navigate in MATLAB to a directory to which you have write permission. You can use either the `cd` function or the Current Directory feature to navigate.

2. Define `m` and `prim_poly` as workspace variables. For example:

   ```
   m = 3; prim_poly = 13; % Examples of valid values
   ```

3. Invoke the `gftable` function:

   ```
   gftable(m,prim_poly); % If you previously defined m and prim_poly
   ```

The function revises or creates `userGftable.mat` in your current working directory to include data relating to your combination of field order and primitive polynomial. After you initially invest the time to invoke `gftable`, subsequent computations using those values of `m` and `prim_poly` should be faster.

Note If you change your current working directory after invoking `gftable`, then you must place `userGftable.mat` on your MATLAB path to ensure that MATLAB can see it. Do this by using the `addpath` command to prefix the directory containing `userGftable.mat` to your MATLAB path. If you have multiple copies of `userGftable.mat` on your path, then use `which('userGftable.mat','-all')` to find out where they are and which one MATLAB is using.

To see how much `gftable` improves the speed of your computations, you can surround your computations with the `tic` and `toc` functions. See the `gftable` reference page for an example.

Selected Bibliography for Galois Fields

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

[2] Lang, Serge, *Algebra*, Third Edition, Reading, Mass., Addison-Wesley, 1993.

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

[4] van Lint, J. H., *Introduction to Coding Theory*, New York, Springer-Verlag, 1982.

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

13

Galois Fields of Odd Characteristic

A *Galois field* is an algebraic field having p^m elements, where p is prime and m is a positive integer. This chapter describes how to work with Galois fields in which p is *odd*. To work with Galois fields having an even number of elements, see Galois Field Computations. The sections in this chapter are as follows.

Galois Field Terminology

Throughout this section, p is an odd prime number and m is a positive integer.

Also, this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [4].

- A *primitive element* of $GF(p^m)$ is a cyclic generator of the group of nonzero elements of $GF(p^m)$. This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power. Primitive elements are called A throughout this section.
- A *primitive polynomial* for $GF(p^m)$ is the minimal polynomial of some primitive element of $GF(p^m)$. As a consequence, it has degree m and is irreducible.

Representing Elements of Galois Fields

This section discusses how to represent Galois field elements using this toolbox's exponential format and polynomial format. It also describes a way to list all elements of the Galois field, because some functions use such a list as an input argument. Finally, it discusses the nonuniqueness of representations of Galois field elements.

The elements of GF(p) can be represented using the integers from 0 to p-1.

When m is at least 2, GF(p^m) is called an extension field. Integers alone cannot represent the elements of GF(p^m) in a straightforward way. MATLAB uses two main conventions for representing elements of GF(p^m): the exponential format and the polynomial format.

Note Both the exponential format and the polynomial format are relative to your choice of a particular primitive element A of GF(p^m).

Exponential Format

This format uses the property that every nonzero element of GF(p^m) can be expressed as A^c for some integer c between 0 and p^m-2. Higher exponents are not needed, because the theory of Galois fields implies that every nonzero element of GF(p^m) satisfies the equation $x^{q-1} = 1$ where $q = p^m$.

The use of the exponential format is shown in the table below.

Element of GF(p^m)	MATLAB Representation of the Element
0	`-Inf`
$A^0 = 1$	`0`
A^1	`1`
...	...
A^{q-2} where `q` = p^m	`q-2`

Although `-Inf` is the standard exponential representation of the zero element, all negative integers are equivalent to `-Inf` when used as *input* arguments in exponential format. This equivalence can be useful; for example, see the concise line of code at the end of the section "Default Primitive Polynomials" on page 13-8.

Note The equivalence of all negative integers and `-Inf` as exponential formats means that, for example, -1 does *not* represent A^{-1}, the multiplicative inverse of A. Instead, -1 represents the zero element of the field.

Polynomial Format

The polynomial format uses the property that every element of GF(p^m) can be expressed as a polynomial in A with exponents between 0 and m-1, and coefficients in GF(p). In the polynomial format, the element

`A(1) + A(2)` $A + A(3)\ A^2 + \ldots + A(m)\ A^{m-1}$

is represented in MATLAB by the vector

```
[A(1) A(2) A(3) ... A(m)]
```

Note The Galois field functions in this toolbox represent a polynomial as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

List of All Elements of a Galois Field

Some Galois field functions in this toolbox require an argument that lists all elements of an extension field GF(p^m). This is again relative to a particular primitive element A of GF(p^m). The proper format for the list of elements is that of a matrix having p^m rows, one for each element of the field. The matrix has m columns, one for each coefficient of a power of A in the polynomial format shown in "Polynomial Format" on page 13-5 above. The first row contains only zeros because it corresponds to the zero element in GF(p^m). If k is between 2 and p^m, then the kth row specifies the polynomial format of the element A^{k-2}.

The minimal polynomial of A aids in the computation of this matrix, because it tells how to express A^m in terms of lower powers of A. For example, the table below lists the elements of $GF(3^2)$, where A is a root of the primitive polynomial $2 + 2x + x^2$. This polynomial allows repeated use of the substitution

$A^2 = -2 - 2A = 1 + A$

when performing the computations in the middle column of the table.

Elements of GF(9)

Exponential Format	Polynomial Format	Row of MATLAB Matrix of Elements
A^{-Inf}	0	0 0
A^0	1	1 0
A^1	A	0 1
A^2	1+A	1 1
A^3	$A + A^2 = A + 1 + A = 1 + 2A$	1 2
A^4	$A + 2A^2 = A + 2 + 2A = 2$	2 0
A^5	2A	0 2
A^6	$2A^2 = 2 + 2A$	2 2
A^7	$2A + 2A^2 = 2A + 2 + 2A = 2 + A$	2 1

Example

An automatic way to generate the matrix whose rows are in the third column of the table above is to use the code below.

```
p = 3; m = 2;
% Use the primitive polynomial 2 + 2x + x^2 for GF(9).
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

The `gftuple` function is discussed in more detail in “Converting and Simplifying Element Formats” on page 13-9.

Nonuniqueness of Representations

A given field has more than one primitive element. If two primitive elements have different minimal polynomials, then the corresponding matrices of elements will have their rows in a different order. If the two primitive elements share the same minimal polynomial, then the matrix of elements of the field is the same.

Note You can use whatever primitive element you want, as long as you understand how the inputs and outputs of Galois field functions depend on the choice of *some* primitive polynomial. It is usually best to use the same primitive polynomial throughout a given script or function.

Other ways in which representations of elements are not unique arise from the equations that Galois field elements satisfy. For example, an exponential format of 8 in GF(9) is really the same as an exponential format of 0, because $A^8 = 1 = A^0$ in GF(9). As another example, the substitution mentioned just before the table shows that the polynomial format [0 0 1] is really the same as the polynomial format [1 1].

Default Primitive Polynomials

This toolbox provides a *default* primitive polynomial for each extension field. You can retrieve this polynomial using the `gfprimdf` function. The command

```
prim_poly = gfprimdf(m,p); % If m and p are already defined
```

produces the standard row-vector representation of the default minimal polynomial for GF(p^m).

For example, the command below shows that the default primitive polynomial for GF(9) is $2 + x + x^2$, *not* the polynomial used in "List of All Elements of a Galois Field" on page 13-5.

```
poly1=gfprimdf(2,3);

poly1 =

     2     1     1
```

To generate a list of elements of GF(p^m) using the default primitive polynomial, use the command

```
field = gftuple([-1:p^m-2]',m,p);
```

Converting and Simplifying Element Formats

This section describes how to convert between the exponential and polynomial formats for Galois field elements, as well as how to simplify a given representation.

Converting to Simplest Polynomial Format

The `gftuple` function produces the simplest polynomial representation of an element of $GF(p^m)$, given either an exponential representation or a polynomial representation of that element. This can be useful for generating the list of elements of $GF(p^m)$ that other functions require.

Using `gftuple` requires three arguments: one representing an element of $GF(p^m)$, one indicating the primitive polynomial that MATLAB should use when computing the output, and the prime p. The table below indicates how `gftuple` behaves when given the first two arguments in various formats.

Behavior of gftuple Depending on Format of First Two Inputs

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Exponential format; c = any integer	Integer m > 1	Polynomial format of A^c, where A is a root of the *default* primitive polynomial for $GF(p^m)$
Example: `tp = gftuple(6,2,3); % c = 6 here`		
Exponential format; c = any integer	Vector of coefficients of primitive polynomial	Polynomial format of A^c, where A is a root of the *given* primitive polynomial
Example: `polynomial = gfprimdf(2,3); tp = gftuple(6,polynomial,3); % c = 6 here`		

Behavior of gftuple Depending on Format of First Two Inputs (Continued)

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Polynomial format of any degree	Integer m > 1	Polynomial format of degree < m, using *default* primitive polynomial for GF(p^m) to simplify
Example: `tp = gftuple([0 0 0 0 0 0 1],2,3);`		
Polynomial format of any degree	Vector of coefficients of primitive polynomial	Polynomial format of degree < m, using the *given* primitive polynomial for GF(p^m) to simplify
Example: `polynomial = gfprimdf(2,3); tp = gftuple([0 0 0 0 0 0 1],polynomial,3);`		

The four examples that appear in the table above all produce the same vector `tp = [2, 1]`, but their different inputs to `gftuple` correspond to the lines of the table. Each example expresses the fact that $A^6 = 2+A$, where A is a root of the (default) primitive polynomial $2 + x + x^2$ for GF(3^2).

Example

This example shows how `gfconv` and `gftuple` combine to multiply two polynomial-format elements of GF(3^4). Initially, `gfconv` multiplies the two polynomials, treating the primitive element as if it were a variable. This produces a high-order polynomial, which `gftuple` simplifies using the polynomial equation that the primitive element satisfies. The final result is the simplest polynomial format of the product.

```
p = 3; m = 4;
a = [1 2 0 1]; b = [2 2 1 2];
notsimple = gfconv(a,b,p) % a times b, using high powers of alpha
simple = gftuple(notsimple,m,p) %Highest exponent of alpha is m-1
```

The output is below.

```
notsimple =

     2     0     2     0     0     1     2


simple =

     2     1     0     1
```

Example: Generating a List of Galois Field Elements

This example applies the conversion functionality to the task of generating a matrix that lists all elements of a Galois field. A matrix that lists all field elements is an input argument in functions such as `gfadd` and `gfmul`. The variables `field1` and `field2` below have the format that such functions expect.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field1 = gftuple([-1:p^m-2]',m,p);

prim_poly = gfprimdf(m,p); % Or any primitive polynomial
% for GF(p^m)
field2 = gftuple([-1:p^m-2]',prim_poly,p);
```

Converting to Simplest Exponential Format

The same function `gftuple` also produces the simplest exponential representation of an element of GF(p^m), given either an exponential representation or a polynomial representation of that element. To retrieve this output, use the syntax

```
[polyformat, expformat] = gftuple(...)
```

The input format and the output `polyformat` are as in the table . In addition, the variable `expformat` contains the simplest exponential format of the element represented in `polyformat`. It is *simplest* in the sense that the exponent is either `-Inf` or a number between 0 and p^m-2.

Example

To recover the exponential format of the element 2 + A that the previous section considered, use the commands below. In this case, `polyformat` contains redundant information, while `expformat` contains the desired result.

```
[polyformat, expformat] = gftuple([2 1],2,3)

polyformat =

     2     1

expformat =

     6
```

This output appears at first to contradict the information in the table , but in fact it does not. The table uses a different primitive element; two plus that primitive element has the polynomial and exponential formats shown below.

```
prim_poly = [2 2 1];
[polyformat2, expformat2] = gftuple([2 1],prim_poly,3)
```

The output below reflects the information in the bottom line of the table.

```
polyformat2 =

     2     1


expformat2 =

     7
```

Arithmetic in Galois Fields

You can add, subtract, multiply, and divide elements of Galois fields using the functions `gfadd`, `gfsub`, `gfmul`, and `gfdiv`, respectively. Each of these functions has a mode for prime fields and a mode for extension fields.

Arithmetic in Prime Fields

Arithmetic in GF(p) is the same as arithmetic modulo p. The functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` accept two arguments that represent elements of GF(p) as integers between 0 and p-1. The third argument specifies p.

Example: Addition Table for GF(5)

The code below constructs an addition table for GF(5). If `a` and `b` are between 0 and 4, then the element `gfp_add(a+1,b+1)` represents the sum `a+b` in GF(5). For example, `gfp_add(3,5) = 1` because 2+4 is 1 modulo 5.

```
p = 5;
row = 0:p-1;
table = ones(p,1)*row;
gfp_add = gfadd(table,table',p)
```

The output is below.

```
gfp_add =

     0     1     2     3     4
     1     2     3     4     0
     2     3     4     0     1
     3     4     0     1     2
     4     0     1     2     3
```

Other values of `p` produce tables for different prime fields GF(`p`). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(`p`).

Arithmetic in Extension Fields

The same arithmetic functions can add elements of $GF(p^m)$ when $m > 1$, but the format of the arguments is more complicated than in the case above. In

general, arithmetic in extension fields is more complicated than arithmetic in prime fields; see the works listed in "Selected Bibliography for Galois Fields" on page 13-22 for details about how the arithmetic operations work.

When working in extension fields, the functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` use the first two arguments to represent elements of $GF(p^m)$ in exponential format. The third argument, which is required, lists all elements of $GF(p^m)$ as described in "List of All Elements of a Galois Field" on page 13-5. The result is in exponential format.

Example: Addition Table for GF(9)

The code below constructs an addition table for $GF(3^2)$, using exponential formats relative to a root of the default primitive polynomial for GF(9). If `a` and `b` are between -1 and 7, then the element `gfpm_add(a+2,b+2)` represents the sum of A^a and A^b in GF(9). For example, `gfpm_add(4,6) = 5` because

$A^2 + A^4 = A^5$

Using the fourth and sixth rows of the matrix `field`, you can verify that

$A^2 + A^4 = (1 + 2A) + (2 + 0A) = 3 + 2A = 0 + 2A = A^5$ modulo 3.

```
p = 3; m = 2; % Work in GF(3^2).
field = gftuple([-1:p^m-2]',m,p); % Construct list of elements.
row = -1:p^m-2;
table = ones(p^m,1)*row;
gfpm_add = gfadd(table,table',field)
```

The output is below.

```
gfpm_add =

  -Inf     0     1     2     3     4     5     6     7
     0     4     7     3     5  -Inf     2     1     6
     1     7     5     0     4     6  -Inf     3     2
     2     3     0     6     1     5     7  -Inf     4
     3     5     4     1     7     2     6     0  -Inf
     4  -Inf     6     5     2     0     3     7     1
     5     2  -Inf     7     6     3     1     4     0
     6     1     3  -Inf     0     7     4     2     5
     7     6     2     4  -Inf     1     0     5     3
```

Note If you used a different primitive polynomial, then the tables would look different. This makes sense because the ordering of the rows and columns of the tables was based on that particular choice of primitive polynomial and not on any natural ordering of the elements of GF(9).

Other values of `p` and `m` produce tables for different extension fields GF(`p^m`). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(`p^m`).

Polynomials over Prime Fields

A polynomial over GF(p) is a polynomial whose coefficients are elements of GF(p). The Communications Toolbox provides functions for

- Changing polynomials in cosmetic ways
- Performing polynomial arithmetic
- Characterizing polynomials as primitive or irreducible
- Finding roots of polynomials in a Galois field

Note The Galois field functions in this toolbox represent a polynomial over GF(p) for odd values of p as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

Cosmetic Changes of Polynomials

To display the traditionally formatted polynomial that corresponds to a row vector containing coefficients, use `gfpretty`. To truncate a polynomial by removing all zero-coefficient terms that have exponents *higher* than the degree of the polynomial, use `gftrunc`. For example,

```
polynom = gftrunc([1 20 394 10 0 0 29 3 0 0])
gfpretty(polynom)
```

The output is below.

```
polynom =

     1    20   394    10     0     0    29     3

                                    2       3       6       7
                  1 + 20 X + 394 X  + 10 X  + 29 X  + 3 X
```

Note If you do not use a fixed-width font, then the spacing in the display might not look correct.

Polynomial Arithmetic

The functions `gfadd` and `gfsub` add and subtract, respectively, polynomials over GF(p). The `gfconv` function multiplies polynomials over GF(p). The `gfdeconv` function divides polynomials in GF(p), producing a quotient polynomial and a remainder polynomial. For example, the commands below show that $2 + x + x^2$ times $1 + x$ over the field GF(3) is $2 + 2x^2 + x^3$.

```
a = gfconv([2 1 1],[1 1],3)
[quot, remd] = gfdeconv(a,[2 1 1],3)
```

The output is below.

```
a =

     2     0     2     1


quot =

     1     1

remd =

     0
```

The previously discussed functions `gfadd` and `gfsub` add and subtract, respectively, polynomials. Because it uses a vector of coefficients to represent a polynomial, MATLAB does not distinguish between adding two polynomials and adding two row vectors elementwise.

Characterization of Polynomials

Given a polynomial over GF(p), the `gfprimck` function determines whether it is irreducible and/or primitive. By definition, if it is primitive then it is

irreducible; however, the reverse is not necessarily true. The `gfprimdf` and `gfprimfd` functions return primitive polynomials.

Given an element of GF(p^m), the `gfminpol` function computes its minimal polynomial over GF(p).

Example

For example, the code below reflects the irreducibility of all minimal polynomials. However, the minimal polynomial of a nonprimitive element is not a primitive polynomial.

```
p = 3; m = 4;
% Use default primitive polynomial here.

prim_poly = gfminpol(1,m,p);
ckprim = gfprimck(prim_poly,p);
% ckprim = 1, since prim_poly represents a primitive polynomial.

notprimpoly = gfminpol(5,m,p);
cknotprim = gfprimck(notprimpoly,p);
% cknotprim = 0 (irreducible but not primitive)
% since alpha^5 is not a primitive element when p = 3.

ckreducible = gfprimck([0 1 1],p);
% ckreducible = -1 since the polynomial is reducible.
```

Roots of Polynomials

Given a polynomial over GF(p), the `gfroots` function finds the roots of the polynomial in a suitable extension field GF(p^m). There are two ways to tell MATLAB the degree m of the extension field GF(p^m), as shown in the table below.

Formats for Second Argument of gfroots

Second Argument	Represents
A positive integer	m as in GF(p^m). MATLAB uses the default primitive polynomial in its computations.
A row vector	A primitive polynomial for GF(p^m). Here m is the degree of this primitive polynomial.

Example: Roots of a Polynomial in GF(9)

The code below finds roots of the polynomial $1 + x^2 + x^3$ in GF(9) and then checks that they are indeed roots. The exponential format of elements of GF(9) is used throughout.

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
   root = rts(ii);
   rootsquared = gfmul(root,root,field);
   rootcubed = gfmul(root,rootsquared,field);
   answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
   % Recall that 1 is really alpha to the zero power.
   % If answer = -Inf, then the variable root represents
   % a root of the polynomial.
end
answer
```

The output shows that A^0 (which equals 1), A^5, and A^7 are roots.

