Communications Toolbox™ 4
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science
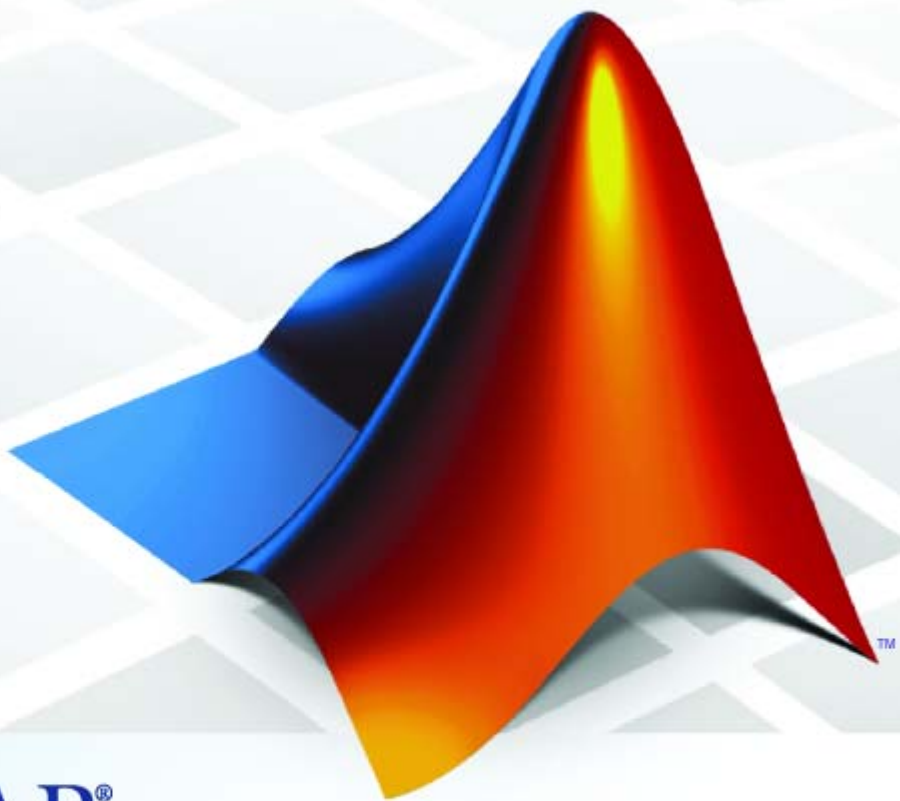
How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Communications Toolbox™ User's Guide

Trademarks

Patents

Contents

Error Rate Test Console

4

BERTool: A Bit Error Rate Analysis GUI

5

Source Coding

6

Error Detection and Correction

7

Interleaving

8

Modulation

9

Special Filters

10

Channels

11

Equalizers

12

Galois Field Computations

13

EyeScope: An Eye Diagram Analysis Tool

14

Galois Fields of Odd Characteristic

A

Analytical Expressions Used in berawgn, bercoding, berfading, and BERTool

B

Algorithms

C

Examples

D

Index

Getting Started

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

- "Product Overview" on page 1-2
- "Studying Components of a Communication System" on page 1-4
- "Simulating a Communication System" on page 1-23
- "Running Simulations Using the Error Rate Test Console" on page 1-35
- "Learning More" on page 1-52

Product Overview

In this section...
"Section Overview" on page 1-2
"Expected Background" on page 1-2

Section Overview

Communications Toolbox software 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

In this section...

Section Overview

Communications Toolbox software 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 Communications Toolbox functionalities build upon the computational and visualization tools in the underlying MATLAB environment.

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 following table indicates the key tasks in solving the problem, along with relevant Communications Toolbox functions. 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 or Method
Generate a random binary data stream	`randint`
Modulate using 16-QAM	`modulate` method on `modem.qammod` object
Add white Gaussian noise	`awgn`
Create a scatter plot	`scatterplot`
Demodulate using 16-QAM	`modulate` method on `modem.qamdemod` object
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
hMod = modem.qammod(M);  % Create a 16-QAM modulator

%% 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');
```
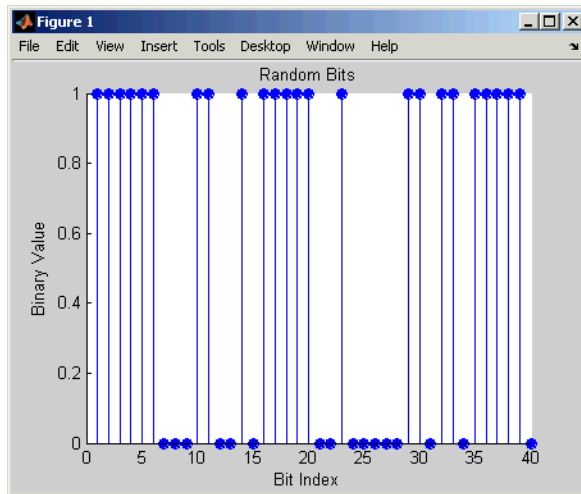
2. Prepare to Modulate. The `modem.qammod` object implements an M-ary QAM modulator, M being 16 in this example. It is configured to receive integers between 0 and 15 rather than 4-tuples of bits. Therefore, you must preprocess the binary data stream x before using the `modulate` method of the object. 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');
```
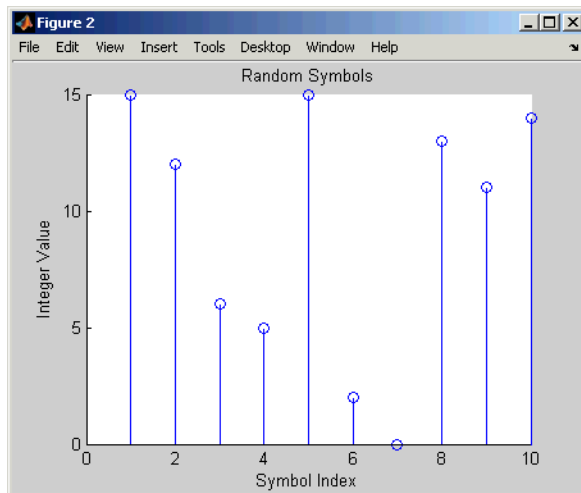
3. Modulate Using 16-QAM. Having defined xsym as a column vector containing integers between 0 and 15, you can use the modulate method of the modem.qammod object to modulate xsym using the baseband representation. Recall that M is 16, the alphabet size.