```
roots =

     0
```

```
     5
     7


answer =

  -Inf  -Inf  -Inf
```

See the reference page for `gfroots` to see how `gfroots` can also provide you with the polynomial formats of the roots and the list of all elements of the field.

Other Galois Field Functions

See the online reference pages for information about these other Galois field functions in the Communications Toolbox:

- `gfcosets`, which produces cyclotomic cosets
- `gffilter`, which filters data using GF(p) polynomials
- `gfprimfd`, which finds primitive polynomials
- `gfrank`, which computes the rank of a matrix over GF(p)
- `gfrepcov`, which converts one binary polynomial representation to another

Selected Bibliography for Galois Fields

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

[2] Lang, Serge, *Algebra*, Third Edition, Reading, Mass., Addison-Wesley, 1993.

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

[4] van Lint, J. H., *Introduction to Coding Theory*, New York, Springer-Verlag, 1982.

Functions — Categorical List

Signal Sources

randerr	Generate bit error patterns
randint	Generate matrix of uniformly distributed random integers
randsrc	Generate random matrix using prescribed alphabet
wgn	Generate white Gaussian noise

Performance Evaluation

berawgn	Bit error rate (BER) for uncoded AWGN channels
bercoding	Bit error rate (BER) for coded AWGN channels
berconfint	BER and confidence interval of Monte Carlo simulation
berfading	Bit error rate (BER) for Rayleigh fading channels
berfit	Fit curve to nonsmooth empirical BER data
bersync	Bit error rate (BER) for imperfect synchronization
biterr	Compute number of bit errors and bit error rate
distspec	Compute the distance spectrum of convolutional code
eyediagram	Generate eye diagram
noisebw	Equivalent noise bandwidth of filter
scatterplot	Generate scatter plot
semianalytic	Calculate bit error rate using semianalytic technique
symerr	Compute number of symbol errors and symbol error rate

Source Coding

arithdeco	Decode binary code using arithmetic decoding
arithenco	Encode sequence of symbols using arithmetic coding
compand	Source code mu-law or A-law compressor or expander
dpcmdeco	Decode using differential pulse code modulation
dpcmenco	Encode using differential pulse code modulation
dpcmopt	Optimize differential pulse code modulation parameters
huffmandeco	Huffman decoder
huffmandict	Generate Huffman code dictionary for source with known probability model
huffmanenco	Huffman encoder
lloyds	Optimize quantization parameters using Lloyd algorithm
quantiz	Produce quantization index and quantized output value

Error-Control Coding

bchdec	BCH decoder
bchenc	BCH encoder
bchgenpoly	Generator polynomial of BCH code
convenc	Convolutionally encode binary data
cyclgen	Produce parity-check and generator matrices for cyclic code
cyclpoly	Produce generator polynomials for cyclic code
decode	Block decoder
encode	Block encoder
gen2par	Convert between parity-check and generator matrices
gfweight	Calculate minimum distance of linear block code
hammgen	Produce parity-check and generator matrices for Hamming code
rsdec	Reed-Solomon decoder
rsdecof	Decode ASCII file that was encoded using Reed-Solomon code
rsenc	Reed-Solomon encoder
rsencof	Encode ASCII file using Reed-Solomon code
rsgenpoly	Generator polynomial of Reed-Solomon code
syndtable	Produce syndrome decoding table
vitdec	Convolutionally decode binary data using Viterbi algorithm

Interleaving/Deinterleaving

algdeintrlv	Restore ordering of symbols using algebraically derived permutation table
algintrlv	Reorder symbols using algebraically derived permutation table
convdeintrlv	Restore ordering of symbols using shift registers
convintrlv	Permute symbols using shift registers
deintrlv	Restore ordering of symbols
heldeintrlv	Restore ordering of symbols permuted using `helintrlv`
helintrlv	Permute symbols using helical array
helscandeintrlv	Restore ordering of symbols in helical pattern
helscanintrlv	Reorder symbols in helical pattern
intrlv	Reorder sequence of symbols
matdeintrlv	Restore ordering of symbols by filling a matrix by columns and emptying it by rows
matintrlv	Reorder symbols by filling a matrix by rows and emptying it by columns
muxdeintrlv	Restore ordering of symbols using specified shift registers
muxintrlv	Permute symbols using shift registers with specified delays
randdeintrlv	Restore ordering of symbols using random permutation
randintrlv	Reorder symbols using random permutation

Analog Modulation/Demodulation

amdemod	Amplitude demodulation
ammod	Amplitude modulation
fmdemod	Frequency demodulation
fmmod	Frequency modulation
pmdemod	Phase demodulation
pmmod	Phase modulation
ssbdemod	Single sideband amplitude demodulation
ssbmod	Single sideband amplitude modulation

Digital Modulation/Demodulation

dpskdemod	Differential phase shift keying demodulation
dpskmod	Differential phase shift keying modulation
fskdemod	Frequency shift keying demodulation
fskmod	Frequency shift keying modulation
genqamdemod	General quadrature amplitude demodulation
genqammod	General quadrature amplitude modulation
modnorm	Scaling factor for normalizing modulation output
mskdemod	Minimum shift keying demodulation
mskmod	Minimum shift keying modulation
oqpskdemod	Offset quadrature phase shift keying demodulation
oqpskmod	Offset quadrature phase shift keying modulation
pamdemod	Pulse amplitude demodulation
pammod	Pulse amplitude modulation
pskdemod	Phase shift keying demodulation
pskmod	Phase shift keying modulation
qamdemod	Quadrature amplitude demodulation
qammod	Quadrature amplitude modulation

Pulse Shaping

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

Special Filters

hank2sys	Convert Hankel matrix to linear system model
hilbiir	Design a Hilbert transform IIR filter
rcosine	Design a raised cosine filter

Lower-Level Functions for Special Filters

rcosfir	Design a raised cosine FIR filter
rcosiir	Design a raised cosine IIR filter

Channels

awgn	Add white Gaussian noise to signal
bsc	Model a binary symmetric channel
filter (channel)	Filter signal with channel object
plot (channel)	Plot channel characteristics with the channel visualization tool
rayleighchan	Construct a Rayleigh fading channel object

reset (channel)	Reset channel object
ricianchan	Construct a Rician fading channel object

Equalizers

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

Galois Field Computations

convmtx	Convolution matrix of Galois field vector
cosets	Produce cyclotomic cosets for a Galois field
dftmtx	Discrete Fourier transform matrix in a Galois field
fft	Discrete Fourier transform
filter (gf)	One-dimensional digital filter over a 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 a Galois field
log	Logarithm in a Galois field
minpol	Find minimal polynomial of a Galois field element
mldivide	Matrix left division \ of Galois arrays
primpoly	Find primitive polynomials for a Galois field

Some additional MATLAB functions that the Communications Toolbox enhances to process elements of Galois fields are below.

`+ -`	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

Computations in Galois Fields of Odd Characteristic

gfadd	Add polynomials over a Galois field
gfconv	Multiply polynomials over a Galois field
gfcosets	Produce cyclotomic cosets for a Galois field
gfdeconv	Divide polynomials over a Galois field
gfdiv	Divide elements of a Galois field
gffilter	Filter data using polynomials over a prime Galois field
gflineq	Find particular solution of `Ax = b` over a prime Galois field
gfminpol	Find minimal polynomial of a Galois field element
gfmul	Multiply elements of a Galois field
gfpretty	Display polynomial in traditional format
gfprimck	Check whether polynomial over a Galois field is primitive
gfprimdf	Provide default primitive polynomials for a Galois field
gfprimfd	Find primitive polynomials for a Galois field
gfrank	Compute rank of matrix over a Galois field
gfrepcov	Convert one binary polynomial representation to another
gfroots	Find roots of polynomial over a prime Galois field

gfsub	Subtract polynomials over a Galois field
gftrunc	Minimize the length of a polynomial representation
gftuple	Simplify or convert the format of elements of a Galois field

Utilities

bi2de	Convert binary vectors to decimal numbers
bin2gray	Convert positive integers into the corresponding Gray-encoded integers
de2bi	Convert decimal numbers to binary vectors
gray2bin	Convert Gray-encoded positive integers to corresponding Gray-decoded integers
iscatastrophic	True for trellis corresponding to a catastrophic convolutional code
istrellis	True for valid trellis structure
marcumq	Generalized Marcum Q function
mask2shift	Convert mask vector to shift for a shift register configuration
oct2dec	Convert octal numbers to decimal numbers
poly2trellis	Convert convolutional code polynomials to trellis description
qfunc	Q function
qfuncinv	Inverse Q function
shift2mask	Convert shift to mask vector for a shift register configuration
vec2mat	Convert vector into matrix

Some additional MATLAB functions in this category are below.

`erf`	Error function
`erfc`	Complementary error function

Graphical User Interface

bertool	Open the bit error rate analysis GUI (BERTool)

15

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, then 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`, Chapter 7, "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. pp. 419.

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

See Also

`algdeintrlv`, Chapter 7, “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. pp. 419.

Purpose

Amplitude demodulation

Syntax

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

Description

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

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

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

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

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

Examples

The code below illustrates the use of a nondefault filter.

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

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

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

See Also

ammod, ssbdemod, fmdemod, pmdemod, Chapter 8, "Modulation"

Purpose

Amplitude modulation

Syntax

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

Description

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

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

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

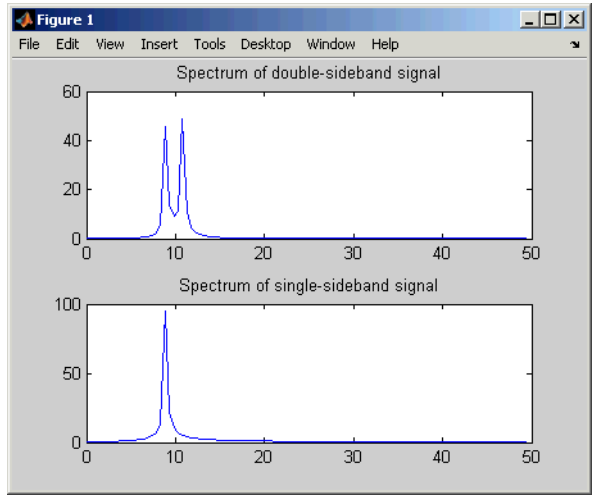
`y = ammod(x,Fc,Fs,ini_phase,carramp)` performs transmitted-carrier modulation instead of suppressed-carrier modulation. The carrier amplitude is `carramp`.

Examples

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

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

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

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

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



See Also

`amdemod`, `ssbmod`, `fmmod`, `pmmod`, Chapter 8, “Modulation”

Purpose

Decode binary code using arithmetic decoding

Syntax

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

Description

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

Examples

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

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

The output is

```
ans =

     1
```

Algorithm

This function uses the algorithm described in [1].

See Also

`arithenco`, "Arithmetic Coding" on page 5-16

References

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

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" on page 5-16

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, then `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`', then `snr` is measured as a ratio and `sigpower` is measured in watts.

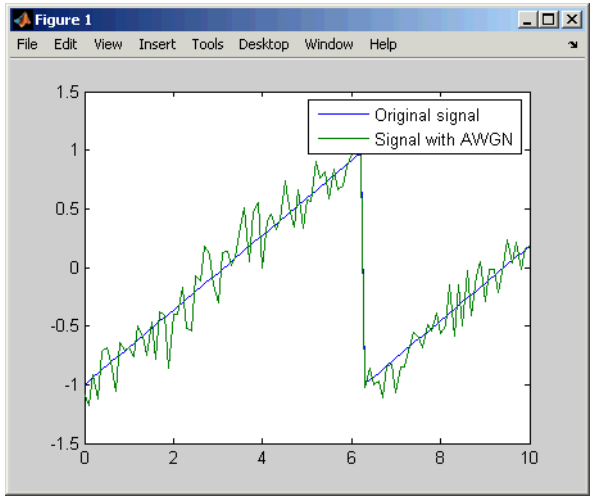
Relationship Among SNR, E_s/N_0, and E_b/N_0

For the relationships between SNR and other measures of the relative power of the noise, see "Describing the Noise Level of an AWGN Channel" on page 10-3.

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 Chapter 1, "Getting Started". The following demos also use `awgn`: `basicsimdemo`, `vitsimdemo`, and `scattereyedemo`.

See Also

`wgn`, `randn`, `bsc`, "AWGN Channel" on page 10-3

Purpose

BCH decoder

Syntax

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

Description

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

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

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

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

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

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 15-17 below.

Limitations

The maximum allowable value of `n` is 65535.

See Also

`bchenc`, `bchgenpoly`, "Block Coding" on page 6-2

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

BCH encoder

Syntax

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

Description

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

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

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

n	k	t
7	4	1

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

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

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

n	k	t
127	120	1
127	113	2
127	106	3

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

n	k	t
255	247	1
255	239	2
255	231	3
255	223	4
255	215	5
255	207	6
255	199	7
255	191	8

n	k	t
255	187	9
255	179	10
255	171	11
255	163	12
255	155	13
255	147	14
255	139	15
255	131	18
255	123	19
255	115	21
255	107	22
255	99	23
255	91	25
255	87	26
255	79	27
255	71	29
255	63	30
255	55	31
255	47	42
255	45	43
255	37	45
255	29	47
255	21	55
255	13	59
255	9	63

n	k	t
511	502	1
511	493	2
511	484	3
511	475	4
511	466	5
511	457	6
511	448	7
511	439	8
511	430	9
511	421	10
511	412	11
511	403	12
511	394	13
511	385	14
511	376	15
511	367	16
511	358	18
511	349	19
511	340	20
511	331	21
511	322	22
511	313	23
511	304	25

n	k	t
511	295	26
511	286	27
511	277	28
511	268	29
511	259	30
511	250	31
511	241	36
511	238	37
511	229	38
511	220	39
511	211	41
511	202	42
511	193	43
511	184	45
511	175	46
511	166	47
511	157	51
511	148	53
511	139	54
511	130	55
511	121	58
511	112	59
511	103	61
511	94	62
511	85	63

n	k	t
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`, “Block Coding” on page 6-2

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 $(X - A^1)(X - A^2)...(X - A^{n-k})$ where A is a root of the default primitive polynomial for the field GF(`n+1`).

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 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 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 generator polynomial $X^4 + X + 1$.

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

```
% Get generator polynomial and error-correction capability.
[genpoly,t] = bchgenpoly(n,k)
```

The output is

```
genpoly = GF(2) array.

Array elements =

     1     0     0     1     1


t =

     1
```

Limitations

The maximum allowable value of `n` is 511.

See Also

`bchenc`, `bchdec`, "Block Coding" on page 6-2

References

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

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,'dpsk',M)
ber = berawgn(EbNo,'fsk',M,coherence)
ber = berawgn(EbNo,'msk',dataenc)
berlb = berawgn(EbNo,'cpfsk',M,modindex,kmin)
```

Graphical Interface

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

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, then 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 table below.

Modulation Scheme	Second Input Argument
Continuous phase frequency shift keying (CPFSK)	`'cpfsk'`
Differential phase shift keying (DPSK)	`'dpsk'`
Frequency shift keying (FSK)	`'fsk'`
Minimum shift keying (MSK)	`'msk'`
Phase shift keying (PSK)	`'psk'`

Modulation Scheme	Second Input Argument
Pulse amplitude modulation (PAM)	`'pam'`
Quadrature amplitude modulation (QAM)	`'qam'`

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 Specific Syntaxes

`ber = berawgn(EbNo,'pam',M)` returns the BER of uncoded PAM over an AWGN channel with coherent demodulation, assuming a Gray-coded signal constellation.

`ber = berawgn(EbNo,'qam',M)` returns the BER of uncoded QAM over an AWGN channel with coherent demodulation, assuming a Gray-coded signal constellation. The alphabet size, `M`, must be at least 4. For cross QAM (`M` not a perfect square), the output `ber` is an *upper bound* on the BER. (Note that the upper bound used here is slightly looser than the upper bound used for cross QAM in the `semianalytic` function.)

`ber = berawgn(EbNo,'psk',M,dataenc)` returns the BER of coherently detected uncoded PSK over an AWGN channel, assuming a Gray-coded signal constellation. `dataenc` is either `'diff'` for differential data encoding or `'nondiff'` for nondifferential data encoding. If `dataenc` is `'diff'` then `M` must be no greater than 4. For details on this calculation, see [2].

`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.

`ber = berawgn(EbNo,'msk',dataenc)` returns the BER of coherently detected uncoded MSK modulation over an AWGN channel. `dataenc`

is either `'diff'` for differential data encoding or `'nondiff'` for nondifferential data encoding. For details on this calculation, see [2].

`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.

Examples

An example using this function is in "Comparing Theoretical and Empirical Error Rates" on page 3-10.

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. However, DQPSK (`'dpsk'` with `M=4`) and differentially encoded PSK (`'psk'` with `'diff'`) have additional limitations, such that the function produces an output of `0` if `EbNo` is large.

See Also

`bercoding`, `berfading`, `bersync`, "Theoretical Performance Results" on page 3-9

References

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

[2] Lindsey, William C. and Marvin K. Simon, *Telecommunication Systems Engineering*, Englewood Cliffs, N.J., Prentice-Hall, 1973.

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

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)
```

Graphical Interface

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

Description

`berub = bercoding(EbNo,'conv',decision,coderate,dspec)` returns an upper bound 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, then `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 [1] 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.

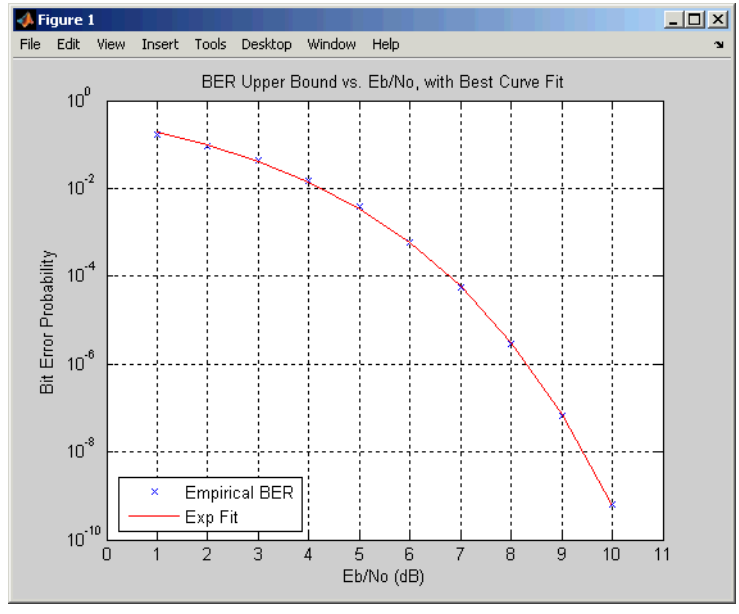
`berub = bercoding(EbNo,'block','soft',n,k,dmin)` returns an upper bound on the BER of an [n,k] binary block code with soft-decision decoding and coherent BPSK or QPSK modulation. `dmin` is the minimum distance of the code.

Examples

An example using this function for a convolutional code is in "Plotting Theoretical Error Rates" on page 3-9.

The next 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" on page 3-9

References

[1] Cedervall, M., and R. Johannesson, "A Fast Algorithm for Computing Distance Spectrum of Convolutional Codes," *IEEE Transactions on Information Theory*, Vol. IT-35, No. 6, Nov. 1989, pp. 1146-1159.

[2] Frenger, Pål, Pål Orten, and Tony 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] Proakis, John G., *Digital Communications*, 4th ed., New York, McGraw-Hill, 2001.

Purpose

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, then the error probability is the bit error rate (BER); if the errors and trials are measured in symbols, then 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, then 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" on page 3-14

See Also

`binofit` (Statistics Toolbox), `mle` (Statistics Toolbox), Chapter 3, "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.

Purpose

Bit error rate (BER) for Rayleigh fading channels

Syntax

```
ber = berfading(EbNo,modtype,M,divorder)
ber = berfading(EbNo,'fsk',2,divorder,coherence)
ber = berfading(EbNo,'psk',2,1,K,phaseerr)
```

Graphical Interface

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

Description

`ber = berfading(EbNo,modtype,M,divorder)` returns the BER of differential phase shift keying (DPSK) or coherent phase shift keying (PSK) modulation over an flat Rayleigh fading channel, with no coding. `EbNo` is the average ratio of bit energy to noise power spectral density, in dB, for each diversity channel. 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. `modtype` represents the type of modulation, and can be either `'dpsk'` or `'psk'`. The argument `M` is the alphabet size, which must be a positive integer power of 2. `divorder` is the diversity order, a positive integer. If `divorder` exceeds 1, then `M` must be 2 or 4 because no well-known theoretical results exist for larger values of `M`.

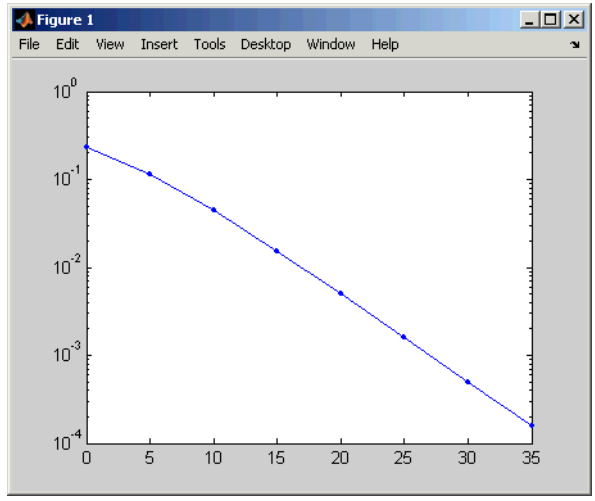
`ber = berfading(EbNo,'fsk',2,divorder,coherence)` returns the BER of uncoded frequency shift keying (FSK) modulation over a flat Rayleigh fading channel. `coherence` indicates whether the function uses coherent or noncoherent demodulation, and can be either `'coherent'` or `'noncoherent'`.

`ber = berfading(EbNo,'psk',2,1,K,phaseerr)` returns the BER of binary phase shift keying (BPSK) over an uncoded flat Rician fading channel, with diversity order 1. `K` is the ratio of specular to diffuse energy (in linear scale). `phaserr` is the standard deviation of the reference carrier phase error (in rad).

Examples

The example below 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 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`, `bercoding`, `bersync`, "Theoretical Performance Results" on page 3-9

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.

Purpose

Fit curve to nonsmooth empirical BER data

Syntax

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

Description

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

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

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

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

Field	Description
`options.Display`	Level of display: `'off'` (default) displays no output; `'iter'` displays output at each iteration; `'final'` displays only the final output; `'notify'` displays output only if the function does not converge.
`options.MaxFunEvals`	Maximum number of function evaluations before optimization ceases. The default is 10^4. Reducing this value might make the function faster but might reduce the quality of the fit.
`options.MaxIter`	Maximum number of iterations before optimization ceases. The default is 10^4. Reducing this value might make the function faster but might reduce the quality of the fit.
`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 15-44 below. `fittype` can be `'exp'`, `'exp+const'`, `'polyRatio'`, or

'doubleExp+const'. To avoid overriding default optimization options, use options = [].

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

Field	Description
fitprops.fitType	The closed-form function type used to generate the fit: 'exp', 'exp+const', 'polyRatio', 'doubleExp+const', or 'all'.
fitprops.coeffs	The coefficients used to generate the fit.
fitprops.sumSqErr	The sum squared error between the log of the fitted BER points and the log of the empirical BER points.
fitprops.exitState	The exit condition of berfit: 'The curve fit converged to a solution.' or 'The maximum number of function evaluations was exceeded.'.
fitprops.funcCount	The number of function evaluations used in minimizing the sum squared error function.
fitprops.iterations	The number of iterations taken in minimizing the sum squared error function. This is not necessarily equal to the number of function evaluations.

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

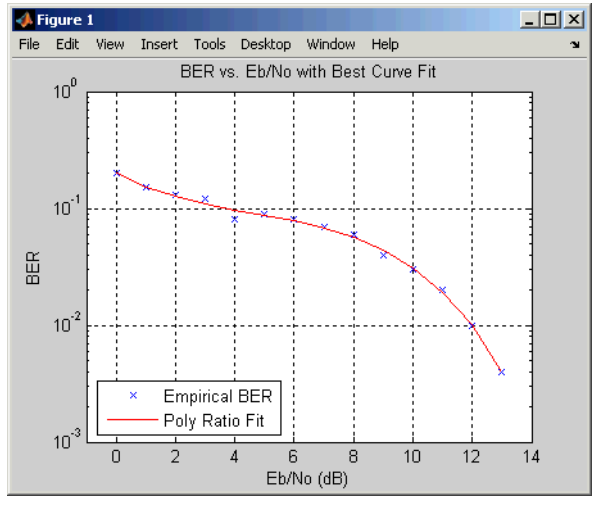
`berfit(empEbNo,empber,fitEbNo,options,'all')` plots the empirical and fitted BER data from all the possible fits listed in "Algorithm" on page 15-44 below. To avoid overriding default options, use `options = []`.

Examples

The examples below illustrate the syntax of the function, but 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" on page 3-14.

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

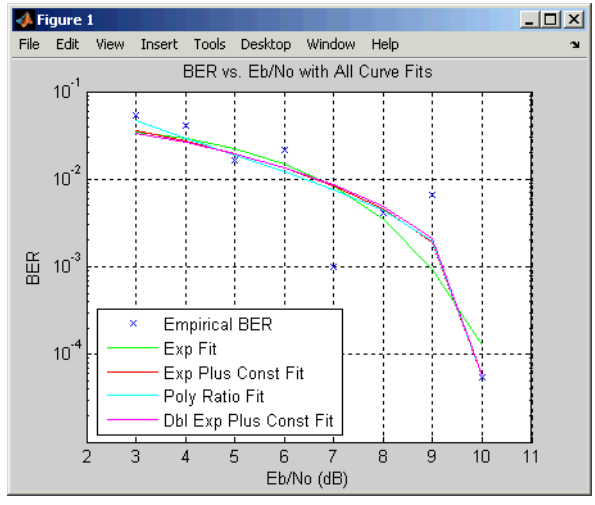
```
EbNo = [0:13];
berdata = [.2 .15 .13 .12 .08 .09 .08 .07 .06 .04 .03 .02 .01 .004];
berfit(EbNo,berdata); % Plot the best fit.
```



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 does not change the fit expression.

The next example plots all fit types that `berfit` considers, for a perturbation of a set of BER data obtained using the `berfading` function. Notice that one of the fit types does not work well for this data, while the other fit types provide much better fits.

```
M = 4; EbNo = [3:10];
berdata = berfading(EbNo,'psk',M,2); % Compute theoretical BER.
noisydata = berdata.*[.93 .92 .5 .89 .058 .35 .8 .01]; % Perturbed data
figure; berfit(EbNo,noisydata,EbNo,[],'all'); % Plot all four fits.
```



The code below 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 .5 .89 .058 .35 .8 .01];
% Say when fit fails to converge.
options = optimset('display','notify');
```

```
disp('*** Trying polynomial ratio fit.') % Poor fit in this case
[fitber1,fitprops1] = berfit(EbNo,noisydata,EbNo,...
   options,'polyRatio')

disp('*** Trying double exponential + constant fit.') % Good fit
[fitber2,fitprops2] = berfit(EbNo,noisydata,EbNo,...
   options,'doubleExp+const')
```

The output is below.

```
*** Trying polynomial ratio fit.

Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 6.136681


fitber1 =

  Columns 1 through 6

    0.0472    0.0289    0.0187    0.0121    0.0077    0.0044

  Columns 7 through 8

    0.0019    0.0001


fitprops1 =

       fitType: 'polyRatio'
        coeffs: [6x1 double]
      sumSqErr: 6.1367
     exitState: [1x56 char]
     funcCount: 10001
    iterations: 3333
```

```
*** Trying double exponential + constant fit.

fitber2 =

  Columns 1 through 6

    0.0338    0.0260    0.0192    0.0134    0.0087    0.0049

  Columns 7 through 8

    0.0021    0.0001


fitprops2 =

       fitType: 'doubleExp+const'
        coeffs: [9x1 double]
      sumSqErr: 6.7044
     exitState: 'The curve fit converged to a solution.'
     funcCount: 1237
    iterations: 822
```

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 `fittype`	Functional Expression
`'exp'`	$f(x) = a_1 \exp\{-[(x - a_2)/a_3]^{a_4}\}$
`'exp+const'`	$f(x) = a_1 \exp\{-[(x - a_2)/a_3]^{a_4}\} + a_5$
`'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'`	$a_1 \exp\{-\left[(x - a_2)/a_3\right]^{a_4}\} + a_5 \exp\{-\left[(x - a_6)/a_7\right]^{a_8}\} + a_9$

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

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

where the fitted BER points are the values in `empber` and where 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.