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

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 9, "Modulation". Also, note that themodulate method of the modem.qammod object does not apply any pulse shaping. To extend this example to use pulse shaping, see "Pulse Shaping Using a Raised Cosine Filter" on page 1-15. 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 11, "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;
```
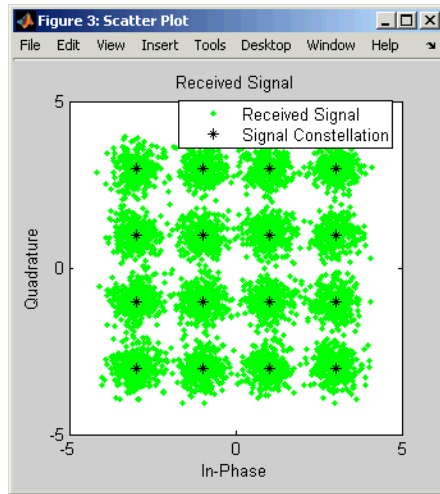
To learn more about scatterplot, see "Scatter Plots" on page 3-21.

6. Demodulate Using 16-QAM. Applying the demodulate method of the modem.qamdemod object 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 = demodulate(modem.qamdemod(M),yrx);
```

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.',numel(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. The Constellation property of the modem.qammod object contains all points in the 16-QAM signal constellation.

```
M = 16; % Number of points in constellation
h=modem.qammod(M); % Modulator object
mapping=h.SymbolMapping; % Symbol mapping vector
pt = h.Constellation; % Vector of all points in constellation
```

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

```
% Plot the constellation.
scatterplot(pt);
```
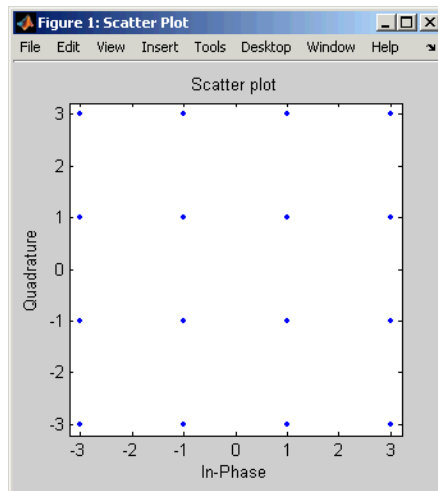
3. Annotate the Plot to Indicate the Mapping. To annotate the plot to show the relationship between `mapping` and `pt`, use the `text` function to place a number in the plot beside each constellation point. The coordinates of the annotation are near the real and imaginary parts of the constellation point, but slightly offset to avoid overlap. The text of the annotation comes from the binary representation of `mapping`. (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(mapping));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.
```
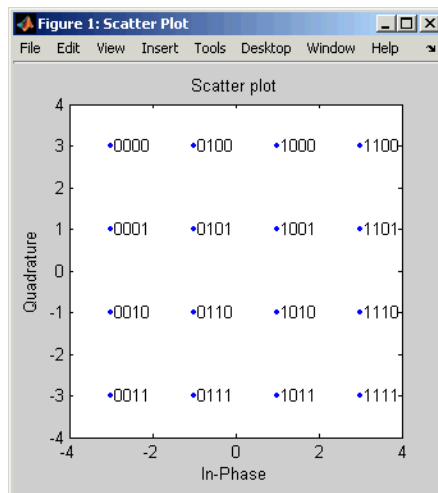

Binary-Coded 16-QAM Signal Constellation

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 the `modem.qammod` object 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.
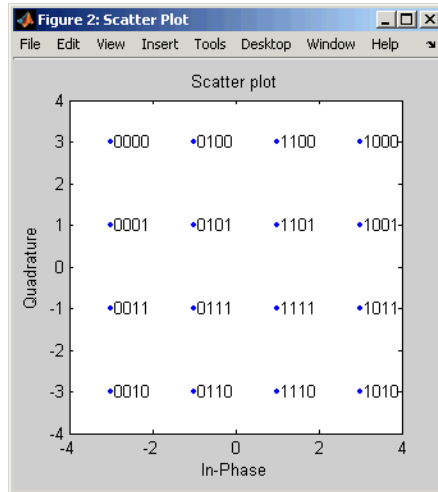
Gray-Coded 16-QAM Signal Constellation

The only difference, compared to the previous example, is that you configure `modem.qammod` object to use a Gray-coded constellation.

```
%% Modified Plot, With Gray Coding
M = 16; % Number of points in constellation
h = modem.qammod('M',M,'SymbolOrder','Gray'); % Modulator object
mapping = h.SymbolMapping; % Symbol mapping vector
pt = h.Constellation; % 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(mapping));
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 10-7 or the `rcosdemo` demonstration for more details.

Solution of Problem

This solution modifies the code from `commdoc_gray.m`. 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, 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);
```
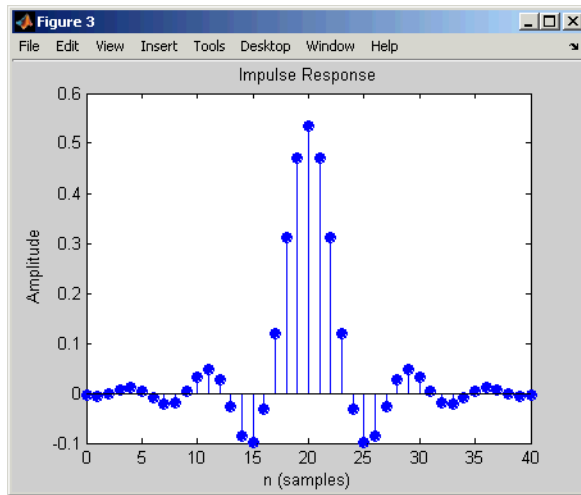

3. Filter the Modulated Signal. To filter the modulated signal, replace the
Transmitted Signal section with following.

```
%% 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.
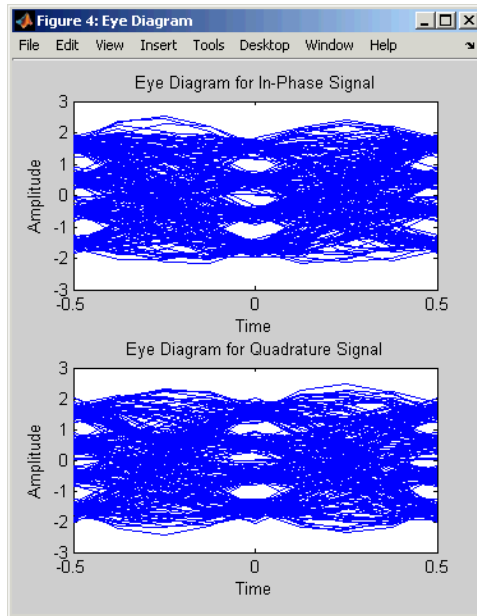
To learn more about `eyediagram`, see "Eye Diagrams" on page 3-20.

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.
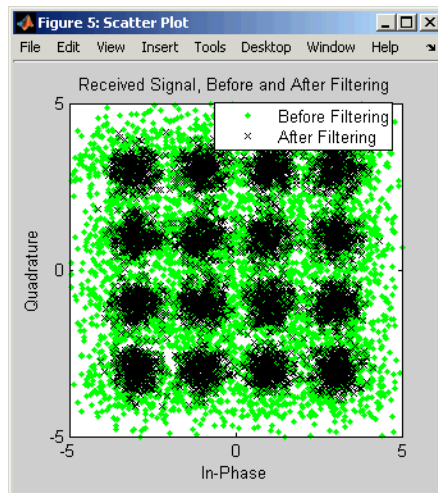
Using a Convolutional Code

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 7-32.

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 1-15. 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 7-42.

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 1-22.

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))
```

1-21

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 8-9
- "Delays from Equalization" on page 12-21
- "Example: Compensating for Group Delays in Data Analysis" on page 10-3
- "Fading Channels" on page 11-7

The "Effect of Delays on Recovery of Convolutionally Interleaved Data" on page 8-10 discussion also includes two typical ways to compensate for delays.

Simulating a Communication System

In this section...
"Section Overview" on page 1-23
"Using BERTool to Run Simulations" on page 1-23
"Varying Parameters and Managing a Set of Simulations" on page 1-31

Section Overview

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.

Using BERTool to Run Simulations

Communications Toolbox software includes a graphical user interface called BERTool. Using the BERTool GUI, you can solve problems like the following:

Problem Modify the modulation example in "Modulating a Random Signal" on page 1-4 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 "Modulating a Random Signal" on page 1-4.

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

- "Solution of Problem" on page 1-24
- "Comparing with Theoretical Results" on page 1-28
- "More About the Simulation Structure" on page 1-30

However, this section is not a comprehensive description of BERTool; for more information about BERTool, see Chapter 5, "BERTool: A Bit Error Rate Analysis GUI".

Solution of Problem

This solution uses code from commdoc_gray.m 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 commdoc_gray.m.

```
% 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 commdoc_gray.m.

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.

4. 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;
```

1-25

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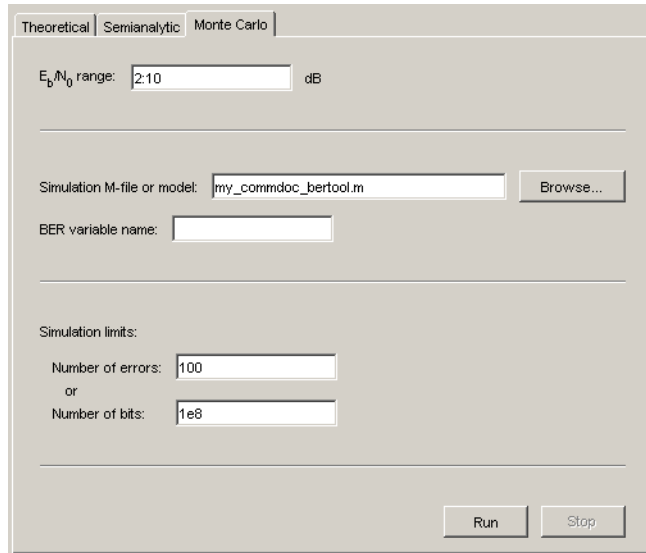
```
% EbNo = 10; % In dB % COMMENT OUT FOR BERTOOL
```

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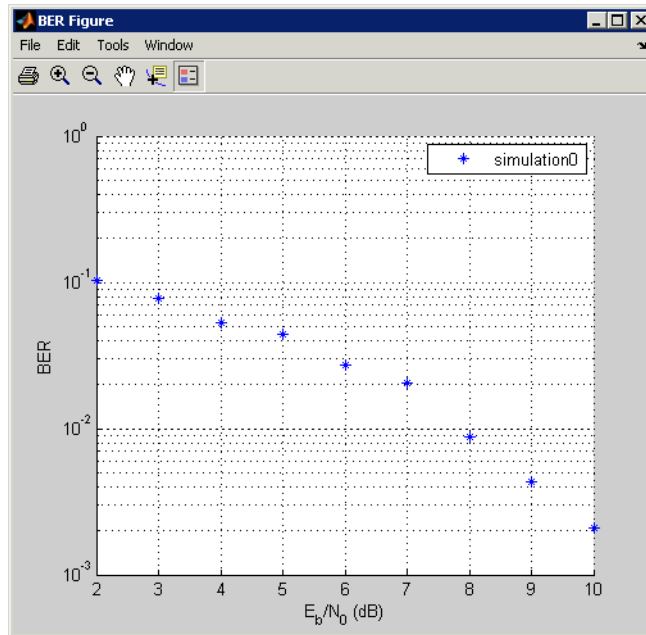
To compare these BER results with theoretical results, leave BERTool open and use the procedure below.

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.
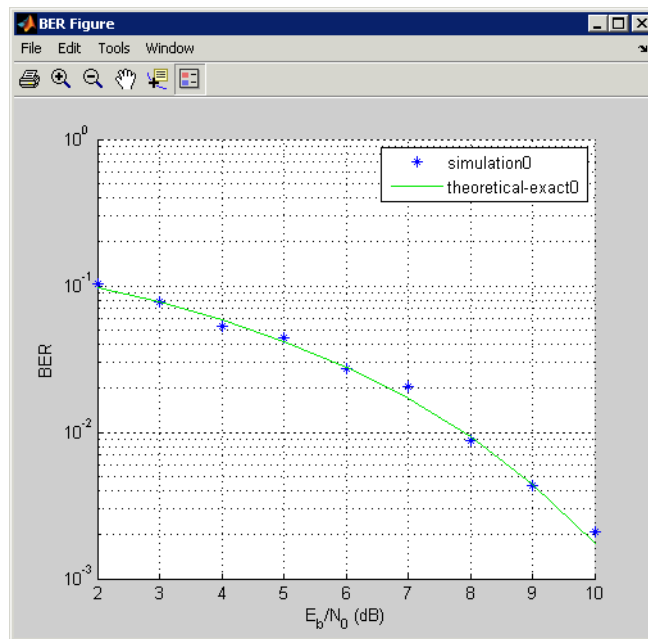
Notice that the plotting markers are close to the theoretical curve. It is relevant that the simulation code used a Gray-coded signal constellation, unlike the first modulation example of this chapter (in "Modulating a

1-29

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 5, "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 11-7

- "Equalizing Using a Loop" on page 12-22

- "Equalizing in Continuous Operation Mode" on page 12-30

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`commdoc_gray.m` 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 practice 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');
```

1-33

```
legend('M = 4','M = 8','M = 16','M = 32',...
    'Location','SouthWest');
```

6. Run the Entire Script. The script creates a plot like the one shown in the following figure.
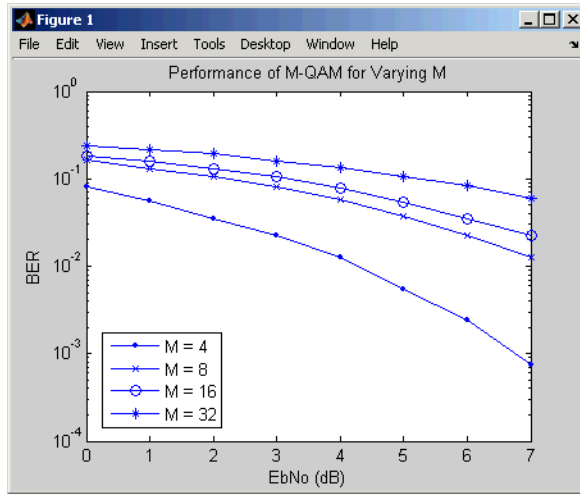
Running Simulations Using the Error Rate Test Console

In this section...
"Loading the Error Rate Test Console" on page 1-35
"Running the Simulation and Obtaining Results" on page 1-36
"Generating an Error Rate Results Figure Window" on page 1-37
"Running the Simulation Using Parallel Computing Toolbox Software" on page 1-39
"Creating a System File and Attaching It to the Test Console" on page 1-40
"Configuring the Test Console and Running a Simulation" on page 1-44
"Optimizing Your System for Faster Simulations" on page 1-47

Loading the Error Rate Test Console

The Error Rate Test Console is a simulation tool for obtaining error rate results. The MATLAB™ software includes a data file for use with the Error Rate Test Console. The data file contains an Error Rate Test Console object with an attached Gray coded modulation system. The test console configuration runs bit error rate simulations for various EbNo and modulation order, or M, values.