See Also

`fminsearch`, `optimset`, Chapter 3, "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** panel.

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, then the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. `timerr` is the standard deviation of the timing error, normalized to the symbol interval. `timerr` must be between 0 and 0.5.

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

Examples

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

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

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

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

```
ber =

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

Limitations

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

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

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

Limitations Related to Extreme Values of Input Arguments

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

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

Algorithm

This function uses formulas from [3].

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

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

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

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

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

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

See Also

`berawgn`, `bercoding`, `berfading`, "Theoretical Performance Results" on page 3-9

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, N.J., Prentice-Hall, 2001.

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

Purpose

Open the bit error rate analysis GUI (BERTool)

Syntax

```
bertool
```

Description

`bertool` launches the Bit Error Rate Analysis Tool (BERTool). BERTool is a Graphical User Interface (GUI) that enables you to analyze communications links' BER performance via simulation-based, semianalytic, or theoretical approach. To learn about BERTool, see Chapter 4, "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, then 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, then 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 the 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`.

Examples

To Gray-encode a vector `x` with a 64-QAM Gray-encoded constellation, use

```
y = bin2gray(x,'qam',64);
```

To Gray-encode a vector `x` with a 64-FSK Gray-encoded constellation and return its map, use

```
[y,map] = bin2gray(x,'fsk',64);
```

To Gray-encode a vector `x` with a 256-DPSK Gray-encoded constellation, use

```
y = bin2gray(x,'dpsk',256);
```

To Gray-encode a vector `x` with a 1024-PSK Gray-encoded constellation, use

```
y = bin2gray(x,'psk',1024);
```

See Also

`gray2bin`

Purpose

Compute number of bit errors and bit error rate

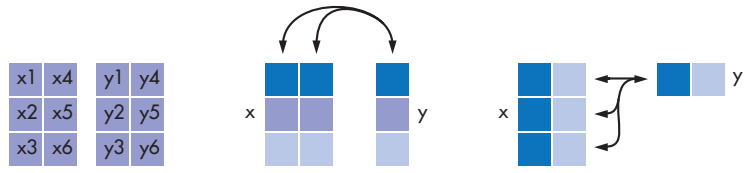
Syntax

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

Description

For All Syntaxes

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



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

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

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

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

For Specific Syntaxes

`[number,ratio] = biterr(x,y)` compares the elements in `x` and `y`. If the largest among all elements of `x` and `y` has exactly `k` bits in its simplest binary representation, then 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 figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then `biterr` compares the vector element by element with *each row (resp., column)* of the matrix. The length of the vector must equal the number of columns (resp., rows) in the matrix. `number` is a column (resp., row) vector whose mth entry indicates the number of bits that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

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

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

Comparing a Two-Dimensional Matrix x with Another Input y

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

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

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

Examples

Example 1

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

```
format rat;
[number,ratio,individual] = biterr([0;0;0],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 `flg` to override the default row-by-row comparison. Notice that `number` and `ratio` are scalars, while `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. Notice that `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” on page 3-2

Purpose

Model a 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.

Examples

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

```
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. Note that 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” on page 10-38.

See Also

`rand`, `awgn`, “Binary Symmetric Channel” on page 10-38

Purpose

Construct a 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" on page 11-8.

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 zeros 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" on page 11-12.

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

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

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

where the * operator denotes the complex conjugate.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `lineareq`, `dfe`, `equalize`, Chapter 11, "Equalizers"

References

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

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

Purpose

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

Syntax

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

Description

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

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

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

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

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

Note The prevailing parameters used in practice are μ= 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

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

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

The output of the A-law compressor is

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

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

See Also

`quantiz`, `dpcmenco`, `dpcmdeco`, "Companding a Signal" on page 5-12

References

[1] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, N.J., 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 kth shift register is `(nrows-k)*slope`, where k = 1, 2, 3,..., `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, then 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” on page 7-9.

Examples

The example in “Effect of Delays on Recovery of Convolutionally Interleaved Data” on page 7-10 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`, Chapter 7, “Interleaving”

Purpose

Convolutionally encode binary data

Syntax

```
code = convenc(msg,trellis)
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" on page 6-34. Each symbol in `msg` consists of `log2(trellis.numInputSymbols)` bits. The vector `msg` contains one or more symbols. The output vector `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits.

`code = convenc(msg,trellis,init_state)` is the same as the syntax above, except that `init_state` specifies the starting state of the encoder registers. The scalar `init_state` is an integer between 0 and `trellis.numStates-1`. If the encoder schematic has more than one input stream, then the shift register that receives the first input stream provides the least significant bits in `init_state`, while the shift register that receives the last input stream provides the most significant bits in `init_state`. To use the default value for `init_state`, specify `init_state` as `0` or `[]`.

`[code,final_state] = convenc(...)` encodes the input message and also returns in `final_state` the encoder's 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 reference page for the `poly2trellis` function.

```
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 ten one-bit symbols. A schematic of

this encoder is in "Trellis Description of a Convolutional Encoder" on page 6-34.

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

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`.

```
trel = 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),trel);
% Encode the rest of msg, using state as an input argument.
code4 = convenc(msg(7:10),trel,fstate);
```

See Also

`vitdec`, `poly2trellis`, `istrellis`, `vitsimdemo`, "Convolutional Coding" on page 6-30

References

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

Purpose

Permute symbols using shift registers

Syntax

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

Description

`intrlved = convintrlv(data,nrows,slope)` permutes the elements in `data` by using a set of `nrows` internal shift registers. The delay value of the kth shift register is `(k-1)*slope`, where k = 1, 2, 3,... `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, then 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 `y1`, obtained using `muxintrlv`, are the same.

```
ans =

     1
```

Another example using this function is in "Effect of Delays on Recovery of Convolutionally Interleaved Data" on page 7-10.

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

`convdeintrlv`, `muxintrlv`, `helintrlv`, Chapter 7, "Interleaving"

Purpose

Convolution matrix of Galois field vector

Syntax

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

Description

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

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

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

Examples

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

```
m = 4;
c = gf([1; 9; 3],m); % Column vector
n = 6;
x = gf(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” on page 12-27

Purpose

Produce cyclotomic cosets for a 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 15-76 below.

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, Mass., Addison-Wesley, 1983, p. 105.

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

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" on page 6-2

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	The row vector representing the polynomial
`'max'`	One generator polynomial having the greatest possible weight	The row vector representing the polynomial
`'all'`	All generator polynomials	A matrix, each row of which represents one such polynomial
a positive integer, L	All generator polynomials having weight L	A 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, then the output `pol` is empty and a warning message is displayed.

Examples

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

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

The output is

```
c1 =

  Columns 1 through 10

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

  Columns 11 through 12

     1     1
     1     1
     0     1


c2 =

  Columns 1 through 10

     1     1     1     1     0     1     0     1     1     0

  Columns 11 through 12

     0     1
```

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

```
c3 = cyclpoly(15,4,6)

No generator polynomial satisfies the given constraints.

c3 =

     []
```

Algorithm

If `opt` is `'min'`, `'max'`, or omitted, then 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” on page 6-2

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,...,flg)
```

Description

`b = de2bi(d)` converts a nonnegative decimal integer `d` to a binary row vector. If `d` is a vector, then 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, then `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, then 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, then `de2bi` treats it like the vector `d(:)`.

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

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

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

Examples

The code below counts to ten in decimal and binary.

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

The output is below.

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

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

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

bb =

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

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

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

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

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

The output is

```
t =

     0     1     1


tleft =

     1     1     0
```

See Also

`bi2de`

Purpose

Block decoder

Syntax

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

Optional Inputs

Input	Default Value
`fmt`	`binary`
`prim_poly`	`gfprimdf(m)` where `n = 2^m-1`
`genpoly`	`cyclpoly(n,k)`
`trt`	Uses `syndtable` to create the syndrome decoding table associated with the method's parity-check matrix

Description

For All Syntaxes

The `decode` function aims to recover messages that were encoded using an error-correction coding technique. The technique and the defining parameters must match those that were used to encode the original signal.

The "For All Syntaxes" on page 15-109 section on the reference page for the `encode` function explains the meanings of `n` and `k`, the possible values of `fmt`, and the possible formats for `code` and `msg`. You should be familiar with the conventions described there before reading the rest of this section. Using the `decode` function with an input argument `code` that was *not* created by the `encode` function might cause errors.

For Specific Syntaxes

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

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

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

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

`[msg,err] = decode(...)` returns a column vector `err` that gives information about error correction. If the code is a convolutional code, then `err` contains the metric calculations used in the decoding decision process. For other types of codes, a nonnegative integer in the rth row of `err` indicates the number of errors corrected in the rth *message* word; a negative integer indicates that there are more errors in the rth word than can be corrected.

`[msg,err,ccode] = decode(...)` returns the corrected code in `ccode`.

`[msg,err,ccode,cerr] = decode(...)` returns a column vector `cerr` whose meaning depends on the format of `code`:

- If `code` is a binary vector, then a nonnegative integer in the rth row of `vec2matcerr` indicates the number of errors corrected in the rth *codeword*; a negative integer indicates that there are more errors in the rth codeword than can be corrected.
- If `code` is not a binary vector, then `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 fifteen bits. Errors are added to the first two bits of the first codeword and the first bit of the second codeword. Then `decode` is used to recover the original message. As a result, the errors are corrected. `err` reflects the fact that the first *message* was recovered after correcting two errors, while the second message was recovered after correcting one error. `cerr` reflects the fact that the first *codeword* was decoded after correcting two errors, while the second codeword was decoded after correcting one error.

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

The output is below.

```
Single-error patterns loaded in decoding table.
 1008 rows remaining.
2-error patterns loaded.  918 rows remaining.
3-error patterns loaded.  648 rows remaining.
4-error patterns loaded.  243 rows remaining.
5-error patterns loaded.  0 rows remaining.
Transpose of err is
     2     1

Transpose of cerr is
     2     1
```

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" on page 6-2

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, then `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`, Chapter 7, “Interleaving”

Purpose

Construct a 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" on page 11-8.

`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. Note that `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" on page 11-14.

Note To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
`EqType`	Fixed value, `'Decision Feedback Equalizer'`
`AlgType`	Name of the adaptive algorithm represented by `alg`
`nWeights`	Number of weights in the forward filter and the feedback filter, in the format `[nfwdweights, nfbkweights]`. The number of weights in the forward filter must be at least 1.
`nSampPerSym`	Number of input samples per symbol (equivalent to `nsamp` input argument). This value relates to both the equalizer structure (See the use of K in "Decision-Feedback Equalizers" on page 11-6.) and an assumption about the signal to be equalized.
`RefTap` (except for CMA equalizers)	Reference tap index, between 1 and `nfwdweights`. Setting this to a value greater than 1 effectively delays the reference signal with respect to the equalizer's input signal.
`SigConst`	Signal constellation, a vector whose length is typically a power of 2.

Property	Description
`Weights`	Vector that concatenates the complex coefficients from the forward filter and the feedback filter. This is the set of w_i values in the schematic in "Decision-Feedback Equalizers" on page 11-6.
`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" on page 11-6.
`ResetBeforeFiltering`	If `1`, each call to `equalize` resets the state of `eqobj` before equalizing. If `0`, the equalization process maintains continuity from one call to the next.
`NumSamplesProcessed`	Number of samples the equalizer processed since the last reset. When you create or reset `eqobj`, this property value is `0`.
Properties specific to the adaptive algorithm represented by `alg`	See reference page for the adaptive algorithm function that created `alg`: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`.

Relationships Among Properties

If you change `nWeights`, then MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

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

An example illustrating relationships among properties is in "Linked Properties of an Equalizer Object" on page 11-14.

Examples

An example is in "Defining an Equalizer Object" on page 11-13.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `equalize`, Chapter 11, "Equalizers"

Purpose

Discrete Fourier transform matrix in a Galois field

Syntax

```
dm = dftmtx(alph)
```

Description

`dm = dftmtx(alph)` returns a Galois array that represents the discrete Fourier transform operation on a Galois vector, with respect to the Galois scalar `alph`. The element `alph` is a primitive nth root of unity in the Galois field $GF(2^m) = GF(n+1)$; that is, n must be the smallest positive value of `k` for which `alph^k` equals 1. The discrete Fourier transform has size n and `dm` is an n-by-n array. The array `dm` represents the transform in the sense that `dm` times any length-n Galois column vector yields the transform of that vector.

Note The inverse discrete Fourier transform matrix is `dftmtx(1/alph)`.

Examples

The example below illustrates the discrete Fourier transform and its inverse, with respect to the element `gf(3,4)`. The example examines the first n powers of that element to make sure that only the nth power equals one. Afterward, the example transforms a random Galois vector, undoes the transform, and checks the result.

```
m = 4;
n = 2^m-1;
a = 3;
alph = gf(a,m);
mp = minpol(alph);
if (mp(1)==1 && isprimitive(mp)) % Check that alph has order n.
    disp('alph is a primitive nth root of unity.')
    dm = dftmtx(alph);
    idm = dftmtx(1/alph);
    x = gf(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 nth 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" on page 12-27

Purpose

Compute the 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" on page 6-34. The output, `spect`, is a structure with these fields:

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

`spect = distspec(trellis)` is the same as `spect = distspec(trellis,1)`.

Examples

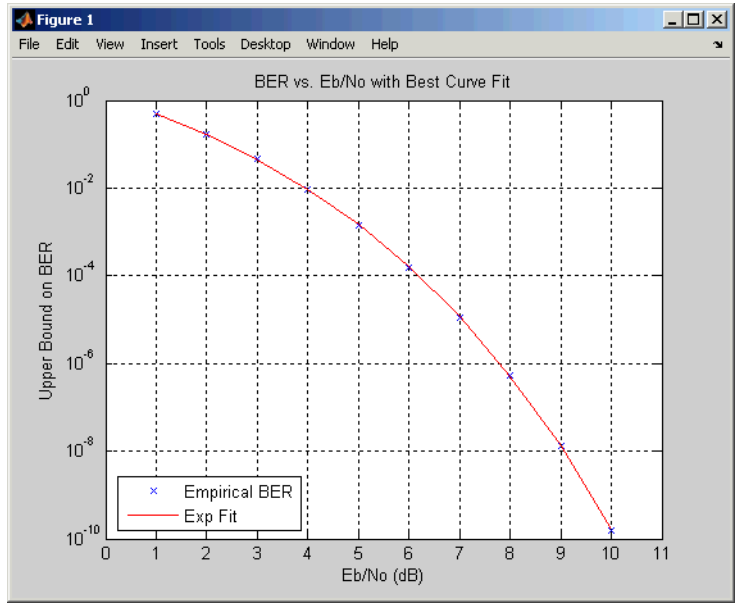
The example below performs these tasks:

- Computes the distance spectrum for the rate 2/3 convolutional code that is depicted on the reference page for the `poly2trellis` function
- Uses the output of `distspec` as an input to the `bercoding` function, to find a theoretical upper bound on the bit error rate for a system that uses this code with coherent BPSK modulation
- Plots the upper bound using the `berfit` function

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])
spect = distspec(trellis,4)
berub = bercoding(1:10,'conv','hard',2/3,spect); % BER bound
berfit(1:10,berub); ylabel('Upper Bound on BER'); % Plot.
```

The output and plot are below.

```
trellis =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]


spect =

     dfree: 5
    weight: [1 6 28 142]
     event: [1 2 8 25]
```



Algorithm

The function uses a tree search algorithm implemented with a stack, as described in [2].

References

[1] Bocharova, Irina E., and Boris 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, Jinn-Ja, Der-June Hwang, and Mao-Chao 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, Pål, Pål Orten, and Tony 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.

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, then `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" on page 5-2, "Representing Codebooks" on page 5-2, or the reference page for `quantiz` in this chapter, 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" on page 5-8 and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 5-10 for examples that use `dpcmdeco`.

See Also

`quantiz`, `dpcmopt`, `dpcmenco`, "Differential Pulse Code Modulation" on page 5-7

References

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

Purpose

Encode using differential pulse code modulation

Syntax

```
indx = dpcmenco(sig,codebook,partition,predictor)
[indx,quants] = dpcmenco(sig,codebook,partition,predictor)
```

Description

`indx = dpcmenco(sig,codebook,partition,predictor)` implements differential pulse code modulation to encode the vector `sig`. `partition` is a vector whose entries give the endpoints of the partition intervals. `codebook`, a vector whose length exceeds the length of `partition` by one, prescribes a value for each partition in the quantization. `predictor` specifies the predictive transfer function. If the transfer function has predictive order M, then `predictor` has length M+1 and an initial entry of 0. The output vector `indx` is the quantization index.

See "Differential Pulse Code Modulation" on page 5-7 for more about the format of `predictor`. See "Representing Partitions" on page 5-2, "Representing Codebooks" on page 5-2, or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[indx,quants] = dpcmenco(sig,codebook,partition,predictor)` is the same as the syntax above, except that `quants` contains the quantization of `sig` based on the quantization parameters. `quants` is a vector of the same size as `sig`.

Note If `predictor` is an order-one transfer function, then the modulation is called a delta modulation.

Examples

See "Example: DPCM Encoding and Decoding" on page 5-8 and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 5-10 for examples that use `dpcmenco`.

See Also

`quantiz`, `dpcmopt`,`dpcmdeco`, "Differential Pulse Code Modulation" on page 5-7

References

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

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" on page 5-7 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" on page 5-2, "Representing Codebooks" on page 5-2, 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" on page 5-10 for an example that uses `dpcmopt`.

See Also

`dpcmenco`, `dpcmdeco`, `quantiz`, `lloyds`, “Differential Pulse Code Modulation” on page 5-7

Purpose

Differential phase shift keying demodulation

Syntax

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

Description

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

Note The first element of the output `z`, or 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use 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” on page 3-14.

See Also

`dpskmod`, `pskdemod`, `pskmod`, Chapter 8, “Modulation”

Purpose

Differential phase shift keying modulation

Syntax

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

Description

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

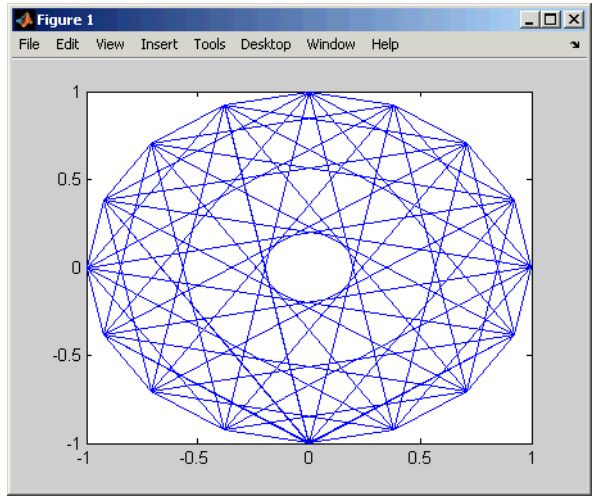
`y = dpskmod(x,M,phaserot)` specifies the phase rotation of the modulation in radians. In this case, the total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

`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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use 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” on page 3-14.

See Also

`dpskdemod`, `pskmod`, `pskdemod`, Chapter 8, “Modulation”

Purpose

Block encoder

Syntax

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

Optional Inputs

Input	Default Value
`fmt`	`binary`
`genpoly`	`cyclpoly(n,k)`
`prim_poly`	`gfprimdf(n-k)`

Description

For All Syntaxes

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

- Linear block
- Cyclic
- Hamming

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

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

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

Information Formats

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

For Specific Syntaxes

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

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

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

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

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

Examples

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

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

% Create 100 messages, k bits each.
msg1 = 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] = hammgen(m);
codehamming2 = encode(msg,n,k,'linear/decimal',genmat);
if codehamming==codehamming2
   disp('The ''linear'' method can create Hamming code.')
```

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

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

The output is

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

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” on page 6-2

Purpose

Equalize signal using an 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" on page 11-8. 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" on page 11-21.

If `eqobj.ResetBeforeFiltering` is `0`, then `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`, then `equalize` resets `eqobj` before starting the equalization operation, overwriting any previous state information in `eqobj`.

`y = equalize(eqobj,x,trainsig)` initially uses a training sequence to adapt the equalizer. After processing the training sequence, the equalizer adapts in decision-directed mode. The vector length of `trainsig` must be less than or equal to `length(x)-(eqobj.RefTap-1)/eqobj.nSampPerSym`.

`[y,yd] = equalize(...)` returns the vector `yd` of detected data symbols.

`[y,yd,e] = equalize(...)` returns the result of the error calculation described in "Error Calculation" on page 11-5. 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" on page 11-17, "Example: Equalizing Multiple Times, Varying the Mode" on page 11-20, and "Example: Adaptive Equalization Within a Loop" on page 11-23.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `dfe`, Chapter 11, "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

`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, then `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, then `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, then `eyediagram` interprets it as a real signal. The figure window contains a single plot.

`eyediagram(x,n,period)` is the same as the syntax above, except that the labels on the horizontal axis range between `-period/2` and `period/2`.

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

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

`eyediagram(x,n,period,offset,plotstring,h)` is the same as the syntax above, except that the eye diagram is in the figure whose handle is `h`, rather than 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

See "Example: Eye Diagrams" on page 3-19 for an example. For an online demonstration, type `playshow scattereyedemo`.

See Also

`scatterplot`, `plot`, `scattereyedemo`, "Eye Diagrams" on page 3-19

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), then 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, then `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” on page 12-27

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`, then `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`, then `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” on page 10-14.

See Also

`rayleighchan`, `ricianchan`, `reset`, “Fading Channels” on page 10-6

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

One-dimensional digital filter over a 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`, then `filter` normalizes the filter coefficients by `a(1)`. As a result, `a(1)` must be nonzero.