1 Load the file containing the Error Rate Test Console and attached Gray coded modulation system. At the MATLAB command line, type:

```
load GrayCodedModulationTester
```

2 Examine the test console by displaying its properties. At the MATLAB command line, type:

```
testConsole
```

MATLAB returns the following output:

```
testConsole =

                Description: 'Error Rate Test Console'
        SystemUnderTestName: 'GrayCodedModulation2'
```

```
                IterationMode: 'Combinatorial'
            SystemResetMode: 'Reset at new simulation point'
        SimulationLimitOption: 'Number of errors or transmissions'
    TransmissionCountTestPoint: 'DemodBitErrors'
          MaxNumTransmissions: 100000000
          ErrorCountTestPoint: 'DemodBitErrors'
                  MinNumErrors: 100
```

Notice that SystemUnderTest is a Gray coded modulation system, and
simulations run until reaching 100 errors or 1e8 bits.

Running the Simulation and Obtaining Results

1 Run the simulation, using the tic and toc commands to measure
simulation time. At the MATLAB command line, type:

```
tic; run(testConsole); toc
```

MATLAB returns output similar to the following:

```
Running simulations...
Elapsed time is 275.671536 seconds.
```

2 Obtain the results of the simulation using the getResults method by
typing the following at the MATLAB command line:

```
grayResults = getResults(testConsole)
```

MATLAB returns the following output:

```
grayResults =

        TestConsoleName: 'commtest.ErrorRate'
    SystemUnderTestName: 'GrayCodedModulation2'
          IterationMode: 'Combinatorial'
              TestPoint: 'DemodBitErrors'
                 Metric: 'ErrorRate'
         TestParameter1: 'EbNo'
         TestParameter2: 'None'
```

In the next section, you use the results object to obtain error values and plot error rate curves.

Generating an Error Rate Results Figure Window

The `semilogy` method generates a figure containing error rate curves for the demodulator bit error test points. The next figure shows EbNo and Error Rate curve for the demodulator bit errors test point. This test point collects bit errors by comparing the bits the Gray coded 16-QAM modulation system transmits with the bits the system receives. In this example, the `TestParameter1` property of `grayResults` represents EbNo.

1 Generate the figure by typing the following at the MATLAB command line:

```
semilogy(grayResults)
```

This script generates the following figure.
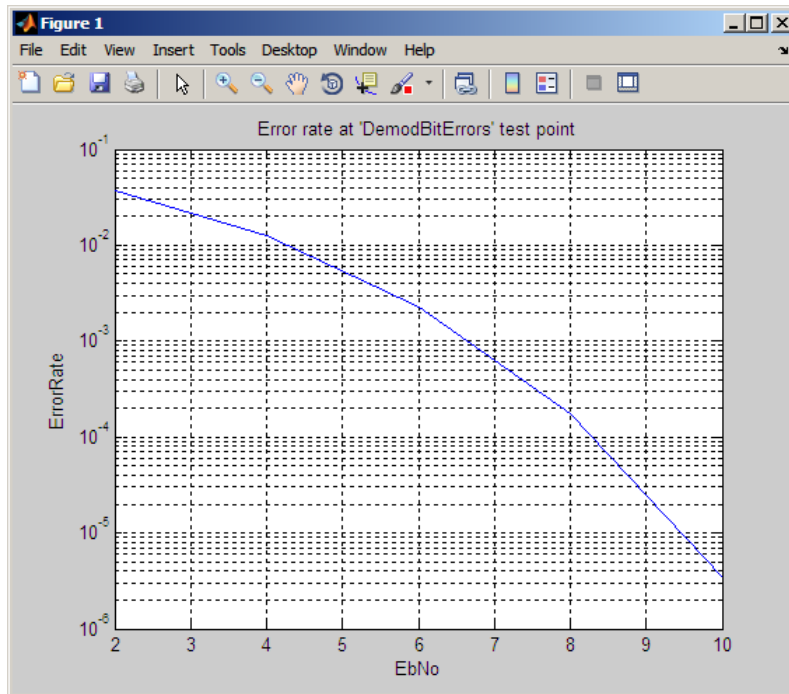
2 Generate the error rate curves for multiple M values by setting the `TestParameter2` property to M. At the MATLAB command line, type:

```
grayResults.TestParameter2 = 'M'
```

The simulation runs for multiple modulation order (M) values. `TestParameter1` controls the x-axis parameter, which is EbNo. This curve contains data for M=2, even though the actual simulation contains multiple M values.

MATLAB returns the following output:

```
grayResults =

        TestConsoleName: 'commtest.ErrorRate'
    SystemUnderTestName: 'GrayCodedModulation2'
          IterationMode: 'Combinatorial'
              TestPoint: 'DemodBitErrors'
                 Metric: 'ErrorRate'
         TestParameter1: 'EbNo'
         TestParameter2: 'M'
```

3 Plot the error rate curves by typing the following at the MATLAB command line.

```
semilogy(grayResults)
```

This script generates the following figure.
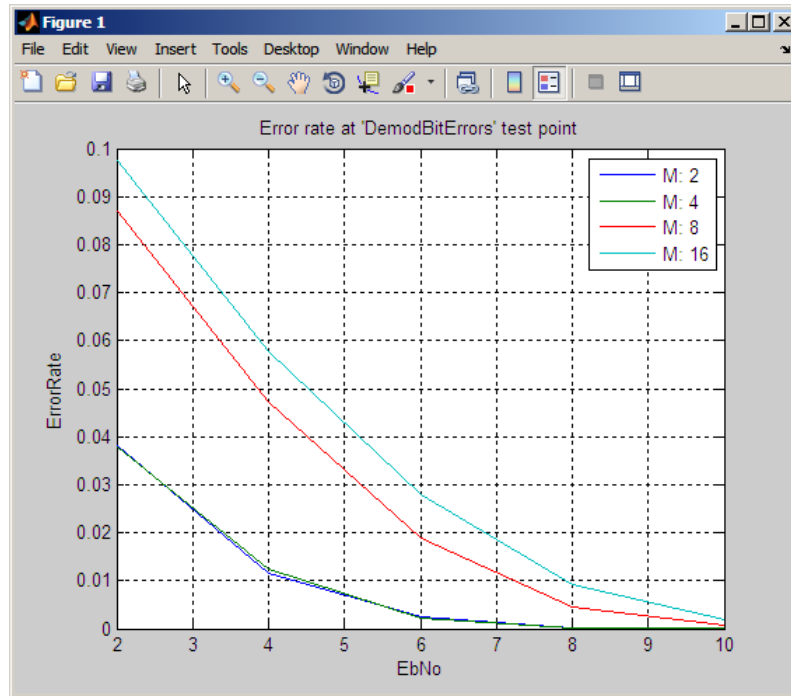
Running the Simulation Using Parallel Computing Toolbox Software

If you have a Parallel Computing Toolbox™ user license and you create a matlabpool, the test console runs the simulation in parallel. This approach reduces the processing time.

Note If you do not have a Parallel Computing Toolbox user license you are unable to perform this section of the tutorial.

1 If you have a Parallel Computing Toolbox license, run the following command to start your default matlabpool:

```
matlabpool
```

1-39

If you have a multicore computer, then the default matlabpool uses the cores as workers.

2 Using the workers, run the simulation. At the MATLAB command line, type:

```
tic; run(testConsole); toc
```

MATLAB returns output similar to the following:

```
4 workers available for parallel computing. Simulations ...,
will be distributed among these workers.
Running simulations...
Elapsed time is 87.449652 seconds.
```

Notice that the simulation runs more than three times as fast than in the previous section.

Creating a System File and Attaching It to the Test Console

In the previous sections, you used an existing Gray coded modulator system file to generate data. In this section, you create a system file and then attach it to the Error Rate Test Console.