The filter is a "Direct Form II Transposed" implementation of the standard difference equation below.

```
a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb) ...
                      - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

`[y,zf] = filter(b,a,x)` returns the final conditions of the filter delays in the Galois vector `zf`. The length of the vector `zf` is `max(size(a),size(b))-1`.

Examples

An example is in “Filtering” on page 12-27.

Purpose

Frequency demodulation

Syntax

```
z = fmdemod(y,Fc,Fs,freqdev)
z = fmdemod(y,Fc,Fs,freqdev,ini_phase)
```

Description

`z = fmdemod(y,Fc,Fs,freqdev)` demodulates the modulating signal `z` from the carrier signal using frequency demodulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least `2*Fc`. The `freqdev` argument is the frequency deviation (Hz) of the modulated signal `y`.

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

Examples

An example using `fmdemod` is on the reference page for `fmmod`.

See Also

`fmmod`, `pmmod`, `pmdemod`, Chapter 8, "Modulation"

fmmod

Purpose

Frequency modulation

Syntax

```
y = fmmod(x,Fc,Fs,freqdev)
y = fmmod(x,Fc,Fs,freqdev,ini_phase)
```

Description

`y = fmmod(x,Fc,Fs,freqdev)` modulates the message signal `x` using frequency modulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least `2*Fc`. The `freqdev` argument is the frequency deviation (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`, Chapter 8, “Modulation”

Purpose

Frequency shift keying demodulation

Syntax

```
z = fskdemod(y,M,freq_sep,nsamp)
z = fskdemod(y,M,freq_sep,nsamp,Fs)
z = fskdemod(y,M,freq_sep,nsamp,Fs,symbol_order)
```

Description

`z = fskdemod(y,M,freq_sep,nsamp)` noncoherently demodulates the complex envelope `y` of a signal using the frequency shift key method. `M` is the alphabet size and must be an integer power of 2. `freq_sep` is the frequency separation between successive frequencies in Hz. `nsamp` is the required number of samples per symbol and must be a positive integer greater than 1. The sampling frequency is 1 Hz. If `y` is a matrix with multiple rows and columns, then 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use 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; N = Fs;
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(N),...
   'measured',[],'dB'); % AWGN channel
msg_rrx = fskdemod(msg_rx,M,freqsep,nsamp,Fs);  % Demodulate
[num,SER] = symerr(msg,msg_rrx); % Symbol error rate
BER = SER*(M/2)/(M-1) % Bit error rate
```

```
BER_theory = berawgn(EbNo,'fsk',M,'noncoherent') % Theoretical BER
```

The output is below. Your BER value might vary because the example uses random numbers.

```
BER =

    0.1006


BER_theory =

    0.1029
```

See Also

`fskmod`, `pskmod`, `pskdemod`, Chapter 8, “Modulation”

Purpose

Frequency shift keying modulation

Syntax

```
y = fskmod(x,M,freq_sep,nsamp)
y = fskmod(x,M,freq_sep,nsamp,Fs)
y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)
y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)
```

Description

`y = fskmod(x,M,freq_sep,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using frequency shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. `freq_sep` is the desired separation between successive frequencies in Hz. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer greater than 1. The sampling rate of `y` is 1 Hz. By the Nyquist sampling theorem, `freq_sep` and `M` must satisfy `(M-1)*freq_sep <= 1`. If `x` is a matrix with multiple rows and columns, then the function processes the columns independently.

`y = fskmod(x,M,freq_sep,nsamp,Fs)` specifies the sampling rate of `y` in Hz. Because the Nyquist sampling theorem implies that the maximum frequency must be no larger than `Fs/2`, the inputs must satisfy `(M-1)*freq_sep <= Fs`.

`y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)` specifies the phase continuity. Set `phase_cont` to `'cont'` to force phase continuity across symbol boundaries in `y`, or `'discont'` to avoid forcing phase continuity. The default is `'cont'`.

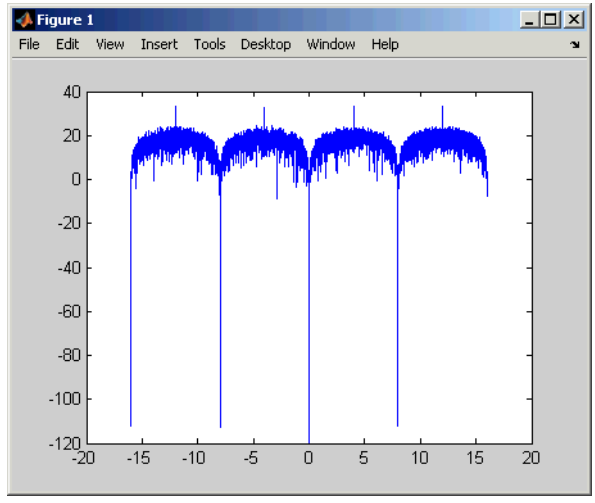
`y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray-coded ordering.

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`, Chapter 8, “Modulation”

Purpose

Convert between parity-check and generator matrices

Syntax

```
parmat = gen2par(genmat)
genmat = gen2par(parmat)
```

Description

`parmat = gen2par(genmat)` converts the standard-form binary generator matrix `genmat` into the corresponding parity-check matrix `parmat`.

`genmat = gen2par(parmat)` converts the standard-form binary parity-check matrix `parmat` into the corresponding generator matrix `genmat`.

The standard forms of the generator and parity-check matrices for an [n,k] binary linear block code are shown in the table below

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k\ P]$ or $[P\ I_k]$	k-by-n
Parity-check	$[-P'\ I_{n-k}]$ or $[I_{n-k}\ -P']$	(n-k)-by-n

.

where I_k is the identity matrix of size k and the `'` symbol indicates matrix transpose. Two standard forms are listed for each type, because different authors use different conventions. For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, -1 = 1 in the binary field.

Examples

The commands below convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

```
parmat = hammgen(3)
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” on page 6-2

Purpose

General quadrature amplitude demodulation

Syntax

```
z = genqamdemod(y,const)
```

Description

`z = genqamdemod(y,const)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. The complex vector `const` specifies the signal mapping. If `y` is a matrix with multiple rows, then 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`, Chapter 8, "Modulation"

genqammod

Purpose

General quadrature amplitude modulation

Syntax

```
y = genqammod(x,const)
```

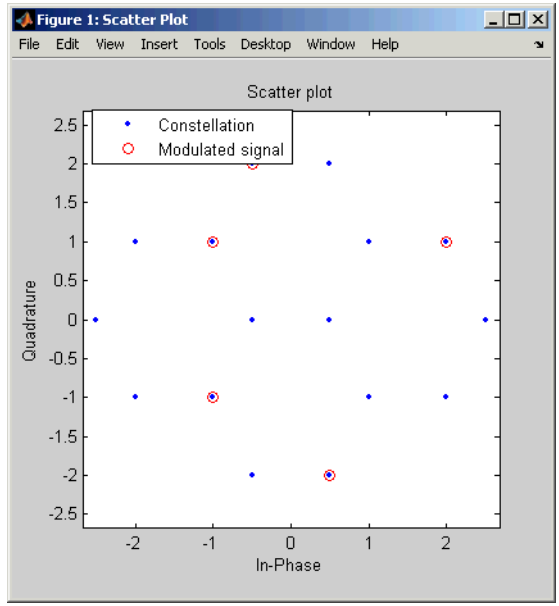
Description

`y = genqammod(x,const)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. The message signal must consist of integers between 0 and `length(const)-1`. The complex vector `const` specifies the signal mapping. If `x` is a matrix with multiple rows, then 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” on page 8-11.

See Also

genqamdemod, qammod, qamdemod, pammod, pamdemod, Chapter 8, “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 Chapter 12, "Galois Field Computations". To learn how the integers in `x` represent elements of GF(`2^m`), see "How Integers Correspond to Galois Field Elements" on page 12-7.

`x_gf = gf(x,m,prim_poly)` is the same as the previous syntax, except that 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 41 is 1 0 0 1 0 1. For more information about the primitive polynomial, see "Specifying the Primitive Polynomial" on page 12-9.

`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” on page 12-5

- "Example: Representing a Primitive Element" on page 12-8
- Other sample code within Chapter 12, "Galois Field Computations"
- The Galois field demonstration: type `playshow gfdemo`.

See Also

`gftable`, list of functions and operators for Galois field computations, `gfdemo`, Chapter 12, "Galois Field Computations"

Purpose

Add polynomials over a Galois field

Syntax

```
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 odd. To work in GF(2^m), apply the + operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction" on page 12-14.

`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, then 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), then extra zeros are added at the end; if it has more than `len` entries, then 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" on page 13-4 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

In the code below, `sum5` is the sum of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and `linpart` is the degree-one part of `sum5`.

```
sum5 = gfadd([2 3 1],[4 2 3],5)
```

```
linpart = gfadd([2 3 1],[4 2 3],5,2)
```

The output is

```
sum5 =

     1     0     4


linpart =

     1     0
```

The code below shows that $A^2 + A^4 = A^1$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
g = gfadd(2,4,field)
```

The output is

```
g =

     1
```

Other examples are in "Arithmetic in Galois Fields" on page 13-13.

See Also

`gfsub`, `gfconv`, `gfmul`, `gfdeconv`, `gfdiv`, `gftuple`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Multiply polynomials over a Galois field

Syntax

```
c = gfconv(a,b,p)
c = gfconv(a,b,field)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `conv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials" on page 12-30.

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,p)` multiplies two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1.

`c = gfconv(a,b,field)` multiplies two GF(p^m) polynomials, where p is a prime number and m is a positive integer. `a`, `b`, and `c` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" on page 13-4 for an explanation of these formats.

Examples

The command below shows that

$$(1 + x + x^4)(x + x^2) = x + 2x^2 + x^3 + x^5 + x^6$$

over GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)
```

The output is

```
gfc =

     0     1     2     1     0     1     1
```

The code below illustrates the identity

$$(x^r + x^s)^p = x^{rp} + x^{sp}$$

for the case in which $p = 7$, $r = 5$, and $s = 3$. (The identity holds when p is any prime number, and r and s are positive integers.)

```
p = 7; r = 5; s = 3;
a = gfrepcov([r s]); % x^r + x^s

% Compute a^p over GF(p).
c = 1;
for ii = 1:p
   c = gfconv(c,a,p);
end;

% Check whether c = x^(rp) + x^(sp).
powers = [];
for ii = 1:length(c)
   if c(ii)~=0
      powers = [powers, ii];
   end;
end;
if (powers==[r*p+1 s*p+1] | powers==[s*p+1 r*p+1])
   disp('The identity is proved for this case of r, s, and p.')
end
```

See Also

`gfdeconv`, `gfadd`, `gfsub`, `gfmul`, `gftuple`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Produce cyclotomic cosets for a Galois field

Syntax

```
c = gfcosets(m,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `cosets` function.

`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" on page 13-4.

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 15-140 below.

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`, Chapter 13, "Galois Fields of Odd Characteristic"

References

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

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

Purpose

Divide polynomials over a Galois field

Syntax

```
[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 odd. To work in GF(2^m), use the `deconv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials" on page 12-30.

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,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" on page 13-4 for an explanation of these formats.

Examples

The code below shows that

$$(x + x^3 + x^4) \div (1 + x) = 1 + x^3 \text{ Remainder } 2$$

in GF(3). It also checks the results of the division.

```
p = 3;
b = [0 1 0 1 1]; a = [1 1];
[quot, remd] = gfdeconv(b,a,p)
% Check the result.
bnew = gfadd(gfconv(quot,a,p),remd,p);
if isequal(bnew,b)
   disp('Correct.')
end;
```

The output is below.

```
quot =

     1     0     0     1

remd =

     2

Correct.
```

Working over GF(3), the code below outputs those polynomials of the form $x^k - 1$ (k = 2, 3, 4,..., 8) that $1 + x^2$ divides evenly.

```
p = 3; m = 2;
a = [1 0 1]; % 1+x^2
for ii = 2:p^m-1
   b = gfrepcov(ii); % x^ii
   b(1) = p-1; % -1+x^ii
   [quot, remd] = gfdeconv(b,a,p);
   % Display -1+x^ii if a divides it evenly.
   if remd==0
      multiple{ii}=b;
      gfpretty(b)
   end
end
```

The output is below.

```
     4
2 + X

     8
2 + X
```

In light of the discussion in "Algorithm" on page 15-155 on the reference page for `gfprimck` along with the irreducibility of $1 + x^2$ over GF(3), this output indicates that $1 + x^2$ is not primitive for GF(9).

Algorithm

The algorithm of `gfdeconv` is similar to that of the MATLAB function `deconv`.

See Also

`gfconv`, `gfadd`, `gfsub`, `gfdiv`, `gftuple`, Chapter 13, "Galois Fields of Odd Characteristic"

gfdiv

Purpose

Divide elements of a Galois field

Syntax

```
quot = gfdiv(b,a,p)
quot = gfdiv(b,a,field)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the `./` operator to Galois arrays. For details, see "Example: Division" on page 12-16.

The `gfdiv` function divides elements of a Galois field. (To divide polynomials over a Galois field, use `gfdeconv` instead.)

`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, then 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" on page 13-4 for an explanation of these formats.

In all cases, an attempt to divide by the zero element of the field results in a "quotient" of `NaN`.

Examples

The code below displays lists of multiplicative inverses in GF(5) and GF(25). It uses column vectors as inputs to `gfdiv`.

```
% Find inverses of nonzero elements of GF(5).
p = 5;
b = ones(p-1,1);
a = [1:p-1]';
```

```
quot1 = gfdiv(b,a,p);
disp('Inverses in GF(5):')
disp('element  inverse')
disp([a, quot1])

% Find inverses of nonzero elements of GF(25).
m = 2;
field = gftuple([-1:p^m-2]',m,p);
b = zeros(p^m-1,1); % Numerator is zero since 1 = alpha^0.
a = [0:p^m-2]';
quot2 = gfdiv(b,a,field);
disp('Inverses in GF(25), expressed in EXPONENTIAL FORMAT with')
disp('respect to a root of the default primitive polynomial:')
disp('element  inverse')
disp([a, quot2])
```

See Also

`gfmul`, `gfdeconv`, `gfconv`, `gftuple`, Chapter 13, “Galois Fields of Odd Characteristic”

Purpose

Filter data using polynomials over a prime Galois field

Syntax

```
y = gffilter(b,a,x,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `filter` function with Galois arrays. For details, see "Filtering" on page 12-27.

`y = gffilter(b,a,x,p)` filters the data `x` using the filter described by vectors `a` and `b`. `y` is the filtered data in GF(`p`). `p` is a prime number, and all entries of `a` and `b` are between 0 and `p-1`.

By definition of the filter, `y` solves the difference equation

```
a(1)y(n) = b(1)x(n)+b(2)x(n-1)+b(3)x(n-2)+...+b(B+1)x(n-B)
                      -a(2)y(n-1)-a(3)y(n-2)-...-a(A+1)y(n-A)
```

where

- `A+1` is the length of the vector `a`
- `B+1` is the length of the vector `b`
- `n` varies between 1 and the length of the vector `x`.

The vector `a` represents the degree-n_a polynomial

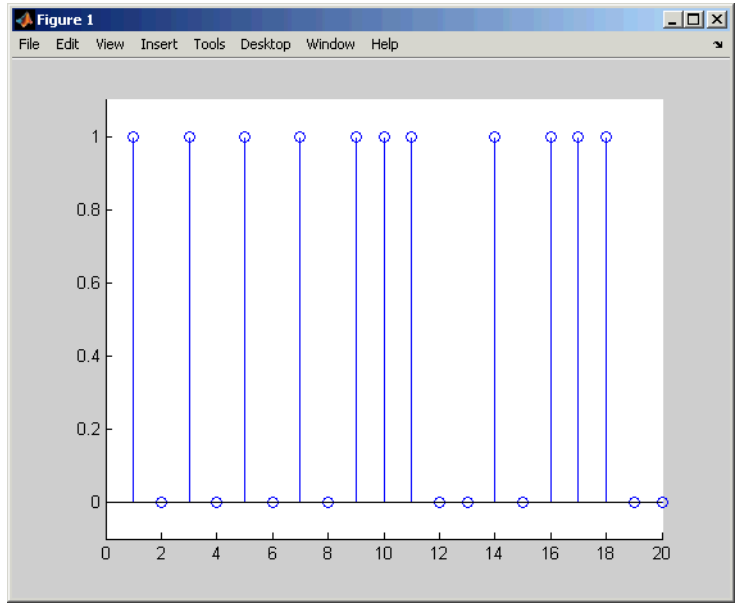
```
a(1)+a(2)x+a(3)x^2+...+a(A+1)x^A
```

Examples

The impulse response of a particular filter is given in the code and diagram below.

```
b = [1 0 0 1 0 1 0 1];
a = [1 0 1 1];
y = gffilter(b,a,[1,zeros(1,19)]);
stem(y);
```

```
axis([0 20 -.1 1.1])
```



See Also

`gfconv`, `gfadd`, `filter`, Chapter 13, "Galois Fields of Odd Characteristic"

gflineq

Purpose

Find particular solution of `Ax = b` over a prime Galois field

Syntax

```
x = gflineq(A,b,p)
[x,vld] = gflineq(...)
```

Description

Note This function performs computations in GF(p) where p is odd. To work in GF(2^m), apply the `\` or `/` operator to Galois arrays. For details, see "Solving Linear Equations" on page 12-25.

`x = gflineq(A,b,p)` returns a particular solution of the linear equation `A x = b` over GF(`p`), where `p` is a prime number. If `A` is a k-by-n matrix and `b` is a vector of length k, then `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, then `x` is empty.

`[x,vld] = gflineq(...)` returns a flag `vld` that indicates the existence of a solution. If `vld = 1`, then the solution `x` exists and is valid; if `vld = 0`, then no solution exists.

Examples

The code below produces some valid solutions of a linear equation over GF(3).

```
A = [2 0 1;
     1 1 0;
     1 1 2];
% An example in which the solutions are valid
[x,vld] = gflineq(A,[1;0;0],3)
```

The output is below.

```
x =

     2
     1
     0
```

```
vld =

     1
```

By contrast, the command below finds that the linear equation has *no* solutions.

```
[x2,vld2] = gflineq(zeros(3,3),[2;0;0],3)
```

The output is below.

```
This linear equation has no solution.

x2 =

     []


vld2 =

     0
```

Algorithm

`gflineq` uses Gaussian elimination.

See Also

`gfadd`, `gfdiv`, `gfroots`, `gfrank`, `gfconv`, `conv`, Chapter 13, "Galois Fields of Odd Characteristic"

gfminpol

Purpose

Find minimal polynomial of a Galois field element

Syntax

```
pol = gfminpol(k,m,p)
pol = gfminpol(k,prim_poly,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `minpol` function with Galois arrays. For details, see "Minimal Polynomials" on page 12-33.

`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, then `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, then `pol` is a matrix having *len* rows; the rth row of `pol` gives the coefficients of the minimal polynomial of $A^{k(r)}$ in order of ascending powers.

`pol = gfminpol(k,prim_poly,p)` is the same as the first syntax listed, except that A is a root of the primitive polynomial for GF(p^m) specified by `prim_poly`. `prim_poly` is a row vector that gives the coefficients of the degree-m primitive polynomial in order of ascending powers.

Examples

The syntax `gfminpol(k,m,p)` is used in the sample code in "Characterization of Polynomials" on page 13-17.

See Also

`gfprimdf`, `gfcosets`, `gfroots`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Multiply elements of a 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 odd. To work in GF(2^m), apply the `.*` operator to Galois arrays. For details, see "Example: Multiplication" on page 12-15.

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, then 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" on page 13-4 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

"Arithmetic in Galois Fields" on page 13-13 contains examples. Also, the code below shows that

$$A^2 \cdot A^4 = A^6$$

where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

```
a = gfmul(2,4,field)
```

The output is

```
a =

     6
```

See Also

`gfdiv`, `gfdeconv`, `gfadd`, `gfsub`, `gftuple`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Display 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, then 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

                                        3    4
                                   2 + 2 X  + X

then the element

                                       22
                                      A

can also be expressed as

                                       2    3
                                   2 + A  + A
```

See Also

`gftuple`, `gfprimdf`, Chapter 13, “Galois Fields of Odd Characteristic”

Purpose

Check whether polynomial over a Galois field is primitive

Syntax

```
ck = gfprimck(a,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `isprimitive` function. For details, see "Finding Primitive Polynomials" on page 12-10.

`ck = gfprimck(a,p)` returns a flag `ck` that indicates whether a polynomial over GF(`p`) is irreducible or primitive. `a` is a row vector that gives the coefficients of the polynomial in order of ascending powers. Each coefficient is between 0 and `p`-1. If m is the degree of the polynomial, then the output `ck` is

- -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)

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" on page 13-17 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`, Chapter 13, "Galois Fields of Odd Characteristic"

References

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

[1] Krogsgaard, K. and Karp, T., *Fast Identification of Primitive Polynomials over Galois Fields: Results from a Course Project*, ICASSP 2005, Philadelphia, PA, 2004.

Purpose

Provide default primitive polynomials for a Galois field

Syntax

```
pol = gfprimdf(m,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials" on page 12-10.

`pol = gfprimdf(m,p)` returns the row vector that gives the coefficients, in order of ascending powers, of the default primitive polynomial for GF(`p^m`). `m` is a positive integer and `p` is a prime number.

Examples

The command below shows that $2 + x + x^2$ is the default primitive polynomial for GF(5^2).

```
pol = gfprimdf(2,5)
pol =

     2     1     1
```

The code below displays the default primitive polynomial for each of the fields GF(3^m), where m ranges between 3 and 5.

```
for m = 3:5
   gfpretty(gfprimdf(m,3))
end
```

The output is below.

```
                  3
1 + 2 X + X

                4
 2 + X + X
```

```
                                5
1 + 2 X + X
```

See Also

`gfprimck`, `gfprimfd`, `gftuple`, `gfminpol`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Find primitive polynomials for a Galois field

Syntax

```
pol = gfprimfd(m,opt,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials" on page 12-10.

- If `m` = 1, then `pol` = [1 1].
- A polynomial is represented as a row containing the coefficients in order of ascending powers.

`pol = gfprimfd(m,opt,p)` searches for one or more primitive polynomials for GF(`p^m`), where `p` is a prime number and `m` is a positive integer. If `m` = 1, then `pol` = [1 1]. If `m` > 1, then the output `pol` depends on the argument `opt` as shown in the table below. Each polynomial is represented in `pol` as a row containing the coefficients in order of ascending powers.

opt	Significance of pol	Format of pol
`'min'`	One primitive polynomial for GF(`p^m`) having the smallest possible number of nonzero terms	The row vector representing the polynomial
`'max'`	One primitive polynomial for GF(`p^m`) having the greatest possible number of nonzero terms	The row vector representing the polynomial

opt	Significance of pol	Format of pol
`'all'`	All primitive polynomials for GF(`p^m`)	A matrix, each row of which represents one such polynomial
A positive integer	All primitive polynomials for GF(`p^m`) that have `opt` nonzero terms	A matrix, each row of which represents one such polynomial

Examples

The code below seeks primitive polynomials for GF(81) having various other properties. Notice that `fourterms` is empty because no primitive polynomial for GF(81) has exactly four nonzero terms. Also notice that `fewterms` represents a *single* polynomial having three terms, while `threeterms` represents *all* of the three-term primitive polynomials for GF(81).

```
p = 3; m = 4; % Work in GF(81).
fewterms = gfprimfd(m,'min',p)
threeterms = gfprimfd(m,3,p)
fourterms = gfprimfd(m,4,p)
```

The output is below.

```
fewterms =

     2     1     0     0     1


threeterms =

     2     1     0     0     1
     2     2     0     0     1
     2     0     0     1     1
     2     0     0     2     1
```

```
No primitive polynomial satisfies the given constraints.

fourterms =

     []
```

Algorithm

`gfprimfd` tests for primitivity using `gfprimck`. If `opt` is `'min'`, `'max'`, or omitted, then 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`, Chapter 13, "Galois Fields of Odd Characteristic"

gfrank

Purpose

Compute rank of matrix over a Galois field

Syntax

```
rk = gfrank(A,p)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `rank` function with Galois arrays. For details, see "Computing Ranks" on page 12-24.

`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 = gfrepcov(poly2)
```

Description

Two logical ways to represent polynomials over GF(2) are listed below:

1 `[A_0 A_1 A_2 ... A_(m-1)]` represents the polynomial

$$A\_0 + A\_1x + A\_2x^2 + \cdots + A\_(m\text{-}1)x^{m-1}$$

Each entry `A_k` is either one or zero.

2 [A_0 A_1 A_2 ... A_(m-1)] represents the polynomial

$$x^{A\_0} + x^{A\_1} + x^{A\_2} + \cdots + x^{A\_(m\text{-}1)}$$

Each entry `A_k` is a nonnegative integer. All entries must be distinct.

Format **1** is the standard form used by the Galois field functions in this toolbox, but there are some cases in which format **2** is more convenient.

`polystandard = gfrepcov(poly2)` converts from the second format to the first, for polynomials of degree *at least* 2. `poly2` and `polystandard` are row vectors. The entries of `poly2` are distinct integers, and at least one entry must exceed 1. Each entry of `polystandard` is either 0 or 1.

Note If `poly2` is a *binary* row vector, then `gfrepcov` assumes that it is already in Format 1 above and returns it unaltered.

Examples

The command below converts the representation format of the polynomial $1 + x^2 + x^5$.

```
polystandard = gfrepcov([0 2 5])
```

```
polystandard =

     1     0     1     0     0     1
```

See Also

`gfpretty`, Chapter 13, “Galois Fields of Odd Characteristic”

Purpose

Find roots of polynomial over a prime Galois field

Syntax

```
rt = gfroots(f,m,p)
rt = gfroots(f,prim_poly,p)
[rt,rt_tuple] = gfroots(...)
[rt,rt_tuple,field] = gfroots(...)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `roots` function with Galois arrays. For details, see "Roots of Polynomials" on page 12-32.

For all syntaxes, `f` is a row vector that gives the coefficients, in order of ascending powers, of a degree-d polynomial.

Note `gfroots` lists each root exactly once, ignoring multiplicities of roots.

`rt = gfroots(f,m,p)` finds roots in GF(`p^m`) of the polynomial that `f` represents. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the default primitive polynomial for GF(`p^m`).

`rt = gfroots(f,prim_poly,p)` finds roots in GF(p^m) of the polynomial that `f` represents. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the degree-m primitive polynomial for GF(p^m) that `prim_poly` represents.

`[rt,rt_tuple] = gfroots(...)` returns an additional matrix `rt_tuple`, whose kth row is the polynomial format of the root `rt(k)`. The polynomial and exponential formats are both relative to the same primitive element.

`[rt,rt_tuple,field] = gfroots(...)` returns additional matrices `rt_tuple` and `field`. `rt_tuple` is described in the paragraph above. `field` gives the list of elements of the extension field. The list of elements, the polynomial format, and the exponential format are all relative to the same primitive element.

Note For a description of the various formats that `gfroots` uses, see "Representing Elements of Galois Fields" on page 13-4.

Examples

"Roots of Polynomials" on page 13-18 contains a description and example of the use of `gfroots`.

As another example, the code below finds the polynomial format of the roots of the primitive polynomial $2 + x^3 + x^4$ for GF(81). It then displays the roots in traditional form as polynomials in `alph`. (The output is omitted here.) Because `prim_poly` is both the primitive polynomial and the polynomial whose roots are sought, `alph` itself is a root.

```
p = 3; m = 4;
prim_poly = [2 0 0 1 1]; % A primitive polynomial for GF(81)
f = prim_poly; % Find roots of the primitive polynomial.
[rt,rt_tuple] = gfroots(f,prim_poly,p);
% Display roots as polynomials in alpha.
for ii = 1:length(rt_tuple)
  gfpretty(rt_tuple(ii,:),'alpha')
end
```

See Also

`gfprimdf`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Subtract polynomials over a Galois field

Syntax

```
c = gfsub(a,b,p)
c = gfsub(a,b,p,len)
c = gfsub(a,b,field)
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the - operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction" on page 12-14.

`c = gfsub(a,b,p)` calculates `a` minus `b`, where `a` and `b` represent polynomials over GF(`p`) and `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1. If `a` and `b` are matrices of the same size, then the function treats each row independently.

`c = gfsub(a,b,p,len)` subtracts row vectors as in the syntax above, except that it returns a row vector of length `len`. The output `c` is a truncated or extended representation of the answer. If the row vector corresponding to the answer has fewer than `len` entries (including zeros), then extra zeros are added at the end; if it has more than `len` entries, then entries from the end are removed.

`c = gfsub(a,b,field)` calculates `a` minus `b`, where `a` and `b` are the exponential format of two elements of GF(p^m), relative to some primitive element of GF(p^m). `p` is a prime number and `m` is a positive integer. `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the answer, relative to the same primitive element. See "Representing Elements of Galois Fields" on page 13-4 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

In the code below, `differ` is the difference of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and `linpart` is the degree-one part of `differ`.

```
differ = gfsub([2 3 1],[4 2 3],5)
linpart = gfsub([2 3 1],[4 2 3],5,2)
```

The output is

```
differ =

     3     1     3


linpart =

     3     1
```

The code below shows that $A^2 - A^4 = A^7$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
d = gfsub(2,4,field)
```

The output is

```
d =

     7
```

See Also

`gfadd`, `gfconv`, `gfmul`, `gfdeconv`, `gfdiv`, `gftuple`, Chapter 13, "Galois Fields of Odd Characteristic"

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" on page 12-9. 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, then 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 would expect `t3` to be similar to `t1` and 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" on page 12-38

gftrunc

Purpose

Minimize the length of a 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`, then 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`, Chapter 13, "Galois Fields of Odd Characteristic"

Purpose

Simplify or convert the format of elements of a Galois field

Syntax

```
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 odd. To perform equivalent computations in GF(2^m), apply the `.^` operator and the `log` function to Galois arrays. For more information, see "Example: Exponentiation" on page 12-17 and "Example: Elementwise Logarithm" on page 12-18, respectively.

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" on page 13-4.

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, then 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, then 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,p)` returns the simplest polynomial format of the elements that `a` represents, where the kth row of `tp` corresponds to the kth row of `a`. The formats are relative to a root of the default primitive polynomial for GF(`p^m`). `m` is a positive integer and `p` is a prime number. If possible, the default primitive polynomial is used to simplify the polynomial formats.

`tp = gftuple(a,prim_poly,p)` returns the simplest polynomial format of the element that `a` represents, where the kth row of `tp` corresponds to the kth row of `a`. `p` is a prime number. The formats are relative to a root of the primitive polynomial whose coefficients are given, in order of ascending powers, by the row vector `prim_poly`. If possible, the function uses this primitive polynomial to simplify the polynomial formats.

`tp = gftuple(a,prim_poly,p,prim_ck)` is the same as `tp = gftuple(a,prim_poly,p)` except that `gftuple` checks whether `prim_poly` represents a polynomial that is indeed primitive. If not, then `gftuple` generates an error and `tp` is not returned. The input argument `prim_ck` can be any number or string; only its existence matters.

`[tp,expform] = gftuple(...)` returns the additional matrix `expform`. The kth row of `expform` is the simplest exponential format of the element that the kth row of `a` represents. All other features are as described in earlier parts of this "Description" section, depending on the input arguments.

Examples

Some examples are in these subsections of Chapter 13, “Galois Fields of Odd Characteristic”:

- “List of All Elements of a Galois Field” on page 13-5 (end of section)
- “Converting to Simplest Polynomial Format” on page 13-9
- “Converting to Simplest Exponential Format” on page 13-11

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`, `gfdiv`, `gfdeconv`, `gfprimdf`, Chapter 13, "Galois Fields of Odd Characteristic"

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" on page 6-2

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`.

Examples

To Gray-decode a vector `x` with a 64-QAM Gray-encoded constellation, use

```
y = gray2bin(x,'qam',64);
```

To Gray-decode a vector `x` with a 64-FSK Gray-encoded constellation and return its map, use

```
[y,map] = gray2bin(x,'fsk',64);
```

To Gray-decode a vector `x` with a 256-DPSK Gray-encoded constellation, use

```
y = gray2bin(x,'dpsk',256);
```

To Gray-decode a vector `x` with a 1024-PSK Gray-encoded constellation, use

```
y = gray2bin(x,'psk',1024);
```

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`), then the syntax `hammgen(m)` is likely to be faster than the syntax `hammgen(m,pol)`.

Examples

The command below exhibits the parity-check and generator matrices for a Hamming code with codeword length $7 = 2^3-1$ and message length $4 = 7-3$.

```
[h,g,n,k] = hammgen(3)

h =

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


g =

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


n =

     7


k =

     4
```

The command below, which uses $1 + x^2 + x^3$ as the primitive polynomial for GF(2^3), shows that the parity-check matrix depends on the choice of primitive polynomial. Notice that `h1` below is different from `h` in the example above.

```
h1 = hammgen(3,[1 0 1 1])
```

```
h1 =

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

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" on page 6-2

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, then `tol` is the order of the conversion. If `tol` < 1, then `tol` is the tolerance in selecting the conversion order based on the singular values. If you omit `tol`, then its default value is 0.01. This conversion uses the singular value decomposition method.

`[num,den,sv] = hank2sys(h,ini,tol)` returns a vector `sv` that lists the singular values of `h`.

`[a,b,c,d] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a corresponding linear system state-space model. `a`, `b`, `c`, and `d` are matrices. The input parameters are the same as in the first syntax above.

`[a,b,c,d,sv] = hank2sys(h,ini,tol)` is the same as the syntax above, except that `sv` is a vector that lists the singular values of `h`.

Examples

```
h = hankel([1 0 1]);
[num,den,sv] = hank2sys(h,0,.01)
```

The output is

```
num =

         0    1.0000    0.0000    1.0000
```

```
den =

    1.0000    0.0000    0.0000    0.0000


sv =

    1.6180
    1.0000
    0.6180
```

See Also

`rcosflt`, `hankel`

Purpose

Restore ordering of symbols permuted using `helintrlv`

Syntax

```
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,init_state)
deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)
```

Description

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)` restores the ordering of symbols in `data` by placing them in an array row by row and then selecting groups in a helical fashion to place in the output, `deintrlved`. `data` must have `col*ngrp` elements. If `data` is a matrix with multiple rows and columns, then it must have `col*ngrp` rows, and the function processes the columns independently. `state` is a structure that holds the final state of the array. `state.value` stores input symbols that remain in the `col` columns of the array and do not appear in the output.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and `col` columns. The function initializes the top of the array with zeros. It then places `col*ngrp` symbols from the input into the next `ngrp` rows of the array. The function places symbols from the array in the output, `intrlved`, placing `ngrp` symbols at a time; the kth group of `ngrp` symbols comes from the kth column of the array, starting from row 1+(k-1)*`stp`. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,init_state)` initializes the array with the symbols contained in `init_state.value` instead of zeros. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver. In this syntax, some output symbols are default values of 0, some are input symbols from `data`, and some are initialization values from `init_state.value`.

`deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)` is the same as the syntax above, except that it does not record the deinterleaver's final state. This syntax is appropriate for the last in a

series of calls to this function. However, if you plan to call this function again to continue the deinterleaving process, then the syntax above is more appropriate.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `helintrlv` function, use the same `col`, `ngrp`, and `stp` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helintrlv` followed by `heldeintrlv` leaves data unchanged, after you take their combined delay of `col*ngrp*ceil(stp*(col-1)/ngrp)` into account. To learn more about delays of convolutional interleavers, see "Delays of Convolutional Interleavers" on page 7-9.

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,ngrp,stp);
[d2,dstate] = heldeintrlv(i2,col,ngrp,stp,dstate);
d3 = heldeintrlv(i3,col,ngrp,stp,dstate);

% Check the results.
d0 = [d1; d2; d3]; % All the deinterleaved data
d0_trunc = d0(delayval+1:end); % Remove the delay.
ser = symerr(data,d0_trunc)
```

The output below shows that no symbol errors occurred.

```
ser =

     0
```

See Also

`helintrlv`, Chapter 7, “Interleaving”

helintrlv

Purpose

Permute symbols using helical array

Syntax

```
intrlved = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state)
```

Description

`intrlved = helintrlv(data,col,ngrp,stp)` permutes the symbols in `data` by placing them in an unlimited-row array in helical fashion and then placing rows of the array in the output, `intrlved`. `data` must have `col*ngrp` elements. If `data` is a matrix with multiple rows and columns, then it must have `col*ngrp` rows, and the function processes the columns independently.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and `col` columns. The function partitions `col*ngrp` symbols from the input into consecutive groups of `ngrp` symbols. The function places the kth group in the array along column k, starting from row 1+(k-1)*`stp`. Positions in the array that do not contain input symbols have default values of 0. The function places `col*ngrp` symbols from the array in the output, `intrlved`, by reading the first `ngrp` rows sequentially. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

`[intrlved,state] = helintrlv(data,col,ngrp,stp)` returns a structure that holds the final state of the array. `state.value` stores input symbols that remain in the `col` columns of the array and do not appear in the output.

`[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state)` initializes the array with the symbols contained in `init_state.value`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver. In this syntax, some output symbols are default values of 0, some are input symbols from `data`, and some are initialization values from `init_state.value`.

Examples

The example below rearranges the integers from 1 to 24.

```
% Interleave some symbols. Record final state of array.
[i1,state] = helintrlv([1:12]',3,4,1);
% Interleave more symbols, remembering the symbols that
% were left in the array from the earlier command.
i2 = helintrlv([13:24]',3,4,1,state);

disp('Interleaved data:')
disp([i1,i2]')
disp('Values left in array after first interleaving operation:')
state.value{:}
```

During the successive calls to `helintrlv`, it internally creates the three-column arrays

```
[1  0  0;
 2  5  0;
 3  6  9;
 4  7 10;
 0  8 11;
 0  0 12]
```

and

```
[13  8 11;
 14 17 12;
 15 18 21;
 16 19 22;
  0 20 23;
  0  0 24]
```

In the second array shown above, the 8, 11, and 12 are values left in the array from the previous call to the function. Specifying the `init_state` input in the second call to the function causes it to use those values rather than default values of 0.

The output from this example is below. (The actual interleaved data is a tall matrix, but 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`, Chapter 7, “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, then it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, then `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-4 diagonals. The function then sends the elements, row by row, to the output `d`.

See Also

`helscanintrlv`, Chapter 7, “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, then it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, then `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`, Chapter 7, “Interleaving”

Purpose

Design a Hilbert transform IIR filter

Syntax

```
hilbiir
hilbiir(ts)
hilbiir(ts,dly)
hilbiir(ts,dly,bandwidth)
hilbiir(ts,dly,bandwidth,tol)
[num,den] = hilbiir(...)
[num,den,sv] = hilbiir(...)
[a,b,c,d] = hilbiir(...)
[a,b,c,d,sv] = hilbiir(...)
```

Description

The function `hilbiir` designs a Hilbert transform filter. The output is either

- A plot of the filter's impulse response, or
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Background Information

An ideal Hilbert transform filter has the transfer function H(s) = -jsgn(s), where sgn(.) is the signum function (`sign` in MATLAB). The impulse response of the Hilbert transform filter is

$$h(t) = \frac{1}{\pi t}$$

Because the Hilbert transform filter is a noncausal filter, the `hilbiir` function introduces a group delay, `dly`. A Hilbert transform filter with this delay has the impulse response

$$h(t) = \frac{1}{\pi(t - \mathrm{dly})}$$

Choosing a Group Delay Parameter

The filter design is an approximation. If you provide the filter's group delay as an input argument, then these two suggestions can help improve the accuracy of the results:

- Choose the sample time `ts` and the filter's group delay `dly` so that `dly` is at least a few times larger than `ts` and `rem(dly,ts) = ts/2`. For example, you can set `ts` to `2*dly/N`, where `N` is a positive integer.
- At the point `t = dly`, the impulse response of the Hilbert transform filter can be interpreted as `0`, `-Inf`, or `Inf`. If `hilbiir` encounters this point, then 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 1-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 1-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 15-195 above for guidelines on choosing `dly`.

`hilbiir(ts,dly,bandwidth)` is the same as the syntax above, except that `bandwidth` specifies the assumed bandwidth of the input signal

and that the filter design might use a compensator for the input signal. If `bandwidth` = 0 or `bandwidth` > 1/(2*`ts`), then `hilbiir` does not use a compensator.

`hilbiir(ts,dly,bandwidth,tol)` is the same as the syntax above, except that `tol` is the tolerance index. If `tol` < 1, then the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < \texttt{tol}$$

If `tol` > 1, then 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 the "Syntaxes for Plots" on page 15-195 section above.

`[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 listed below.

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`, Chapter 9, "Special Filters"

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, N.J., 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, then `dsig` is a vector; if any signal value in `dict` is alphabetical, then `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" on page 5-14

References

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

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, then it must be either a row or a column.
- `p`, a probability vector whose kth element is the probability with which the source produces the kth element of `symbols`. The length of `p` must equal the length of `symbols`.

The outputs of `huffmandict` are

- `dict`, a two-column cell array in which the first column lists the distinct signal values from `symbols` and the second column lists the corresponding Huffman codewords. In the second column, each Huffman codeword is represented as a numeric row vector.
- `avglen`, the average length among all codewords in the dictionary, weighted according to the probabilities in the vector `p`.

For Specific Syntaxes

`[dict,avglen] = huffmandict(symbols,p)` generates a binary Huffman code dictionary using the maximum variance algorithm.

`[dict,avglen] = huffmandict(symbols,p,N)` generates an `N`-ary Huffman code dictionary using the maximum variance algorithm. `N` is

an integer between 2 and 10 that must not exceed the number of source symbols whose probabilities appear in the vector `p`.

`[dict,avglen] = huffmandict(symbols,p,N,variance)` generates an N-ary Huffman code dictionary with the minimum variance if `variance` is `'min'` and the maximum variance if `variance` is `'max'`. `N` is an integer between 2 and 10 that must not exceed the length of the vector `p`.

Examples

```
symbols = [1:5];
p = [.3 .3 .2 .1 .1];
[dict,avglen] = huffmandict(symbols,p)
samplecode = dict{5,2} % Codeword for fifth signal value
```

The output is below, where the first column of `dict` lists the values in `symbols` and the second column lists the corresponding codewords.

```
dict =

    [1]    [1x2 double]
    [2]    [1x2 double]
    [3]    [1x2 double]
    [4]    [1x3 double]
    [5]    [1x3 double]


avglen =

    2.2000


samplecode =

     1     1     0
```

See Also

`huffmanenco`, `huffmandeco`, “Huffman Coding” on page 5-14

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, then 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` may 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” on page 5-14

References

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

Purpose

Inverse discrete Fourier transform

Syntax

```
ifft(x)
```

Description

`ifft(x)` is the inverse discrete Fourier transform (DFT) of the Galois vector `x`. If `x` is in the Galois field GF(2^m), then the length of `x` must be 2^m-1.

Examples

For an example using `ifft`, see the reference page for `fft`.

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `x` must be in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithm

If `x` is a column vector, then `ifft` applies `dftmtx` to the multiplicative inverse of the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`fft`, `dftmtx`, “Signal Processing Operations in Galois Fields” on page 12-27

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, then the function processes the symbols independently. If `x` is a matrix with multiple rows, then the function treats each column as a channel and processes the columns independently.

Examples

An example in "Combining Pulse Shaping and Filtering with Modulation" on page 8-10 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`

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, then `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, then 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`, Chapter 7, “Interleaving”

Purpose

True for trellis corresponding to a catastrophic convolutional code

Syntax

```
iscatastrophic(s)
```

Description

`iscatastrophic(s)` returns `true` if the trellis `s` corresponds to a convolutional code that will cause catastrophic error propagation. It will otherwise return `false`.

See Also

`convenc`, `istrellis`, `poly2trellis`, `struct`, "Convolutional Coding" on page 6-30

isprimitive

Purpose

True for primitive polynomial for a 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`, then 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`, Chapter 12, “Galois Field Computations”

Purpose

True for valid trellis structure

Syntax

```
[isok,status] = istrellis(s)
```

Description

`[isok,status] = istrellis(s)` checks if the input `s` is a valid trellis structure. If the input is a valid trellis structure, then `isok` is 1 and `status` is an empty string. Otherwise, `isok` is 0 and `status` is a string that indicates why `s` is not a valid trellis structure.

A valid trellis structure is a MATLAB structure whose fields are as in the table below.

Fields of a Valid Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

In the `nextStates` matrix, each entry is an integer between 0 and `numStates`-1. The element in the sth row and uth column denotes the

next state when the starting state is s-1 and the input bits have decimal representation u-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the `nextStates` matrix stores the next states when the current set of input values is {0,...,0,1}.

To convert the state to a decimal value, use this rule: If k exceeds 1, then the shift register that receives the first input stream in the encoder provides the least significant bits in the state number, while the shift register that receives the last input stream in the encoder provides the most significant bits in the state number.

In the `outputs` matrix, the element in the sth row and uth column denotes the encoder's output when the starting state is s-1 and the input bits have decimal representation u-1. To convert to decimal value, use the first output bit as the MSB.

Examples

These commands assemble the fields into a very simple trellis structure, and then verify the validity of the trellis structure.

```
trellis.numInputSymbols = 2;
trellis.numOutputSymbols = 2;
trellis.numStates = 2;
trellis.nextStates = [0 1;0 1];
trellis.outputs = [0 0;1 1];
[isok,status] = istrellis(trellis)
```

The output is below.

```
isok =

     1


status =

     ''
```

Another example of a trellis is in “Trellis Description of a Convolutional Encoder” on page 6-34.

See Also

`poly2trellis`, `struct`, `convenc`, `vitdec`, “Convolutional Coding” on page 6-30

Purpose

Construct a 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" on page 11-8.

`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. Note that `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" on page 11-14.

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" on page 11-5.) 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" on page 11-3.
`WeightInputs`	Vector of tap weight inputs. This is the set of u_i values in the schematic in "Symbol-Spaced Equalizers" on page 11-3.

Property	Description
`ResetBeforeFiltering`	If `1`, each call to `equalize` resets the state of `eqobj` before equalizing. If `0`, the equalization process maintains continuity from one call to the next.
`NumSamplesProcessed`	Number of samples the equalizer processed since the last reset. When you create or reset `eqobj`, this property value is `0`.
Properties specific to the adaptive algorithm represented by `alg`	See reference page for the adaptive algorithm function that created `alg`: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`.

Relationships Among Properties

If you change `nWeights`, then MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

Property	Adjusted Value
`Weights`	`zeros(1,nWeights)`
`WeightInputs`	`zeros(1,nWeights)`
`StepSize` (Variable-step-size LMS equalizers)	`InitStep*ones(1,nWeights)`
`InvCorrMatrix` (RLS equalizers)	`InvCorrInit*eye(nWeights)`

An example illustrating relationships among properties is in "Linked Properties of an Equalizer Object" on page 11-14.

Examples

For examples that use this function, see "Equalizing Using a Training Sequence" on page 11-17, "Example: Equalizing Multiple Times,

Varying the Mode" on page 11-20, and "Example: Adaptive Equalization Within a Loop" on page 11-23.

See Also

`lms`, `signlms`, `normlms`, `varlms`, `rls`, `cma`, `dfe`, `equalize`, Chapter 11, "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" on page 5-2, "Representing Codebooks" on page 5-2, 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 The relative change in distortion between iterations is less than 10^{-7}.
on page 219 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 case The relative change in distortion between iterations is less than 10^{-7}. on page 219 of "Algorithm" on page 15-219 below, `reldistor` is the relative change in distortion between the last two iterations. In case The distortion is less than `eps*max(training_set)`, where `eps` is the MATLAB floating-point relative accuracy.
on page 219 , `reldistor` is the same as `distor`.