For this example, use commdoc_gray as a starting point for your system file. This example outlines the tasks necessary for converting legacy code to a system file you can attach to the Error Rate Test Console.

1 Copy the system basic API template, SystemBasicTemplate.m, as MyGrayCodedModulation.m.

2 Rename the references to the system name in the file. First, rename the system definition by changing the class name to MyGrayCodedModulation. Replace the following lines, lines 1 and 2, of the file:

```
classdef SystemBasicTemplate < testconsole.SystemBasicAPI
%SystemBasicTemplate Template for creating a system
```

with these lines:

```
classdef MyGrayCodedModulation < testconsole.SystemBasicAPI
%MyGrayCodedModulation Gray coded modulation system
```

3 Rename the constructor by replacing:

```
function obj = SystemBasicTemplate
%SystemBasicTemplate Construct a system
```

with

```
function obj = MyGrayCodedModulation
%MyGrayCodedModulation Construct a Gray coded modulation system
```

4 Enter a description for your system. Update the `obj.Description` parameter with the following information:

```
obj.Description = 'Gray coded modulation';
```

Because you are not using the `reset` and `setup` methods for this system, leave these methods empty.

5 Copy lines 12–44 from commdoc_gray.m to the body of the `run` method.

6 Copy Lines 54–57 from commdoc_gray.m to the body of the `run` method.

7 Change EbNo to a test parameter. As a test parameter, EbNo becomes a variable, which allows simulations to run for different values. Locate the following line of syntax in the file:

```
EbNo = 10; % In dB
```

Replace it with:

```
EbNo = getTestParameter(obj,'EbNo');
```

8 Add modulation order, M, as a new test parameter for the simulation. Locate the following syntax:

```
 M = 16;                        % Size of signal constellation
```

Replace it with:

```
M = getTestParameter(obj,'M');
```

9 Register the test parameters to the test console.

- Declare EbNo as a test parameter by placing the following line of code in the body of the register method:

  ```
  registerTestParameter(obj,'EbNo',0,[-50 50]);
  ```

  The parameter defaults to 0 dB and can take values between -50 dB and 50 dB.

- Declare M as a test parameter by placing the following line of code in the body of the register method:

  ```
  registerTestParameter(obj,'M',16,[2 1024]);
  ```

  The parameter defaults to 16 QAM Modulation and can take values from 2 through 1024.

10 Add EbNo and M to the test parameters list in the MyGrayCodedModulationFile file.

```
% Test Parameters
   properties
       EbNo = 0;
 M = 16;
   end
```

This adds EbNo and M to the possible test parameters list. EbNo defaults to a value of 0 dB. M defaults to a value of 16.

11 Define test probe locations in the run method. In this example, you are calculating end-to-end error rate. This calculation requires transmitted bits and received bits. Add one probe for obtaining transmitted bits and one probe for received bits.

- Locate the random binary data stream creation code by searching for the following lines:

  ```
  % Create a binary data stream as a column vector.
  x = randi([0 1],n,1); % Random binary data stream
  ```

- Add a probe, TxBits, after the random binary data stream creation:

```
% Create a binary data stream as a column vector.
x = randi([O 1],n,1); % Random binary data stream
setTestProbeData(obj,'TxBits',x);
```

This code sends the random binary data stream, *x*, to the probe TxBits.

- Locate the demodulation code by searching for the following lines:

```
% Demodulate signal using 16-QAM.
z = demodulate(hDemod,yRx);
```

- Add a probe, RxBits, after the demodulation code.

```
% Demodulate signal using 16-QAM.
z = demodulate(hDemod,yRx);
setTestProbeData(obj,'RxBits',z);
```

This code sends the binary received data stream, *z*, to the probe RxBits.

12 Add the test probes to the test console, making it possible to obtain data from the system. Add these probes to the function `register(obj)` by typing:

```
function register(obj)
% REGISTER Register the system with a test console
% REGISTER(H) registers test parameters and test probes of the
% system, H, with a test console.

            registerTestParameter(obj,'EbNo',O,[-50 50]);

            registerTestProbe(obj,'TxBits')
            registerTestProbe(obj,'RxBits')
        end
```

13 Save the file. The file is ready for use with the system.

14 Create a Gray coded modulation system by typing the following at the MATLAB command line:

```
mySystem = MyGrayCodedModulation
```

MATLAB returns the following output:

```
mySystem =

    Description: 'Gray coded modulation'
           EbNo: 0
M: 16
```

15 Create an Error Rate Test Console by typing the following at the MATLAB command line:

```
testConsole = commtest.ErrorRate
```

The MATLAB software returns the following output:

```
testConsole =

                     Description: 'Error Rate Test Console'
              SystemUnderTestName: 'commtest.MPSKSystem'
                      FrameLength: 500
                    IterationMode: 'Combinatorial'
                  SystemResetMode: 'Reset at new simulation point'
            SimulationLimitOption: 'Number of transmissions'
         TransmissionCountTestPoint: 'Not set'
               MaxNumTransmissions: 1000
```

16 Attach the system file MyGrayCodedModulation to the error rate test console by typing the following at the MATLAB command line:

```
attachSystem(testConsole, mySystem)
```

Configuring the Test Console and Running a Simulation

1 Configure the test console to obtain error rate metrics from the attached system. The Error Rate Test Console defines metrics as number of errors, number of transmissions, and error rate.

At the MATLAB command line, type:

```
registerTestPoint(testConsole, 'DemodBitErrors', 'TxBits',
'RxBits');
```

This line defines the test point, DemodBitErrors, and compares bits from the TxBits probe to the bits from the RxBits probe.

2 Run the simulations for EbNo values starting at 2 dB and ending at 10 dB, with a step size of 2 dB and M values of 2, 4, 8, and 16. At the MATLAB command line, type:

```
setTestParameterSweepValues(testConsole, 'EbNo', 2:2:10)
setTestParameterSweepValues(testConsole, 'M', [2 4 8 16])
```

3 Set the simulation limit to the number of transmissions.

```
testConsole.SimulationLimitOption = 'Number of transmissions'
```

4 Set the maximum number of transmissions to 1000.

```
testConsole.MaxNumTransmissions = 1000
```

5 Configure the Error Rate Test Console so it uses the demodulator bit error test point for determining the number of transmitted bits.

```
testConsole.TransmissionCountTestPoint = 'DemodBitErrors'
```

6 Run the simulation. At the MATLAB command line, type:

```
run(testConsole)
```

7 Obtain the results of the simulation. At the MATLAB command line, type:

```
grayResults = getResults(testConsole)
```

8 To obtain more accurate results, run the simulations for a given minimum number of errors. In this example, you also limit the number of simulation bits so that the simulations do not run indefinitely. At the MATLAB command line, type:

```
testConsole.SimulationLimitOption = 'Number of errors or transmissions';
testConsole.MinNumErrors = 100;
testConsole.ErrorCountTestPoint = 'DemodBitErrors';
```

1-45

```
testConsole.MaxNumTransmissions = 1e8;
testConsole
```

9 Run the simulation by typing the following at the MATLAB command line.

```
tic; run(testConsole); toc
```

10 Generate the new results in a Figure window by typing the following at the MATLAB command line.

```
grayResults = getResults(testConsole);
grayResults.TestParameter2 = 'M'
plot(grayResults)
```