Examples

The code below optimizes the quantization parameters for a sinusoidal transmission via a 3-bit channel. Because the typical data is sinusoidal, `training_set` is a sampled sine wave. Because the channel can transmit 3 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

1. The relative change in distortion between iterations is less than 10^{-7}.
2. The distortion is less than `eps*max(training_set)`, where `eps` is the MATLAB floating-point relative accuracy.

See Also

`quantiz`, `dpcmopt`, Chapter 5, “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.

lms

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" on page 11-8.

`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, while 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" on page 11-12.

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" on page 11-17, "Example: Equalizing Multiple Times,

Varying the Mode" on page 11-20, and "Example: Adaptive Equalization Within a Loop" on page 11-23.

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

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

where the * operator denotes the complex conjugate.

See Also

`signlms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `dfe`, `equalize`, Chapter 11, "Equalizers"

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Purpose

Logarithm in a Galois field

Syntax

```
y = log(x)
```

Description

`y = log(x)` computes the logarithm of each element in the Galois array `x`. That is, `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 = marcumq(a,b)
Q = marcumq(a,b,m)
```

Description

`Q = marcumq(a,b)` computes the Marcum Q function of `a` and `b`, defined by

$$Q(a,b) = \int_b^\infty x \exp\left(-\frac{x^2+a^2}{2}\right) I_0(ax)dx$$

where `a` and `b` are nonnegative real numbers. In this expression, I_0 is the modified Bessel function of the first kind of zero order.

`Q = marcumq(a,b,m)` computes the generalized Marcum Q, defined by

$$Q\ (a,b) = \frac{1}{a^{m-1}} \int_b^\infty x^m \exp\left(-\frac{x^2+a^2}{2}\right) I_{m-1}(ax)dx$$

where `a` and `b` are nonnegative real numbers, and `m` is a nonnegative integer. In this expression, $I_{\text{m-1}}$ is the modified Bessel function of the first kind of order m-1.

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] McGee, W. F., "Another Recursive Method of Computing the Q Function," *IEEE Transactions on Information Theory*, vol. IT-16, July, 1970, pp. 500-501.

Purpose

Convert mask vector to shift for a shift register configuration

Syntax

```
shift = mask2shift(prpoly,mask)
```

Description

`shift = mask2shift(prpoly,mask)` returns the shift that is equivalent to a mask, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `mask` input is a binary vector whose length is the degree of the primitive polynomial.

Note To save time, `mask2shift` does not check that `prpoly` is primitive. If it is not primitive, then 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, then the equivalent shift s solves the equation $A^s = m(A)$. To interpret the vector `mask` as a polynomial, treat `mask` as a list of coefficients in order of descending powers.

Examples

The first command below converts a mask of $x^3 + 1$ into an equivalent shift, for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$. The second command

shows that a mask of 1 is equivalent to a shift of 0. In both cases, notice that the length of the `mask` vector is one less than the length of the `prpoly` vector.

```
s = mask2shift([1 1 0 0 1],[1 0 0 1])
s2 = mask2shift([1 1 0 0 1],[0 0 0 1])
```

The output is below.

```
s =

     4


s2 =

     0
```

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 a 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, then it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, then `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `matintrlv` function, use the same `Nrows` and `Ncols` inputs in both functions. In that case, the two functions are inverses in the sense that applying `matintrlv` followed by `matdeintrlv` leaves data unchanged.

Examples

The code below illustrates the inverse relationship between `matintrlv` and `matdeintrlv`.

```
Nrows = 2; Ncols = 3;
data = [1 2 3 4 5 6; 2 4 6 8 10 12]';
a = matintrlv(data,Nrows,Ncols); % Interleave.
b = matdeintrlv(a,Nrows,Ncols) % Deinterleave.
```

The output below shows that `b` is the same as `data`.

```
b =

     1     2
     2     4
     3     6
     4     8
     5    10
     6    12
```

See Also `matintrlv`, Chapter 7, “Interleaving”

Purpose

Reorder symbols by filling a 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, then it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, then `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`, Chapter 7, “Interleaving”

minpol

Purpose

Find minimal polynomial of a Galois field element

Syntax

```
pl = minpol(x)
```

Description

`pl = minpol(x)` finds the minimal polynomial of each element in the Galois column vector `x`. The output `pl` is an array in GF(2). The kth row of `pl` lists the coefficients, in order of descending powers, of the minimal polynomial of the kth element of `x`.

Note The output is in GF(2) even if the input is in a different Galois field.

Examples

The code below uses `m = 4` and finds that the minimal polynomial of `gf(2,m)` is just the primitive polynomial used for the field GF(`2^m`). This is true for any value of `m`, not just the value used in the example.

```
m = 4;
A = gf(2,m)
pl = minpol(A)
```

The output is below. Notice that the row vector `[1 0 0 1 1]` represents the polynomial `D^4 + D + 1`.

```
A = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     2


pl = GF(2) array.

Array elements =

     1     0     0     1     1
```

Another example is in “Minimal Polynomials” on page 12-33.

See Also

`cosets`, “Polynomials over Galois Fields” on page 12-30

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” on page 12-25.

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, then `A \ B` is the same as `(A'*A) \ (A'*B)`.

If `A` is an M-by-N wide matrix where M < N, then `A \ B` is the same as `A' * ((A*A') \ B)`. This solution is not unique.

See Also

"Linear Algebra in Galois Fields" on page 12-23

Purpose

Equalize linearly modulated signal using Viterbi algorithm

Syntax

```
y = mlseeq(x,chcffs,const,tblen,opmode)
y = mlseeq(x,chcffs,const,tblen,opmode,nsamp)
y = mlseeq(...,'rst',nsamp,preamble,postamble)
y = mlseeq(...,'cont',nsamp,...
init_metric,init_states,init_inputs)
[y,final_metric,final_states,final_inputs] = ...
mlseeq(...,'cont',...)
```

Description

`y = mlseeq(x,chcffs,const,tblen,opmode)` equalizes the baseband signal vector `x` using the Viterbi algorithm. `chcffs` is a vector that represents the channel coefficients. `const` is a complex vector that lists the points in the ideal signal constellation, in the same sequence that the system's modulator uses. `tblen` is the traceback depth. The equalizer traces back from the state with the best metric. `opmode` denotes the operation mode of the equalizer, where the choices are described in the table below.

Value of opmode	Typical Usage
`'rst'`	Enables you to specify a preamble and postamble that accompany your data. The function processes `x` independently of data from any other invocations of this function. This mode incurs no output delay.
`'cont'`	Enables you to save the equalizer's internal state information for use in a subsequent invocation of this function. Repeated calls to this function are useful if your data is partitioned into a series of smaller vectors that you process within a loop, for example. This mode incurs an output delay of `tblen` symbols.

`y = mlseeq(x,chcffs,const,tblen,opmode,nsamp)` specifies the number of samples per symbol in `x`, that is, the oversampling factor.

The vector length of `x` must be a multiple of `nsamp`. When `nsamp>1`, the `chcffs` input represents the oversampled channel coefficients.

Preamble and Postamble in Reset Operation Mode

`y = mlseeq(...,'rst',nsamp,preamble,postamble)` specifies the preamble and postamble that you expect to precede and follow, respectively, the data in the input signal. The vectors `preamble` and `postamble` consist of integers between 0 and M-1, where M is the order of the modulation, that is, the number of elements in `const`. To omit a preamble or postamble, specify `[]`.

When the function applies the Viterbi algorithm, it initializes state metrics in a way that depends on whether you specify a preamble and/or postamble:

- If the preamble is nonempty, the function decodes the preamble and assigns a metric of 0 to the decoded state. If the preamble does not decode to a unique state (that is, if the length of the preamble is less than the channel memory), the decoder assigns a metric of 0 to all states that can be represented by the preamble. The traceback path ends at one of the states represented by the preamble.
- If the preamble is unspecified or empty, the decoder initializes the metrics of all states to 0.
- If the postamble is nonempty, the traceback path begins at the smallest of all possible decoded states that are represented by the postamble.
- If the postamble is unspecified or empty, the traceback path starts at the state with the smallest metric.

Additional Syntaxes in Continuous Operation Mode

`y = mlseeq(...,'cont',nsamp,...`
`init_metric,init_states,init_inputs)` causes the equalizer to start with its state metrics, traceback states, and traceback inputs specified by `init_metric`, `init_states`, and `init_inputs`, respectively. These three inputs are typically the extra outputs from a previous call to this

function, as in the syntax below. Each real number in `init_metric` represents the starting state metric of the corresponding state. `init_states` and `init_inputs` jointly specify the initial traceback memory of the equalizer. The table below shows the valid dimensions and values of the last three inputs, where `numStates` is M^{L-1}, M is the order of the modulation, and L is the number of symbols in the channel's impulse response (with no oversampling). To use default values for all of the last three arguments, specify them as `[],[],[]`.

Input Argument	Meaning	Matrix Size	Range of Values
`init_metric`	State metrics	1 row, `numStates` columns	Real numbers
`init_states`	Traceback states	`numStates` rows, `tblen` columns	Integers between 0 and `numStates-1`
`init_inputs`	Traceback inputs	`numStates` rows, `tblen` columns	Integers between 0 and M-1

`[y,final_metric,final_states,final_inputs] = ... mlseeq(...,'cont',...)` returns the normalized state metrics, traceback states, and traceback inputs, respectively, at the end of the traceback decoding process. `final_metric` is a vector with `numStates` elements that correspond to the final state metrics. `final_states` and `final_inputs` are both matrices of size `numStates`-by-`tblen`.

Examples

The example below illustrates how to use reset operation mode on an upsampled signal.

```
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 = mlseeq(msgRx,chanest,const,tblen,'rst',nsamp); % Equalize.
msgEqIdx = pamdemod(msgEq,M); % Demodulate.

[nerrs ber] = biterr(msgIdx, msgEqIdx) % Bit error rate
```

The output is below. Your results might vary because the example uses random numbers.

```
nerrs =

     1


ber =

    0.0010
```

The example in “Example: Continuous Operation Mode” on page 11-31 illustrates how to use the final state and initial state arguments when invoking `mlseeq` repeatedly.

The example in “Example: Using a Preamble” on page 11-34 illustrates how to use a preamble.

See Also

`equalize`, “Using MLSE Equalizers” on page 11-28

References

[1] Proakis, John G., *Digital Communications*, Fourth Edition, New York, McGraw-Hill, 2001.

[2] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, Wiley, 1996.

modnorm

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).

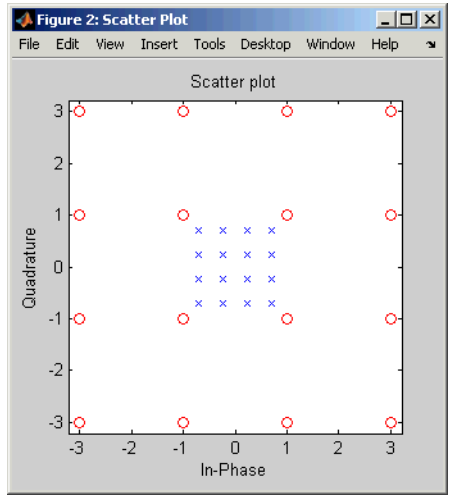
Examples

The code below illustrates how to use `modnorm` to transmit a quadrature amplitude modulated signal having a peak power of 1 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, whereas 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” on page 8-11.

See Also

`pammod`, `pamdemod`, `qammod`, `qamdemod`, Chapter 8, “Modulation”

mskdemod

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

`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, then the function treats the columns as independent channels and processes them independently.

`z = mskdemod(y,nsamp,dataenc)` specifies the method of encoding data for MSK. `dataenc` can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

`z = mskdemod(y,nsamp,dataenc,ini_phase)` specifies the initial phase of the demodulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of `dataenc`, set `dataenc` to `[]`.

`z = mskdemod(y,nsamp,dataenc,ini_phase,ini_state)` specifies the initial state of the demodulator. `ini_state` contains the last half symbol of the previously received signal. `ini_state` is an `nsamp`-by-C matrix, where C is the number of channels in `y`.

`[z,phaseout] = mskdemod(...)` returns the final phase of `y`, which is important for demodulating a future signal. The output `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values `0`, `pi/2`, `pi`, and `3*pi/2`.

`[z,phaseout,stateout] = mskdemod(...)` returns the final `nsamp` values of `y`, which is useful for demodulating the first symbol of a future signal. `stateout` has the same dimensions as the `ini_state` input.

Examples

The example below illustrates how to modulate and demodulate within a loop. To provide continuity from one iteration to the next, the syntaxes for `mskmod` and `mskdemod` use initial phases and/or state as both input and output arguments.

```
% Define parameters.
numbits = 99; % Number of bits per iteration
numchans = 2; % Number of channels (columns) in signal
nsamp = 16; % Number of samples per symbol

% Initialize.
numerrs = 0; % Number of bit errors seen so far
demod_ini_phase = zeros(1,numchans);  % Modulator phase
mod_ini_phase = zeros(1,numchans);    % Demodulator phase
ini_state = complex(zeros(nsamp,numchans)); % Demod. state

% Main loop
for iRuns = 1 : 10
    x = 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 below.

```
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`, Chapter 8, “Modulation”

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

`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, then the function treats the columns as independent channels and processes them independently.

`y = mskmod(x,nsamp,dataenc)` specifies the method of encoding data for MSK. `dataenc` can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

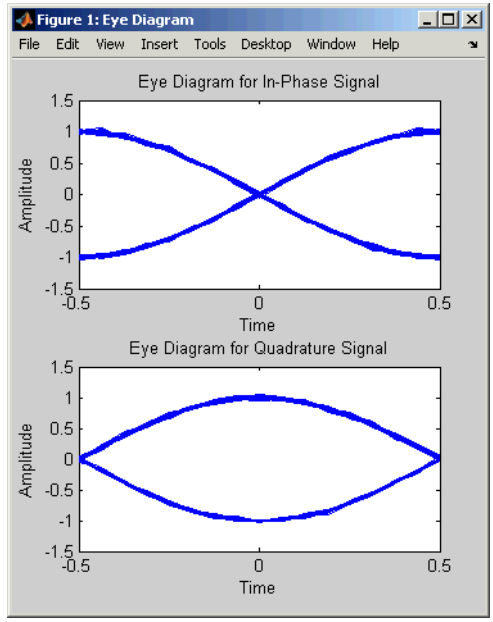
`y = mskmod(x,nsamp,dataenc,ini_phase)` specifies the initial phase of the MSK modulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of `dataenc`, set `dataenc` to `[]`.

`[y,phaseout] = mskmod(...)` returns the final phase of `y`. This is useful for maintaining phase continuity when you are modulating a future bit stream with differentially encoded MSK. `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values `0`, `pi/2`, `pi`, and `3*pi/2`.

Examples

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`, Chapter 8, "Modulation"

Purpose

Restore ordering of symbols using specified shift registers

Syntax

```
deintrlved = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay,init_state)
```

Description

`deintrlved = muxdeintrlv(data,delay)` restores the ordering of elements in `data` by using a set of internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, then the function processes the columns independently.

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

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

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `muxintrlv` function, use the same `delay` input in both functions. In that case, the two functions are inverses in the sense that applying `muxintrlv` followed by `muxdeintrlv` leaves data unchanged, after you take their combined delay of `length(delay)*max(delay)` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers” on page 7-9.

Examples

The example below illustrates how to use the state input and output when invoking `muxdeintrlv` repeatedly. Notice that `[deintrlved1; deintrlved2]` is the same as `deintrlved`.

```
delay = [0 4 8 12]; % Delays in shift registers
symbols = 100; % Number of symbols to process
% Interleave random data.
intrlved = muxintrlv(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” on page 7-6.

References

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

See Also

`muxintrlv`, Chapter 7, “Interleaving”

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, then 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” on page 7-6 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`, Chapter 7, “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

$$\frac{\mathrm{Fs}\sum_{i=1}^{N}|h(i)|^2}{\left|\sum_{i=1}^{N}h(i)\right|^2}$$

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” on page 11-8.

`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” on page 11-12.

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" on page 11-21.

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

$$(\texttt{LeakageFactor})w + \frac{(\texttt{StepSize})u^* e}{u^H u + \texttt{Bias}}$$

where the * operator denotes the complex conjugate and H denotes the Hermitian transpose.

See Also

`lms`, `signlms`, `varlms`, `rls`, `cma`, `lineareq`, `dfe`, `equalize`, Chapter 11, "Equalizers"

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Purpose

Convert octal numbers to decimal numbers

Syntax

```
d = oct2dec(c)
```

Description

`d = oct2dec(c)` converts an octal matrix `c` to a decimal matrix `d`, element by element. In both octal and decimal representations, the rightmost digit is the least significant.

Examples

The command below converts a 2-by-2 octal matrix.

```
d = oct2dec([12 144;0 25])

d =

    10   100
     0    21
```

For instance, the octal number 144 is equivalent to the decimal number 100 because 144 (octal) = $1*8^2 + 4*8^1 + 4*8^0 = 64 + 32 + 4 = 100$.

See Also

`bi2de`

Purpose

Offset quadrature phase shift keying demodulation

Syntax

```
z = oqpskdemod(y)
z = oqpskdemod(y,ini_phase)
```

Description

`z = oqpskdemod(y)` demodulates the complex envelope `y` of an OQPSK modulated signal. The function implicitly downsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `y` is a matrix with multiple rows, then 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`, Chapter 8, "Modulation"

oqpskmod

Purpose

Offset quadrature phase shift keying modulation

Syntax

```
y = oqpskmod(x)
y = oqpskmod(x,ini_phase)
```

Description

`y = oqpskmod(x)` outputs the complex envelope `y` of the modulation of the message signal `x` using offset quadrature phase shift keying (OQPSK) modulation. The message signal must consist of integers between 0 and 3. The function implicitly upsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `x` is a matrix with multiple rows, then 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`, Chapter 8, “Modulation”

Purpose

Pulse amplitude demodulation

Syntax

```
z = pamdemod(y,M)
z = pamdemod(y,M,ini_phase)
z = pamdemod(y,M,ini_phase,symbol_order)
```

Description

`z = pamdemod(y,M)` demodulates the complex envelope `y` of a pulse amplitude modulated signal. `M` is the alphabet size. The ideal modulated signal should have a minimum Euclidean distance of 2.

`z = pamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

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

Examples

The example in "Comparing Theoretical and Empirical Error Rates" on page 3-10 uses this function.

See Also

`pammod`, `qamdemod`, `qammod`, `pskdemod`, `pskmod`, Chapter 8, "Modulation"

pammod

Purpose

Pulse amplitude modulation

Syntax

```
y = pammod(x,M)
y = pammod(x,M,ini_phase)
y = pammod(x,M,ini_phase,symbol_order)
```

Description

`y = pammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using pulse amplitude modulation. `M` is the alphabet size. The message signal must consist of integers between 0 and `M-1`. The modulated signal has a minimum Euclidean distance of 2. If `x` is a matrix with multiple rows, then 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray constellation ordering.

Examples

The example in “Comparing Theoretical and Empirical Error Rates” on page 3-10 uses this function.

See Also

`pamdemod`, `qammod`, `qamdemod`, `pskmod`, `pskdemod`, Chapter 8, “Modulation”

Purpose

Plot channel characteristics with the channel visualization tool

Syntax

```
plot(h)
```

Description

`plot(h)`, where `h` is a channel object, will launch the channel visualization tool. This GUI tool will allow you to plot channel characteristics in various ways. See “Using the Channel Visualization Tool” on page 10-25 for details.

Examples

Examples using this plotting tool are found in “Examples of Using the Channel Visualization Tool” on page 10-35.

See Also

`filter`, `rayleighchan`, `ricianchan`

pmdemod

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` (Hz). `z` and the carrier signal have sampling rate `Fs` (Hz), 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" on page 8-5 uses `pmdemod`.

See Also

`pmmod`, `fmmod`, `fmdemod`, Chapter 8, "Modulation"

Purpose

Phase modulation

Syntax

```
y = pmmod(x,Fc,Fs,phasedev)
y = pmmod(x,Fc,Fs,phasedev,ini_phase)
```

Description

`y = pmmod(x,Fc,Fs,phasedev)` modulates the message signal `x` using phase modulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least `2*Fc`. The `phasedev` argument is the phase deviation of the modulated signal in radians.

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

Examples

The example in "Analog Modulation Example" on page 8-5 uses `pmmod`.

See Also

`pmdemod`, `fmmod`, `fmdemod`, Chapter 8, "Modulation"

poly2trellis

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 the Communications Blockset.

`trellis = poly2trellis(ConstraintLength,CodeGenerator)` performs the conversion for a rate k/n feedforward encoder. `ConstraintLength` is a 1-by-k vector that specifies the delay for the encoder's k input bit streams. `CodeGenerator` is a k-by-n matrix of octal numbers that specifies the n output connections for each of the encoder's k input bit streams.

`trellis = poly2trellis(ConstraintLength,CodeGenerator,... FeedbackConnection)` is the same as the syntax above, except that it applies to a feedback, not feedforward, encoder. `FeedbackConnection` is a 1-by-k vector of octal numbers that specifies the feedback connections for the encoder's k input bit streams.

For both syntaxes, the output is a MATLAB structure whose fields are as in the table below.

Fields of the Output Structure trellis for a Rate k/n Code

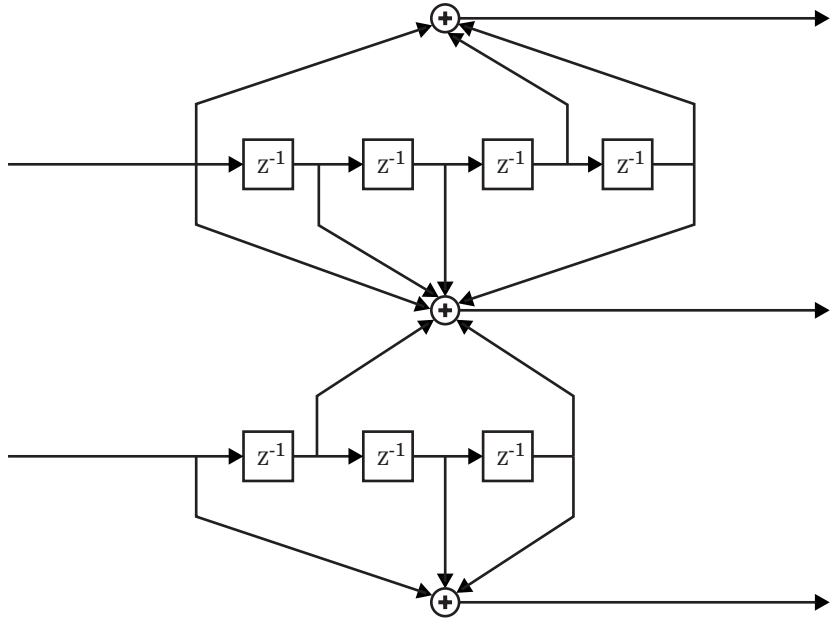
Field in trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

For more about this structure, see the reference page for the `istrellis` function.

Examples

An example of a rate 1/2 encoder is in “Polynomial Description of a Convolutional Encoder” on page 6-30.

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, then 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" on page 6-30

Purpose

Find primitive polynomials for a 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, then `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`, Chapter 12, “Galois Field Computations”

Purpose

Phase shift keying demodulation

Syntax

```
z = pskdemod(y,M)
z = pskdemod(y,M,ini_phase)
z = pskdemod(y,M,ini_phase,symbol_order)
```

Description

`z = pskdemod(y,M)` demodulates the complex envelope `y` of a PSK modulated signal. `M` is the alphabet size and must be an integer power of 2. The initial phase of the modulation is zero. If `y` is a matrix with multiple rows and columns, then 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray-coded ordering.