This script generates the following figure.
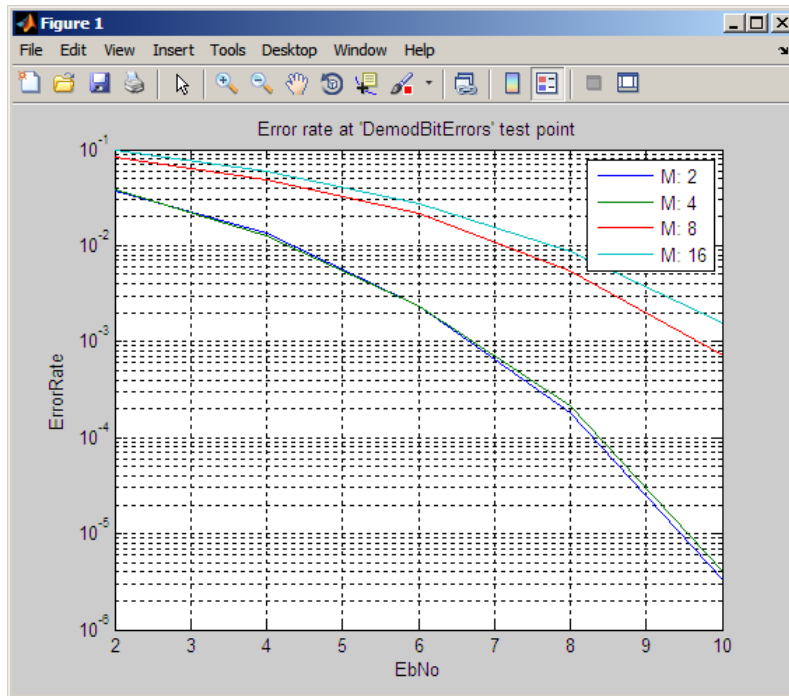
Optimizing Your System for Faster Simulations

In the previous example, the system only utilizes the run method. Every time the object calls the run method, which is every 3e4 bits for this simulation, the object sets the M and SNR values. This time interval includes: obtaining numbers from the test console, calculating intermediate values, and setting other variables.

In contrast, the system basic API provides a setup method where the Error Rate Test Console configures the system once for each simulation point. This change relieves the run method from getting and setting simulation parameters, thus reducing simulation time.

1 Save the file MyGrayCodedModulation as MyGrayCodedModulationOptimized.

2 In the MyGrayCodedModulationOptimized file, replace the constructor name and the class definition name.

- Locate the following lines of code:

```
classdef MyGrayCodedModulation < testconsole.SystemBasicAPI
%MyGrayCodedModulation Gray coded modulation system
```

- Replace them with:

```
classdef MyGrayCodedModulationOptimized < ...,
testconsole.SystemBasicAPI
%MyGrayCodedModulationOptimized Gray coded modulation system
```

3 Cut lines 58–60, 62, and 84–88 and paste them into the setup method.

4 Replace M with the object property M and EbNo with the object property EbNo. This change provides access to these values from the run method.

```
obj.M = getTestParameter(obj,'M');
obj.EbNo = getTestParameter(obj,'EbNo');
```

5 The object calculates the SNR value once and accesses it in the run method. Define an SNR property and store the SNR value in this property. Define this property as a private property because it is not a test parameter, but an internal variable. At the MATLAB command line, type:

1-47

```
% Internal variables
 properties (Access = private)
     SNR
 end

    function setup(obj)
        %SETUP    Initialize the system
        % SETUP(H) gets current test parameter value(s) from the test
        % console and initializes system, H, accordingly.

        % Get M
        obj.M = getTestParameter(obj,'M');
        k = log2(obj.M);                  % Number of bits per symbol

        % Get EbNo and calculate SNR
        obj.EbNo = getTestParameter(obj,'EbNo');
        obj.SNR = obj.EbNo + 10*log10(k) - 10*log10(nSamp);
    end
```

6 In the run method, replace M with obj.M and SNR with obj.SNR.

```
         hMod = modem.qammod(obj.M);    % Create
a 16-QAM modulator
         yNoisy = awgn(yTx,obj.SNR,'measured');
```

Notice that the run method creates the QAM modulator and demodulator.

7 Move the QAM modulator and demodulator out of the run method by creating hMod and hDemod properties and adding two new internal variables, Modulator and Demodulator. Storing the QAM modulator and demodulator as properties moves their creation out of the run method while still providing access to them.

```
% Internal variables
properties (Access = private)
    SNR
    Modulator
    Demodulator
end
```

8 Create the modulator and demodulator once for the entire simulation run, by defining them in the constructor instead of the `run` method.

Cut lines 78–83 and paste them in the constructor.

```
function obj = GrayCodedModulationOptimized
    % GrayCodedModulationOptimized Construct
  % a Gray coded modulation system

     obj.Description = 'Gray coded modulation';

     %% Create Modulator and Demodulator
     hMod = modem.qammod(M);         % Create a 16-QAM modulator
     hMod.InputType = 'Bit';         % Accept bits as inputs
     hMod.SymbolOrder = 'Gray';      % Accept bits as inputs
     hDemod = modem.qamdemod(hMod);
  end
```

9 Locate the following lines of code, which creates the modulator:

```
hMod = modem.qammod(M);         % Create a 16-QAM modulator
hMod.InputType = 'Bit';         % Accept bits as inputs
hMod.SymbolOrder = 'Gray';      % Accept bits as inputs
```

Replace them with these lines of code:

```
obj.Modulator = modem.qammod(obj.M);     % Create a QAM modulator
obj.Modulator.InputType = 'Bit';         % Accept bits as inputs
obj.Modulator.SymbolOrder = 'Gray';      % Accept bits as inputs
```

10 Locate the following line of code, which creates the demodulator.

```
hDemod = modem.qamdemod(hMod);
```

Replace it with the following line of code:

```
obj.Demodulator = modem.qamdemod(obj.Modulator);
```

11 Locate the setup region of the file:

```
function setup(obj)
% SETUP    Initialize the system
```

1-49

```
    % SETUP(H) gets current test parameter value(s) from the test
    %  console and initializes system, H, accordingly.
```

12 Set the M value of the modulator and demodulator by adding the following lines of code to the setup:

```
nSamp = 1;                        % Oversampling rate

% Get M and calculate k
obj.M = getTestParameter(obj,'M');
k = log2(obj.M);                  % Number of bits per symbol
obj.Modulator.M = obj.M;
obj.Demodulator.M = obj.M;

% Get EbNo and calculate SNR
obj.EbNo = getTestParameter(obj,'EbNo');
obj.SNR = obj.EbNo + 10*log10(k) - 10*log10(nSamp);
end
```

13 Save the file.

14 Run the simulation by typing the following at the MATLAB command line.

```
tic; run(testConsole); toc
```

Notice how quickly the simulation runs with the new code changes.

15 Generate the new results in a Figure window by typing the following at the MATLAB command line.

```
grayResults = getResults(testConsole);
grayResults.TestParameter2 = 'M'
plot(grayResults)
```

This script generates the following figure.
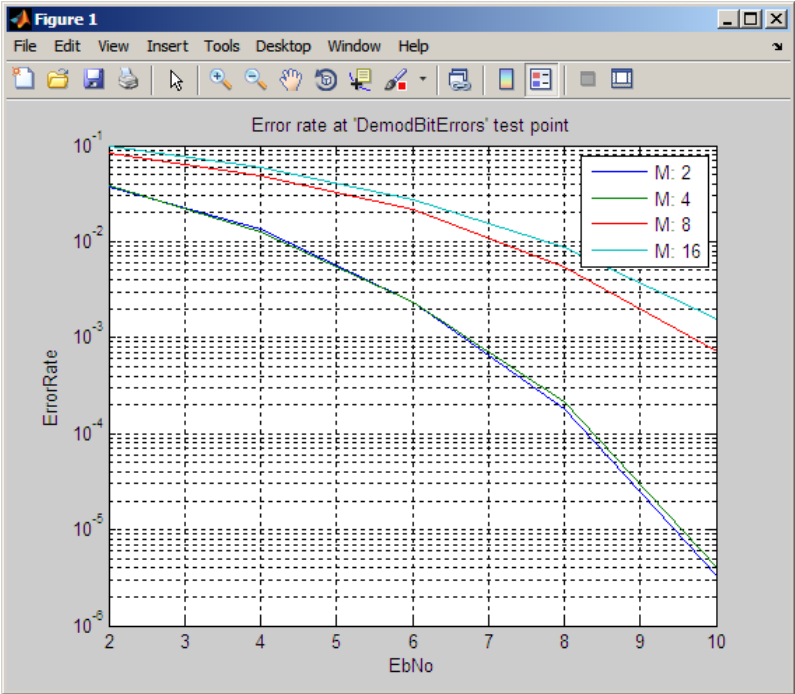
Learning More

In this section...
"Online Help" on page 1-52
"Demos" on page 1-52
"The MathWorks Online" on page 1-52

Online Help

To find online documentation, select **Product 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 Communications Toolbox documentation 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 Communications Toolbox documentation on the MathWorks Web site, point your Web browser to

```
http://www.mathworks.com/access/helpdesk/help/toolbox/comm/
```

Other resources for the Communications Toolbox product are available at

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

Signal Sources

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

- "White Gaussian Noise" on page 2-2
- "Random Symbols" on page 2-3
- "Random Integers" on page 2-4
- "Random Bit Error Patterns" on page 2-5

For more 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. 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 11-3 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, use the command below to prescribe the skewed distribution. 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], 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 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, where 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) introduces exactly one error into each codeword.

On the other hand, to perturb each codeword by introducing one error with probability 0.4 and two errors with probability 0.6, use the command below instead.

```
% 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*. 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

In this section...
"Section Overview" on page 3-2
"Using Simulated Data to Compute Bit and Symbol Error Rates" on page 3-2
"Example: Computing Error Rates" on page 3-3
"Comparing Symbol Error Rate and Bit Error Rate" on page 3-4

Section Overview

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 Communications Toolbox is covered in other parts of this guide. This section describes how to compare the data messages that enter and leave the simulation.

Another example of computing performance results via simulation is in "Curve Fitting for Error Rate Plots" on page 3-15 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, 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), 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 this example uses random numbers.

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

Comparing 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 9. 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

In this section...
"Section Overview" on page 3-5
"When to Use the Semianalytic Technique" on page 3-5
"Procedure for the Semianalytic Technique" on page 3-6
"Example: Using the Semianalytic Technique" on page 3-7

Section Overview

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 Communications Toolbox helps you implement the semianalytic technique by performing some of the analysis.

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, the calculated BER is too low. For instance, if the modeled system has a phase rotation that places the received signal points outside their proper decision regions, 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`. Shape the resultant signal with rectangular pulse shaping, using the oversampling factor that you will later use to filter the modulated signal. Store the result of this step as `txsig` for later use.

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. If you use a square-root raised cosine filter, use it on the nonoversampled modulated signal and specify the oversampling factor in the filtering function. If you use another filter type, you can apply it to the rectangularly pulse shaped signal.

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, 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(1i*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;
```

This example creates a figure like the one below.
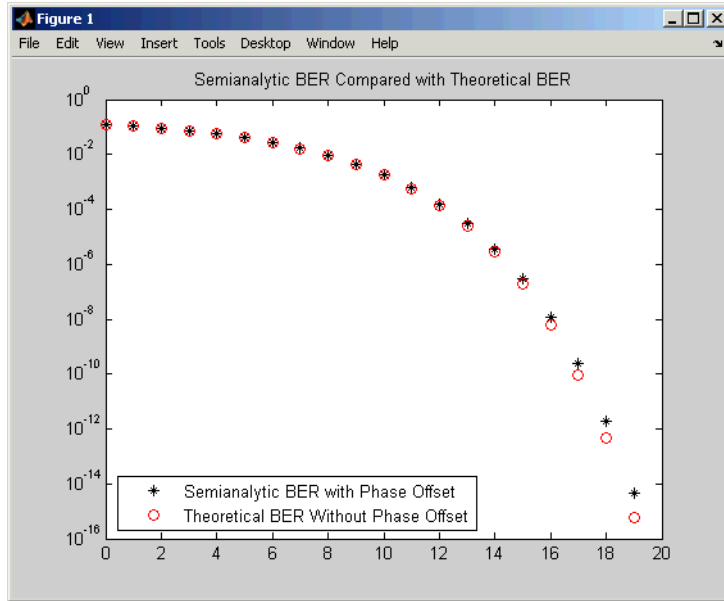
Theoretical Performance Results

In this section...
"Computing Theoretical Error Statistics" on page 3-10
"Plotting Theoretical Error Rates" on page 3-10
"Comparing Theoretical and Empirical Error Rates" on page 3-11

Computing Theoretical Error Statistics

While the biterr function discussed above can help you gather empirical
error statistics, you might also compare those results to 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 and Rician 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_O (dB)'); ylabel('Upper Bound on BER');
title('Theoretical Bound on BER for Convolutional Coding');
grid on;
```

This example produces the following plot.
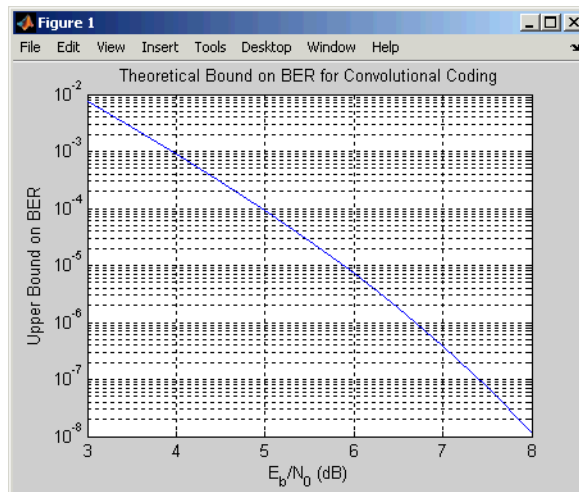
Comparing Theoretical and Empirical Error Rates

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];
[ber, ser] = berawgn(EbNo,'pam',M);
```

```
% Plot theoretical results.
figure; semilogy(EbNo,ser,'r');
xlabel('E_b/N_O (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;
```

This example produces a plot like the one in the following figure. Your plot might vary because the simulation uses random numbers.
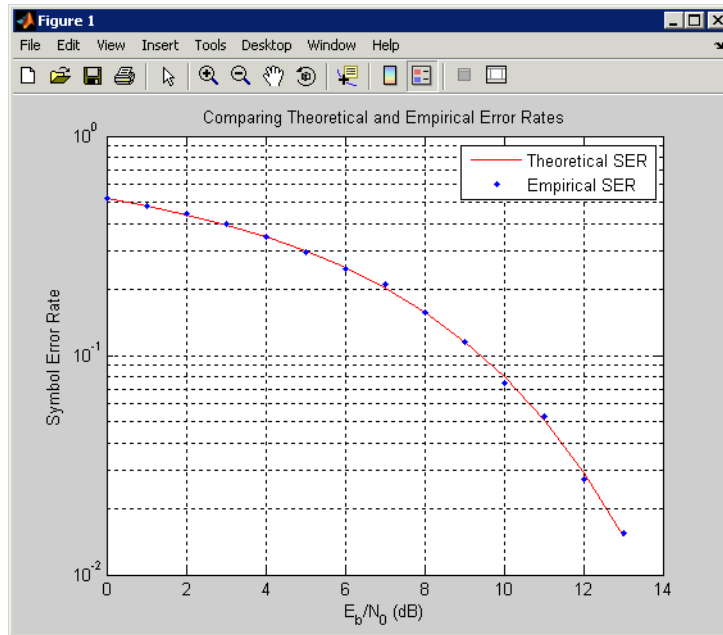
Error Rate Plots

In this section...