Examples

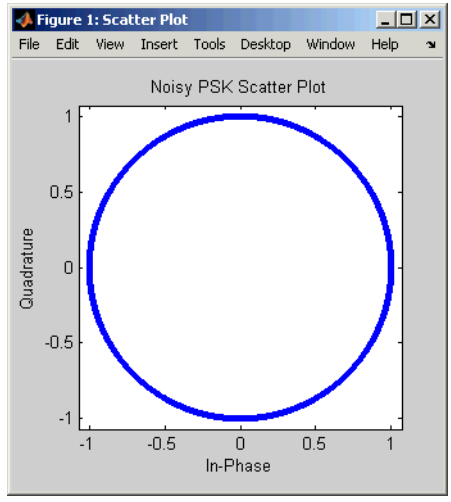
The example below compares PSK and PAM (phase amplitude modulation) to show that PSK is more sensitive to phase noise. This is the expected result because the PSK constellation is circular, while the PAM constellation is linear.

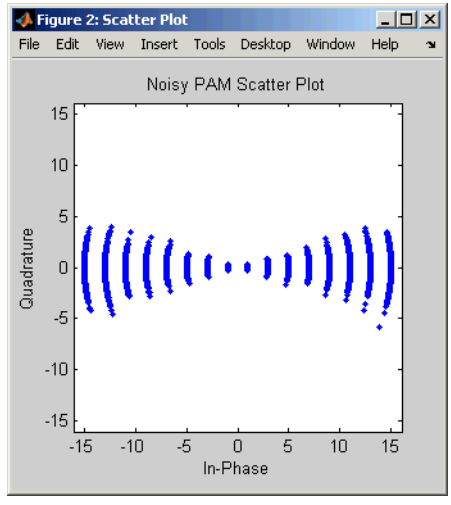
```
len = 10000; % Number of symbols
M = 16; % Size of alphabet
msg = randint(len,1,M); % Original signal

% 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 the example uses random numbers.

```
numerrs_psk =

   374


numerrs_pam =

     1
```





See Also

`pskmod`, `qamdemod`, `qammod`, `dpskmod`, `dpskdemod`, `modnorm`, Chapter 8, "Modulation"

pskmod

Purpose

Phase shift keying modulation

Syntax

```
y = pskmod(x,M)
y = pskmod(x,M,ini_phase)
y = pskmod(x,M,ini_phase,symbol_order)
```

Description

`y = pskmod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using phase shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The initial phase of the modulation is zero. If `x` is a matrix with multiple rows and columns, then 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray constellation ordering.

Examples

The examples in “Constellation for 16-PSK” on page 8-12 and on the reference page for `pskdemod` use this function.

See Also

`dpskmod`, `dpskdemod`, `pskdemod`, `pammod`, `pamdemod`, `qammod`, `qamdemod`, `modnorm`, Chapter 8, “Modulation”

Purpose

Quadrature amplitude demodulation

Syntax

```
z = qamdemod(y,M)
z = qamdemod(y,M,ini_phase)
z = qamdemod(y,M,ini_phase,symbol_order)
```

Description

`z = qamdemod(y,M)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. `M` is the alphabet size and must be an integer power of 2. The constellation is the same as in `qammod`. If `y` is a matrix with multiple rows, then the function processes the columns independently.

`z = qamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

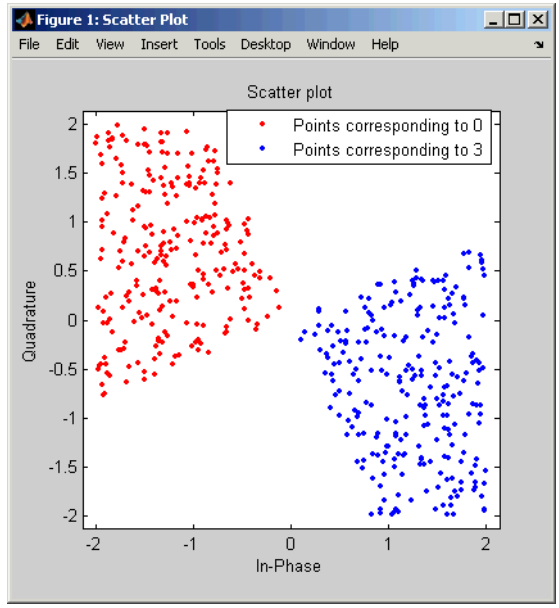
`z = qamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray-coded ordering.

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. You might also 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” on page 8-8.

See Also

`qammod`, `genqamdemod`, `genqammod`, `pamdemod`, `modnorm`, Chapter 8, “Modulation”

Purpose

Quadrature amplitude modulation

Syntax

```
y = qammod(x,M)
y = qammod(x,M,ini_phase)
y = qammod(x,M,ini_phase,symbol_order)
```

Description

`y = qammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The signal constellation is rectangular or cross-shaped, and the nearest pair of points in the constellation is separated by 2. If `x` is a matrix with multiple rows, then 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 will use a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it will use a Gray constellation ordering.

Examples

Examples using this function are in “Computing the Symbol Error Rate” on page 8-8 and “Examples of Signal Constellation Plots” on page 8-11.

See Also

`qamdemod`, `genqammod`, `genqamdemod`, `pammod`, `pamdemod`, `modnorm`, Chapter 8, “Modulation”

qfunc

Purpose

Q function

Syntax

```
y = qfunc(x)
```

Description

`y = qfunc(x)` is one minus the cumulative distribution function of the standardized normal random variable, evaluated at each element of the real array `x`. For a scalar `x`, the formula is

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-t^2/2)\,dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2}\operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Examples

The example below computes the Q function on a matrix, element by element.

```
x = [0 1 2; 3 4 5];
format short e % Switch to floating point format for displays.
y = qfunc(x)
format % Return to default format for displays.
```

The output is below.

```
y =

  5.0000e-001  1.5866e-001  2.2750e-002
  1.3499e-003  3.1671e-005  2.8665e-007
```

See Also

`qfuncinv`, `erf`, `erfc`, `erfcx`, `erfinv`, `erfcinv`

Purpose

Inverse Q function

Syntax

```
y = qfuncinv(x)
```

Description

`y = qfuncinv(x)` returns the argument of the Q function at which the Q function's value is `x`. The input `x` must be a real array with elements between 0 and 1, inclusive.

For a scalar `x`, the Q function is one minus the cumulative distribution function of the standardized normal random variable, evaluated at `x`. The Q function is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-t^2/2)dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2}\operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

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

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, then `index` is a vector whose kth entry is

- 0 if `sig`(k) ≤ `partition`(1)
- m if `partition`(m) < `sig`(k) ≤ `partition`(m+1)
- n if `partition`(n) < `sig`(k)

`[index,quants] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `codebook` prescribes a value for each partition in the quantization and `quants` contains the quantization of `sig` based on the quantization levels and prescribed values. `codebook` is a vector whose length exceeds the length of `partition` by one. `quants` is a row vector whose length is the same as the length of `sig`. `quants` is related to `codebook` and `index` by

```
quants(ii) = codebook(index(ii)+1);
```

where `ii` is an integer between 1 and `length(sig)`.

`[index,quants,distor] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `distor` estimates the mean square distortion of this quantization data set.

Examples

The command below rounds several numbers between 1 and 100 up to the nearest multiple of ten. `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” on page 5-2

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. The function is predictable for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, then 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.

Examples

For an example using random interleaving and deinterleaving, see "Example: Block Interleavers" on page 7-3.

See Also

`randintrlv`, Chapter 7, "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, then it is the number of nonzero entries in each row.
- If `errors` is a row vector, then it lists the possible number of nonzero entries in each row.
- If `errors` is a matrix having two rows, then 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`.

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`, Chapter 2, “Signal Sources”

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, then `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`.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed in the range from 0 to 7, you can use either of the following commands.

```
out = randint(10,10,[0,7]);
```

```
out = randint(10,10,8);
```

See Also

`rand`, `randsrc`, `randerr`, Chapter 2, “Signal Sources”

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. 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, then the function processes the columns independently.

Examples

For an example using random interleaving and deinterleaving, see "Example: Block Interleavers" on page 7-3.

See Also

`randdeintrlv`, Chapter 7, "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 one.

`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`.

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`, Chapter 2, “Signal Sources”

Purpose

Construct a 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.

`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 sets automatically. To learn how to view or change the values of a channel object, see "Viewing Object Properties" on page 10-9 or "Changing Object Properties" on page 10-10.

Writeable Properties

Property	Description
`InputSamplePeriod`	Sample period of the signal on which the channel acts, measured in seconds
`MaxDopplerShift`	Maximum Doppler shift of the channel, in Hz

Property	Description
`PathDelays`	Vector listing the delays of the discrete paths, in seconds
`AvgPathGaindB`	Vector listing the average gain of the discrete paths, in dB
`NormalizePathGains`	If `1`, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
`StoreHistory`	If `1`, channel state information 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

Property	Description	When MATLAB Sets or Updates Value
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

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. If you change the value of PathDelays, then MATLAB truncates or zero-pads the value of AvgPathGaindB if necessary to adjust its vector length. MATLAB might also change the values of read-only properties such as PathGains and ChannelFilterDelay.

Visualization of Channel

The characteristics of a channel can be plotted using the channel visualization tool. See for details.

Examples

Several examples using this function are in “Fading Channels” on page 10-6.

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. Notice that 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
         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
```

```
          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

This toolbox models a fading channel as a linear FIR filter, with tap weights given by

$$g_n = \sum_k \text{sinc}(\tau_k / T - n) h_k \text{ for } -N_1 \le n \le N_2$$

where

- The summation has one term for each major path.
- $\{\tau_k\}$ is the set of path delays.
- T is the input sample period.
- N_1 and N_2 are chosen so that $|g_n|$ is small when n is less than $-N_1$ or greater than N_2.
- N_1 is the value of the object's `ChannelFilterDelay` property.
- $\{h_k\}$ is the set of complex path gains, which are not correlated with each other.

To generate a particular path gain h_k, the function performs these steps:

1. Generates white Gaussian noise
2. Passes the noise through a filter whose power spectrum corresponds to the Jakes Doppler spectrum

3 Interpolates values so that the sample period is consistent with that of the signal

4 Adjusts accordingly to obtain the correct average path gain

See Also

`ricianchan`, `filter`, `plot (channel)`, `reset`, "Fading Channels" on page 10-6

References

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

Purpose

Design a raised cosine 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`	`[-3,3]`
`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))*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 length-two vector that indicates the number of symbol periods before and after the peak response. `rate` is the number of points in each input symbol period of length `T`. `rate` must be greater than 1. The input sample rate is `T` samples per second, while the output sample rate is `T*rate` samples per second.

The order of the FIR filter is

```
(n_T(2)-n_T(1))*rate
```

The arguments `n_T`, `rate`, and `T` are optional inputs whose default values are [-3,3], 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' then this syntax is the same as the previous one.

The impulse response of a square root raised cosine filter is

$$h(t) = 4R\frac{\cos((1+R)\pi t/T) + \dfrac{\sin((1-R)\pi t/T)}{4R\dfrac{t}{T}}}{\pi\sqrt{T}(1-(4Rt/T)^2)}$$

`rcosfir(...)` produces plots of the time and frequency responses of the raised cosine filter.

`rcosfir(...,colr)` uses the string `colr` to determine the plotting color. The choices for `colr` are the same as those listed for the `plot` function.

`[b,sample_time] = rcosfir(...)` returns the FIR filter and its sample time.

Examples

The commands below compare different rolloff factors.

```
rcosfir(0);
subplot(211); hold on;
subplot(212); hold on;
rcosfir(.5,[],[],[],[],'r-');
rcosfir(1,[],[],[],[],'g-');
```

See Also

`rcosiir`, `rcosflt`, `rcosine`, `firrcos`, `rcosdemo`, Chapter 9, "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,'filter_type',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
`filter_type`	`fir/normal`
`r`	`0.5`
`delay`	`3`
`tol`	`0.01`
`den`	`1`

Description

The function `rcosflt` passes an input signal through a raised cosine filter. You can either let `rcosflt` design a raised cosine filter automatically or you can specify the raised cosine filter yourself using input arguments.

Designing the Filter Automatically

`y = rcosflt(x,Fd,Fs)` designs a raised cosine FIR filter and then filters the input signal `x` using it. The sample frequency for the digital input signal `x` is `Fd`, and the sample frequency for the output signal `y` is `Fs`. The ratio `Fs/Fd` must be an integer. In the course of filtering, `rcosflt` upsamples the data by a factor of `Fs/Fd`, by inserting zeros between samples. The order of the filter is `1+2*delay*Fs/Fd`, where `delay` is 3 by default. If `x` is a vector, then the sizes of `x` and `y` are related by this equation.

```
length(y) = (length(x) + 2 * delay)*Fs/Fd
```

Otherwise, `y` is a matrix, each of whose columns is the result of filtering the corresponding column of `x`.

`y = rcosflt(x,Fd,Fs,'filter_type',r,delay,tol)` designs a raised cosine FIR or IIR filter and then filters the input signal `x` using it. The ratio `Fs/Fd` must be an integer. `r` is the rolloff factor for the filter, a real number in the range [0, 1]. `delay` is the filter's group delay, measured in input samples. The actual group delay in the filter design is `delay/Fd` seconds. The input `tol` is the tolerance in the IIR filter design. FIR filter design does not use `tol`.

The characteristics of `x`, `Fd`, `Fs`, and `y` are as in the first syntax.

The fourth input argument, '`filter_type`', is a string that determines the type of filter that `rcosflt` should design. Use one of the values in the table below.

Values of filter_type to Determine the Type of Filter

Type of Filter	Value of filter_type
FIR raised cosine filter	`fir` or `fir/normal`
IIR raised cosine filter	`iir` or `iir/normal`
Square-root FIR raised cosine filter	`fir/sqrt`
Square-root IIR raised cosine filter	`iir/sqrt`

`y = rcosflt(x,Fd,Fs,'filter_type/Fs',r,delay,tol)` is the same as the previous syntax, except that it assumes that `x` has sample frequency `Fs`. This syntax does not upsample `x` any further. If `x` is a vector, then the relative sizes of `x` and `y` are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

As before, if x is a nonvector matrix, then 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 `filter_type` includes `fir`, then omit `den`. This syntax uses the same arguments x, Fd, Fs, and `filter_type` 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, then 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, then the relative sizes of x and y are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

Additional Output

`[y,t] = rcosflt(...)` outputs t, a vector that contains the sampling time points of y.

See Also

rcosine, rcosfir, rcosiir, rcosdemo, Chapter 9, “Special Filters”

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose

Design a raised cosine IIR filter

Syntax

```
[num,den] = rcosiir(R,T_delay,rate,T,tol)
[num,den] = rcosiir(R,T_delay,rate,T,tol,filter_type)
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, then it becomes the order of the IIR filter; if `tol` is less than 1,

then 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,filter_type)` designs a square-root raised cosine filter if `filter_type` is `'sqrt'`. If `filter_type` is 'normal' then 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`, Chapter 9, "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 a 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 `Fd`. The sampling frequency for the filter is `Fs`. The ratio `Fs/Fd` must be a positive integer greater than 1. The default rolloff factor is .5. The filter's group delay, which is the time between the input to the filter and the filter's peak response, is three input samples. Equivalently, the group delay is `3/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag)` designs a raised cosine filter using directions in the string variable `type_flag`. Filter types are listed in the table below, along with the corresponding values of `type_flag`.

Types of Filter and Corresponding Values of type_flag

Type of Filter	Value of type_flag
Finite impulse response (FIR)	`'default'` or `'fir/normal'`
Infinite impulse response (IIR)	`'iir'` or `''iir/normal''`
Square-root raised cosine FIR	`'sqrt'` or `'fir/sqrt'`
Square-root raised cosine IIR	`'iir/sqrt'`

The default tolerance value in IIR filter design is 0.01.

`[num,den] = rcosine(Fd,Fs,type_flag,r)` specifies the rolloff factor, `r`. The rolloff factor is a real number in the range [0, 1].

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay)` specifies the filter's group delay, measured in input samples. `delay` is a positive integer. The actual group delay in the filter design is `delay/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol)` specifies the tolerance in the IIR filter design. FIR filter design does not use `tol`.

See Also

`rcosflt`, `rcosiir`, `rcosfir`, `rcosdemo`, Chapter 9, "Special Filters"

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

rectpulse

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, then 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" on page 8-10 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.

```
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`

reset (channel)

Purpose

Reset channel object

Syntax

```
reset(chan)
reset(chan,randstate)
```

Description

`reset(chan)` resets the channel object `chan`, initializing the `PathGains` and `NumSamplesProcessed` properties as well as internal filter states. This syntax is useful when you want the effect of creating a new channel.

`reset(chan,randstate)` resets the channel object `chan` and initializes the state of the random number generator that the channel uses. `randstate` is a two-element column vector. This syntax is useful when you want to repeat previous numerical results that started from a particular state.

Example

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.

```
% 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))];
```

```
isequal(y1,y2) % y1 and y2 should be the same.
```

The output is below.

```
ans =

     1
```

See Also

`rayleighchan`, `ricianchan`, `filter`, "Fading Channels" on page 10-6

reset (equalizer)

Purpose

Reset equalizer object

Syntax

```
reset(eqobj)
```

Description

`reset(eqobj)` resets the equalizer object `eqobj`, initializing the `Weights`, `WeightInputs`, and `NumSamplesProcessed` properties as well as adaptive algorithm states. If `eqobj` is a CMA equalizer, then `reset` does not change the `Weights` property.

See Also

`dfe`, `equalize`, `lineareq`, Chapter 11, “Equalizers”

Purpose

Construct a Rician fading channel object

Syntax

```
chan = ricianchan(ts,fd,k)
chan = ricianchan(ts,fd,k,tau,pdb)
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 this channel, the specular component has zero phase and the phase does not change with the Doppler shift. You can model the effect of the channel on a signal `x` by using the syntax `y = filter(chan,x)`.

`chan = ricianchan(ts,fd,k,tau,pdb)` constructs a frequency-selective ("multiple path") fading channel object that models the first discrete path as a Rician fading process and each of the remaining discrete paths 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.

`chan = ricianchan` constructs a frequency-flat channel object with no Doppler shift and a K-factor of 1. 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 sets automatically. To learn how to view or change the values of a channel object, see "Viewing Object Properties" on page 10-9 or "Changing Object Properties" on page 10-10.

Writeable Properties

Property	Description
`InputSamplePeriod`	Sample period of the signal on which the channel acts, measured in seconds
`MaxDopplerShift`	Maximum Doppler shift of the channel, in Hz
`KFactor`	Rician K-factor (scalar) for first path
`PathDelays`	Vector listing the delays of the discrete paths, in seconds
`AvgPathGaindB`	Vector listing the average gain of the discrete paths, in dB
`NormalizePathGains`	If `1`, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
`StoreHistory`	If `1`, channel state information 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, `'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

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. If you change the value of one of these properties, then MATLAB truncates or zero-pads the value of the other property if necessary to adjust its vector length.

If you change the value of `PathDelays` or `AvgPathGaindB`, MATLAB might also change the values of read-only properties such as `PathGains` and `ChannelFilterDelay`.

Channel Visualization

The characteristics of a channel can be plotted using the channel visualization tool. See for details.

Examples

The example in "Quasi-Static Channel Modeling" on page 10-20 uses this function.

Algorithm

This function produces a model for a Rayleigh channel, adds a constant to the first path gain, and then normalizes to correct the set of average path gains. To learn about the algorithm for producing a Rayleigh channel model, see "Algorithm" on page 15-290 on the `rayleighchan` reference page.

See Also

`rayleighchan`, `filter`, `plot (channel)`, `reset`, "Fading Channels" on page 10-6

References

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

Purpose

Construct a 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" on page 11-8.

`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" on page 11-12.

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" on page 11-13 and "Example: Adaptive Equalization Within a Loop" on page 11-23.

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of inputs, u, and the current inverse correlation matrix, P, this adaptive algorithm first computes the Kalman gain vector, K:

$$K = \frac{Pu}{(\texttt{ForgetFactor}) + u^H Pu}$$

where H denotes the Hermitian transpose.

Then the new inverse correlation matrix is given by

$(\texttt{ForgetFactor})^{-1}(P - Ku^H P)$

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`, Chapter 11, "Equalizers"

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Purpose

Reed-Solomon decoder

Syntax

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

Description

`decoded = rsdec(code,n,k)` attempts to decode the received signal in `code` using an `[n,k]` Reed-Solomon decoding process with the narrow-sense generator polynomial. `code` is a Galois array of symbols having m bits each. Each `n`-element row of `code` represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, then `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'`, then 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`, then 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" on page 6-7.

Algorithm

`rsdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in "References" on page 15-316 below.

Limitations

`n` and `k` must differ by an even integer. `n` must be between 3 and 65535.

See Also

`rsenc`, `gf`, `rsgenpoly`, "Block Coding" on page 6-2

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

Decode ASCII file that was encoded using Reed-Solomon code

Syntax

```
rsdecof(file_in,file_out); rsdecof(file_in,file_out,err_cor);
```

Description

This function is the inverse process of the function `rsencof` in that it decodes a file that `rsencof` encoded.

`rsdecof(file_in,file_out)` decodes the ASCII file `file_in` that was previously created by the function `rsencof` using an error-correction capability of 5. The decoded message is written to `file_out`. Both `file_in` and `file_out` are string variables.

Note If the number of characters in `file_in` is not an integer multiple of 127, then the function appends `char(4)` symbols to the data it must decode. If you encode and then decode a file using `rsencof` and `rsdecof`, respectively, then the decoded file might have `char(4)` symbols at the end that the original file does not have.

`rsdecof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 *`err_cor`. The value in `err_cor` must match the value used in `rsencof` when `file_in` was created.

Examples

An example is on the reference page for `rsencof`.

See Also

`rsencof`, “Block Coding” on page 6-2

Purpose

Reed-Solomon encoder

Syntax

```
code = rsenc(msg,n,k)
code = rsenc(msg,n,k,genpoly)
code = rsenc(...,paritypos)
```

Description

`code = rsenc(msg,n,k)` encodes the message in `msg` using an [`n`,`k`] Reed-Solomon code with the narrow-sense generator polynomial. `msg` is a Galois array of symbols having m bits each. Each `k`-element row of `msg` represents a message word, where the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, then `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” on page 6-5 and “Creating and Decoding Reed-Solomon Codes” on page 6-7.

Limitations

`n` and `k` must differ by an even integer. `n` must be between 3 and 65535.

See Also

`rsdec`, `gf`, `rsgenpoly`, “Block Coding” on page 6-2

Purpose

Encode ASCII file using Reed-Solomon code

Syntax

```
rsencof(file_in,file_out); rsencof(file_in,file_out,err_cor);
```

Description

`rsencof(file_in,file_out)` encodes the ASCII file `file_in` using (127, 117) Reed-Solomon code. The error-correction capability of this code is 5 for each block of 127 codeword characters. This function writes the encoded text to the file `file_out`. Both `file_in` and `file_out` are string variables.

`rsencof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 * `err_cor`.

Note If the number of characters in `file_in` is not an integer multiple of 127 - 2 * `err_cor`, then the function appends `char(4)` symbols to `file_out`.

Examples

The file `matlabroot/toolbox/comm/comm/oct2dec.m` contains text help for the `oct2dec` function in this toolbox. The commands below encode the file using `rsencof` and then decode it using `rsdecof`.

```
file_in = [matlabroot '/toolbox/comm/comm/oct2dec.m'];
file_out = 'encodedfile'; % Or use another filename
rsencof(file_in,file_out) % Encode the file.

file_in = file_out;
file_out = 'decodedfile'; % Or use another filename
rsdecof(file_in,file_out) % Decode the file.
```

To see the original file and the decoded file in the MATLAB workspace, use the commands below (or similar ones if you modified the filenames above).

```
type oct2dec.m
type decodedfile
```

See Also

`rsdecof`, “Block Coding” on page 6-2

rsgenpoly

Purpose

Generator polynomial of Reed-Solomon code

Syntax

```
genpoly = rsgenpoly(n,k)
genpoly = rsgenpoly(n,k,prim_poly)
genpoly = rsgenpoly(n,k,prim_poly,b)
[genpoly,t] = rsgenpoly(...)
```

Description

`genpoly = rsgenpoly(n,k)` returns the narrow-sense generator polynomial of a Reed-Solomon code with codeword length `n` and message length `k`. The codeword length `n` must have the form 2^m-1 for some integer m, 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 $(X - A^1)(X - A^2)...(X - A^{2t})$ 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 $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$ 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(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1    13    12     8     7
```

For additional examples, see “Parameters for Reed-Solomon Codes” on page 6-5.

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” on page 6-2

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, then `scatterplot` interprets the first column as in-phase components and the second column as quadrature components.
- If `x` is a complex vector, then `scatterplot` interprets the real part as in-phase components and the imaginary part as quadrature components.
- If x is a real vector, then `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" on page 3-22 or the example on the reference page for `qamdemod`. Both examples illustrate how to plot multiple signals in a single scatter plot.

For an online demonstration, type `playshow scattereyedemo`.

See Also

`eyediagram`, `plot`, `scattereyedemo`, `scatter`, "Scatter Plots" on page 3-22

Purpose

Calculate bit error rate using semianalytic technique

Syntax

```
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,EbNo)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den,EbNo)
[ber,avgampl,avgpower] = semianalytic(...)
```

Graphical Interface

As an alternative to the `semianalytic` function, invoke the BERTool GUI (`bertool`) and use the **Semianalytic** panel.

Description

`ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)` returns the bit error rate (BER) of a system that transmits the complex baseband vector signal `txsig` and receives the noiseless complex baseband vector signal `rxsig`. Each of these signals has `Nsamp` samples per symbol. `Nsamp` is also the sampling rate of `txsig` and `rxsig`, in Hz. The function assumes that `rxsig` is the input to the receiver filter, and the function filters `rxsig` with an ideal integrator. `modtype` is the modulation type of the signal and `M` is the alphabet size. The table below lists the valid values for `modtype` and `M`.

Modulation Scheme	Value of modtype	Valid Values of M
Differential phase shift keying (DPSK)	`'dpsk'`	2, 4
Minimum shift keying (MSK) with differential encoding	`'msk/diff'`	2
Minimum shift keying (MSK) with nondifferential encoding	`'msk/nondiff'`	2

Modulation Scheme	Value of modtype	Valid Values of M
Phase shift keying (PSK) with differential encoding, where the phase offset of the constellation is 0	`'psk/diff'`	2, 4
Phase shift keying (PSK) with nondifferential encoding, where the phase offset of the constellation is 0	`'psk/nondiff'`	2, 4, 8, 16, 32, or 64
Offset quaternary phase shift keying (OQPSK)	`'oqpsk'`	4
Quadrature amplitude modulation (QAM)	`'qam'`	4, 8, 16, 32, 64, 128, 256, 512, 1024

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, then 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 semi-analytic technique is in "Procedure for the Semianalytic Technique" on page 3-6. Sample code is in "Example: Using the Semianalytic Technique" on page 3-7.

Limitations

The function makes several important assumptions about the communication system. See "When to Use the Semianalytic Technique"

on page 3-5 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" on page 3-5

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[2] Pasupathy, Subbarayan, "Minimum Shift Keying: A Spectrally Efficient Modulation," *IEEE Communications Magazine*, July, 1979, pp. 14-22.

Purpose

Convert shift to mask vector for a 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.

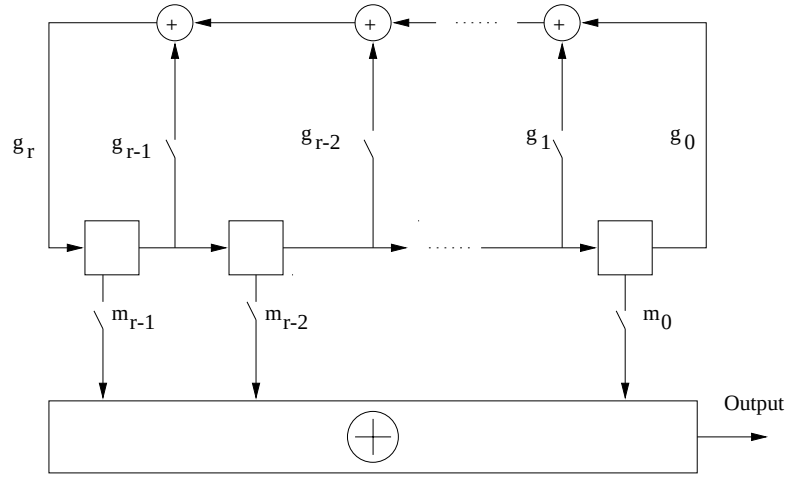
Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, then the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

Definition of Equivalent Mask

The equivalent mask for the shift s is the remainder after dividing the polynomial x^s by the primitive polynomial. The vector `mask` represents the remainder polynomial by listing the coefficients in order of descending powers.

Shifts, Masks, and Pseudonoise Sequence Generators

Linear feedback shift registers are part of an implementation of a pseudonoise sequence generator. Below is a schematic diagram of a pseudonoise sequence generator. All adders perform addition modulo 2.



The primitive polynomial determines the state of each switch labeled g_k, while 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, then 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 the Communications Blockset 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.

Purpose

Construct a signed least mean square (LMS) adaptive algorithm object

Syntax

```
alg = signlms(stepsize)
alg = lms(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" on page 11-8.

`alg = signlms(stepsize)` constructs an adaptive algorithm object based on the signed least mean square (LMS) algorithm with a step size of `stepsize`.

`alg = lms(stepsize,algtype)` constructs an adaptive algorithm object of type `algtype` from the family of signed LMS algorithms. The table below lists the possible values of `algtype`.

Value of `algtype`	**Type of Signed LMS Algorithm**
`'Sign LMS'`	Sign LMS (default)
`'Signed Regressor LMS'`	Signed regressor LMS
`'Sign Sign LMS'`	Sign-sign LMS

Properties

The table below describes the properties of the signed LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see "Accessing Properties of an Adaptive Algorithm" on page 11-12.

Property	Description
`AlgType`	The type of signed LMS algorithm, corresponding to the `algtype` input argument. You cannot change the value of this property after creating the object.
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

- (`LeakageFactor`) w + (`StepSize`) u^*sgn(Re(e)), for sign LMS
- (`LeakageFactor`) w + (`StepSize`) sgn(Re(u)) Re(e), for signed regressor LMS
- (`LeakageFactor`) w + (`StepSize`) sgn(Re(u)) sgn(Re(e)), for sign-sign LMS

where the * operator denotes the complex conjugate and sgn denotes the signum function (`sign` in MATLAB).

See Also

`lms`, `normlms`, `varlms`, `rls`, `cma`, `lineareq`, `dfe`, `equalize`, Chapter 11, "Equalizers"

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

Purpose

Single sideband amplitude demodulation

Syntax

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

Description

For All Syntaxes

`z = ssbdemod(y,Fc,Fs)` demodulates the single sideband amplitude modulated signal `y` from the carrier signal having frequency `Fc` (Hz). The carrier signal and `y` have sampling rate `Fs` (Hz). The modulated signal has zero initial phase, and can be an upper- or lower-sideband signal. The demodulation process uses the lowpass filter specified by `[num,den] = butter(5,Fc*2/Fs)`.

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

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

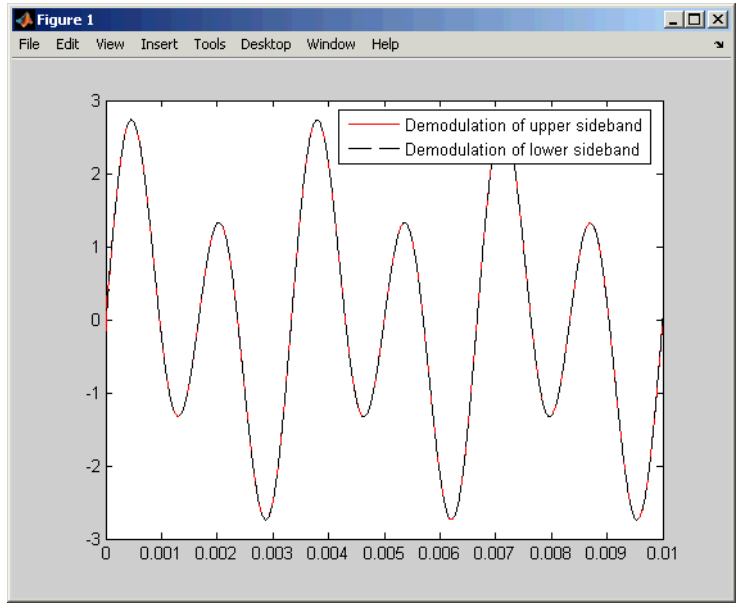
`z = ssbdemod(y,Fc,Fs,ini_phase,num,den)` specifies the numerator and denominator of the lowpass filter used in the demodulation.

Examples

The code below shows that `ssbdemod` can demodulate an upper-sideband or lower-sideband signal.

```
Fc = 12000; Fs = 270000;
t = [0:1/Fs:0.01]';
s = sin(2*pi*300*t)+2*sin(2*pi*600*t);
y1 = ssbmod(s,Fc,Fs,0); % Lower-sideband modulated signal
y2 = ssbmod(s,Fc,Fs,0,'upper'); % Upper-sideband modulated signal
s1 = ssbdemod(y1,Fc,Fs); % Demodulate lower sideband
s2 = ssbdemod(y2,Fc,Fs); % Demodulate upper sideband
% Plot results to show that the curves overlap.
figure; plot(t,s1,'r-',t,s2,'k--');
```

```
legend('Demodulation of upper sideband','Demodulation of lower sideband')
```



See Also

ssbmod, amdemod, Chapter 8, “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`, Chapter 8, “Modulation”

Purpose

Compute number of symbol errors and symbol error rate

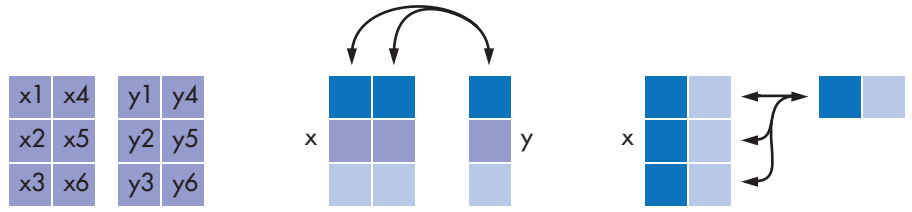
Syntax

```
[number,ratio] = symerr(x,y)
[number,ratio] = symerr(x,y,flg)
[number,ratio,loc] = symerr(...)
```

Description

For All Syntaxes

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



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

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

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

The output `number` is a scalar or vector that indicates the number of elements that differ. The size of `number` is determined by the optional input `flg` and by the dimensions of `x` and `y`. The output `ratio` equals `number` divided by the total number of elements in the *smaller* input.

For Specific Syntaxes

`[number,ratio] = symerr(x,y)` compares the elements in `x` and `y`. The sizes of `x` and `y` determine which elements are compared:

- If `x` and `y` are matrices of the same dimensions, then `symerr` compares `x` and `y` element by element. `number` is a scalar. See schematic (a) in the figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then `symerr` compares the vector element by element with *each row (resp., column)* of the matrix. The length

of the vector must equal the number of columns (resp., rows) in the matrix. `number` is a column (resp., row) vector whose mth entry indicates the number of elements that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

`[number,ratio] = symerr(x,y,flg)` is similar to the previous syntax, except that `flg` can override the defaults that govern which elements `symerr` compares and how `symerr` computes the outputs. The values of `flg` are '`overall`', '`column-wise`', and '`row-wise`'. The table below describes the differences that result from various combinations of inputs. In all cases, `ratio` is `number` divided by the total number of elements in `y`.

Comparing a Two-Dimensional Matrix x with Another Input y

Shape of y	flg	Type of Comparison	number
Two-dim. matrix	`'overall'` (default)	Element by element	Total number of symbol errors
	`'column-wise'`	mth column of `x` vs. mth column of `y`	Row vector whose entries count symbol errors in each column
	`'row-wise'`	mth row of `x` vs. mth row of `y`	Column vector whose entries count symbol errors in each row

Shape of y	flg	Type of Comparison	number
Column vector	`'overall'`	y vs. each column of x	Total number of symbol errors
	`'column-wise'` (default)	y vs. each column of x	Row vector whose entries count symbol errors in each column of x
Row vector	`'overall'`	y vs. each row of x	Total number of symbol errors
	`'row-wise'` (default)	y vs. each row of x	Column vector whose entries count symbol errors in each row of x

`[number,ratio,loc] = symerr(...)` returns a binary matrix `loc` that indicates which elements of `x` and `y` differ. An element of `loc` is zero if the corresponding comparison yields no discrepancy, and one otherwise.

Examples

On the reference page for `biterr`, the last example uses `symerr`.

The command below illustrates how `symerr` works when one argument is a vector and the other is a matrix. It compares the vector `[1,2,3]'` to the columns

$$\begin{bmatrix}1\\3\\3\end{bmatrix}, \begin{bmatrix}1\\2\\3\end{bmatrix}, \begin{bmatrix}3\\2\\8\end{bmatrix}, \text{ and } \begin{bmatrix}1\\2\\3\end{bmatrix}$$

of the matrix.

```
num = symerr([1 2 3]',[1 1 3 1;3 2 2 2; 3 3 8 3])
```

```
num =

     1     0     2     0
```

As another example, the command below illustrates the use of `flg` to override the default row-by-row comparison. Notice that `number` and `ratio` are scalars.

```
format rat;
[number,ratio,loc] = symerr([1 2; 3 4],[1 3],'overall')
```

The output is below.

```
number =

      3


ratio =

     3/4


loc =

      0            1
      1            1
```

See Also

`biterr`, "Performance Results via Simulation" on page 3-2

syndtable

Purpose

Produce syndrome decoding table

Syntax

```
t = syndtable(h)
```

Description

`t = syndtable(h)` returns a decoding table for an error-correcting binary code having codeword length n and message length k. `h` is an (n-k)-by-n parity-check matrix for the code. `t` is a 2^{n-k}-by-n binary matrix. The rth row of `t` is an error pattern for a received binary codeword whose syndrome has decimal integer value r-1. (The syndrome of a received codeword is its product with the transpose of the parity-check matrix.) In other words, the rows of `t` represent the coset leaders from the code's standard array.

When converting between binary and decimal values, the leftmost column is interpreted as the *most* significant digit. This differs from the default convention in the `bi2de` and `de2bi` commands.

Examples

An example is in "Decoding Table" on page 6-22.

See Also

`decode`, `hammgen`, `gfcosets`, "Block Coding" on page 6-2

References

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

Purpose

Construct variable-step-size least mean square (LMS) adaptive algorithm object

Syntax

```
alg = varlms(initstep,incstep,minstep,maxstep)
```

Description

The `varlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Using Adaptive Equalizer Functions and Objects" on page 11-8.

`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" on page 11-12.

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.

Property	Description
`InitStep`	Initial value of step size when the algorithm starts
`IncStep`	Increment by which the step size changes from iteration to iteration
`MinStep`	Minimum value of step size
`MaxStep`	Maximum value of step size

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` or `dfe` function), the equalizer object has a `StepSize` property. The property value is a vector that lists the current step size for each weight in the equalizer.

Examples

For an example that uses this function, see "Linked Properties of an Equalizer Object" on page 11-14.

Algorithm

Referring to the schematics presented in "Overview of Adaptive Equalizer Classes" on page 11-3, define w as the vector of all current weights w_i and define u as the vector of all inputs u_i. Based on the current step size, μ, this adaptive algorithm first computes the quantity

$$\mu_0 = \mu + (\texttt{IncStep})\ \mathrm{Re}(gg_{prev})$$

where $g = ue^*$, g_{prev} is the analogous expression from the previous iteration, and the * operator denotes the complex conjugate.

Then the new step size is given by

- μ_0, if it is between `MinStep` and `MaxStep`
- `MinStep`, if $\mu_0 <$ `MinStep`
- `MaxStep`, if $\mu_0 >$ `MaxStep`

The new set of weights is given by

(LeakageFactor) $w + 2 \mu g^*$

See Also

lms, signlms, normlms, rls, cma, lineareq, dfe, equalize, Chapter 11, “Equalizers”

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

vec2mat

Purpose

Convert vector into matrix

Syntax

```
mat = vec2mat(vec,matcol)
mat = vec2mat(vec,matcol,padding)
[mat,padded] = vec2mat(...)
```

Description

`mat = vec2mat(vec,matcol)` converts the vector `vec` into a matrix with `matcol` columns, creating one row at a time. If the length of `vec` is not a multiple of `matcol`, then extra zeros are placed in the last row of `mat`. The matrix `mat` has `ceil(length(vec)/matcol)` rows.

`mat = vec2mat(vec,matcol,padding)` is the same as the first syntax, except that the extra entries placed in the last row of `mat` are not necessarily zeros. The extra entries are taken from the matrix `padding`, in order. If `padding` has fewer entries than are needed, then the last entry is used repeatedly.

`[mat,padded] = vec2mat(...)` returns an integer `padded` that indicates how many extra entries were placed in the last row of `mat`.

Note `vec2mat` is similar to the built-in MATLAB function `reshape`. However, given a vector input, `reshape` creates a matrix one *column* at a time instead of one row at a time. Also, `reshape` requires the input and output matrices to have the same number of entries, whereas `vec2mat` places extra entries in the output matrix if necessary.

Examples

```
vec = [1 2 3 4 5];
[mat,padded] = vec2mat(vec,3)
[mat2,padded2] = vec2mat(vec,4)
mat3 = vec2mat(vec,4,[10 9 8; 7 6 5; 4 3 2])
```

The output is below.

```
mat =

     1     2     3
     4     5     0


padded =

     1


mat2 =

     1     2     3     4
     5     0     0     0


padded2 =

     3


mat3 =

     1     2     3     4
     5    10     7     4
```

See Also

reshape

Purpose

Convolutionally decode binary data using Viterbi algorithm

Syntax

```
decoded = vitdec(code,trellis,tblen,opmode,dectype)
decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec)
decoded = ...
  vitdec(...,'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" on page 6-34 and the reference page for the `istrellis` function. `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits. Each symbol in the vector `decoded` consists of `log2(trellis.numInputSymbols)` bits. `tblen` is a positive integer scalar that specifies the traceback depth. If the code rate is 1/2, then a typical value for `tblen` is about five times the constraint length of the code.

The string `opmode` indicates the decoder's operation mode and its assumptions about the corresponding encoder's operation. Choices are in the table below.

Values of opmode Input

Value	Meaning
`'cont'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. A delay equal to `tblen` symbols elapses before the first decoded symbol appears in the output. This mode is appropriate when you invoke this function repeatedly and want to preserve continuity between successive invocations. See the continuous operation mode syntaxes below.

Value	Meaning
`'term'`	The encoder is assumed to have both started and ended at the all-zeros state, which is true for the default syntax of the `convenc` function. The decoder traces back from the all-zeros state. This mode incurs no delay. This mode is appropriate when the uncoded message (that is, the input to `convenc`) has enough zeros at the end to fill all memory registers of the encoder. If the encoder has `k` input streams and constraint length vector `constr` (using the polynomial description of the encoder), then "enough" means `k*max(constr-1)`.
`'trunc'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. This mode incurs no delay. This mode is appropriate when you cannot assume the encoder ended at the all-zeros state and when you do not want to preserve continuity between successive invocations of this function.

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.

Value	Meaning
`'hard'`	`code` contains binary input values.
`'soft'`	For soft-decision decoding, use the syntax below. Note that `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.

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, then 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. Notice that 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`. Notice also that the bit error rate computations must account for the delay that the continuous operation mode incurs.

```
trel = poly2trellis(3,[6 7]); % Define trellis.
msg = randint(100,1,2,123); % Random data
code = convenc(msg,trel); % Encode.
ncode = rem(code + randerr(200,1,[0 1;.95 .05]),2); % Add noise.
tblen = 3; % Traceback length
decoded1 = vitdec(ncode,trel,tblen,'cont','hard'); %Hard decision
% Use unquantized decisions.
ucode = 1-2*ncode; % +1 & -1 represent zero & one, respectively.
decoded2 = vitdec(ucode,trel,tblen,'cont','unquant');
% To prepare for soft-decision decoding, map to decision values.
[x,qcode] = quantiz(1-2*ncode,[-.75 -.5 -.25 0 .25 .5 .75],...
[7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
decoded3 = vitdec(qcode',trel,tblen,'cont','soft',3);
% Compute bit error rates, using the fact that the decoder
% output is delayed by tblen symbols.
[n1,r1] = biterr(decoded1(tblen+1:end),msg(1:end-tblen));
[n2,r2] = biterr(decoded2(tblen+1:end),msg(1:end-tblen));
[n3,r3] = biterr(decoded3(tblen+1:end),msg(1:end-tblen));
disp(['The bit error rates are:   ',num2str([r1 r2 r3])])
```

The output is

```
The bit error rates are:   0.020619    0.020619    0.020619
```

The example below illustrates how to use the final state and initial state arguments when invoking `vitdec` repeatedly. Notice that `[decoded4;decoded5]` is the same as `decoded6`.

```
trel = poly2trellis(3,[6 7]);
code = convenc(randint(100,1,2,123),trel);
% Decode part of code, recording final state for later use.
[decoded4,f1,f2,f3] = vitdec(code(1:100),trel,3,'cont','hard');
% Decode the rest of code, using state input arguments.
decoded5 = vitdec(code(101:200),trel,3,'cont','hard',f1,f2,f3);
% Decode the entire code in one step.
decoded6 = vitdec(code,trel,3,'cont','hard');
isequal(decoded6,[decoded4; decoded5])
```

The output is

```
ans =

     1
```

For additional examples, see “Examples of Convolutional Coding” on page 6-40.

See Also

`convenc`, `poly2trellis`, `istrellis`, `vitsimdemo`, `vitsimexample`, “Convolutional Coding” on page 6-30

References

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

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.

Examples

To generate a column vector of length 100 containing real white Gaussian noise of power 0 dBW, use this command:

```
y1 = wgn(100,1,0);
```

To generate a column vector of length 100 containing complex white Gaussian noise, each component of which has a noise power of 0 dBW, use this command:

```
y2 = wgn(100,1,0,'complex');
```

See Also `randn`, `awgn`, Chapter 2, “Signal Sources”

A

Examples

Use this list to find examples in the documentation.

Modulation

Special Filters

Convolutional Coding

Simulating Communication Systems

Performance Evaluation

Source Coding

Block Coding

Interleaving

Channels

Equalizers

Galois Field Computations

Index

A

B

C

D

E

F

G

H

I

J

K

L

P

Q

R

S

T

V

W