Section Overview

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 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 3-11. 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 3-10, 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 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" on page 3-16
- "Simulating the System Using a Loop" on page 3-16

• "Plotting the Empirical Results and the Fitted Curve" on page 3-18

Setting Up Parameters for the Simulation

The first step in the example sets up the parameters to 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, 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 = 100000; % Number of bits in each trial
M = 2; % DBPSK is binary.
hMod = modem.dpskmod('M', M); % Create a DPSK modulator
hDemod = modem.dpskdemod(hMod); % Create a DPSK
                    % demodulator using the modulator object
EbNomin = 0; EbNomax = 9; % 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 = modulate(hMod, msg); % Modulate.
      rxsig = awgn(txsig, snr, 'measured'); % Add noise.
      decodmsg = demodulate(hDemod, rxsig); % Demodulate.
      if (ntrials==0)
         % The first symbol of a differentially encoded transmission
         % is discarded.
         newerrs = biterr(msg(2:end),decodmsg(2:end)); ...
% Errors in this trial
      else
```

```
            newerrs = biterr(msg,decodmsg); % Errors in this trial
        end
        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)-1,.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 this example uses random numbers.

```
EbNo = 0 dB, 189 errors, BER = 0.18919
EbNo = 1 dB, 139 errors, BER = 0.13914
EbNo = 2 dB, 105 errors, BER = 0.10511
EbNo = 3 dB, 66 errors, BER = 0.066066
EbNo = 4 dB, 40 errors, BER = 0.04004
EbNo = 5 dB, 18 errors, BER = 0.018018
EbNo = 6 dB, 6 errors, BER = 0.006006
EbNo = 7 dB, 11 errors, BER = 0.0055028
EbNo = 8 dB, 5 errors, BER = 0.00071439
EbNo = 9 dB, 5 errors, BER = 0.00022728
EbNo = 10 dB, 5 errors, BER = 1.006e-005
```

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,[],'exp');

% Also plot confidence intervals.
```

```
hold on;
for jj=1:numEbNos
    semilogy([EbNovec(jj) EbNovec(jj)],intv{jj},'g-+');
end
hold off;
```
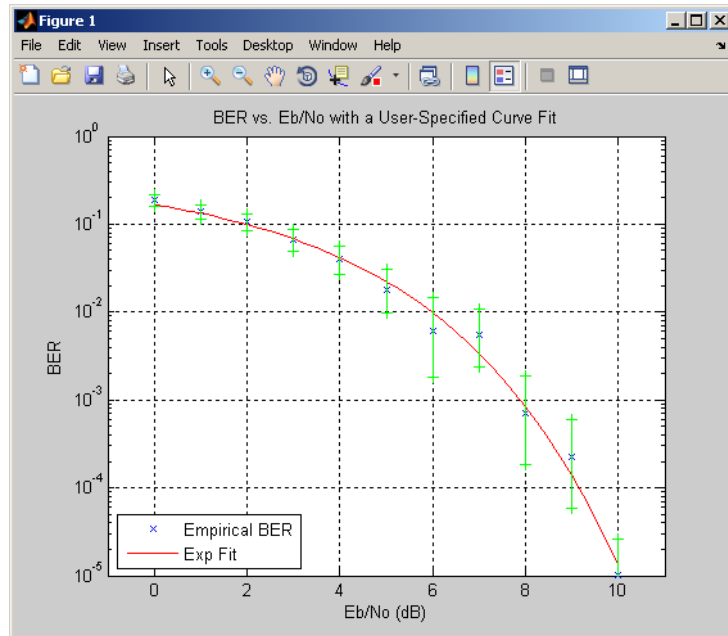
Eye Diagrams

In this section...
"Section Overview" on page 3-20
"EyeScope" on page 3-20

Section Overview

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. The resulting diagram consists of many overlapping curves.

To obtain a more useful eye diagram, create vertical histograms of the input signal. A vertical histogram is defined as the histogram of the amplitude crossings of the input signal at a given time. The eye diagram can be constructed by combining a series of vertical histograms from zero to T seconds, where T is a multiple of the symbol duration.

To produce an eye diagram from a signal, use the `commscope.eyediagram` object. For more information, see the reference page for `commscope.eyediagram`, and the demo, `scattereyedemo`.

EyeScope

Use EyeScope to examine eye diagram results in a user-friendly, graphical environment. EyeScope shows both the eye diagram figure and measurement results in a unified GUI, providing a more efficient means for viewing results.

Scatter Plots

In this section...
"Section Overview" on page 3-21
"Viewing Signals Using Scatter Plots" on page 3-21

Section Overview

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 `commscope.ScatterPlot`.

Scatter plots are often used to visualize the signal constellation associated with digital modulation. For more information, see Plotting Signal Constellations. A scatter plot can be useful when comparing system performance to a published standard, such as 3GPP or DVB standards.

The scatter plot feature is part of the `commscope` package. Users can create the scatter plot object in two ways: using a default object or by defining parameter-value pairs. For more information, see the `commscope.ScatterPlot` help page.

Viewing Signals Using Scatter Plots

In this example, you will observe the received signals for a QPSK modulated system. The output symbols are pulse shaped, using a raised cosine filter.

1 Create a QPSK modulator object. Type the following at the MATLAB command line:

```
hMod = modem.pskmod('M', 4, 'PhaseOffset', pi/4);
```

2 Create an upsampling filter, with an upsample rate of 16. Type the following at the MATLAB command line:

```
Rup = 16;  % up sampling rate
hFilDesign = fdesign.pulseshaping(Rup,'Raised Cosine', ...
```

```
'Nsym,Beta',Rup,0.50);
hFil = design(hFilDesign);
```

3 Create the transmit signal. Type the following at the MATLAB command line:

```
d = randi([0 hMod.M-1], 100, 1);    % Generate data symbols
sym = modulate(hMod, d);            % Generate modulated symbols
xmt = filter(hFil, upsample(sym, Rup));
```

4 Create a scatter plot and set the samples per symbol to the upsampling rate of the signal. Type the following at the MATLAB command line:

```
hScope = commscope.ScatterPlot
hScope.SamplesPerSymbol = Rup;
```

In this simulation, the absolute sampling rate or symbol rate is not specified. Use the default value for `SamplingFrequency`, which is 8000. This results in 2000 symbols per second symbol rate.

5 Set the constellation value of the scatter plot to the expected constellation. Type the following at the MATLAB command line:

```
hScope.Constellation = hMod.Constellation;
```

6 Since the pulse shaping filter introduces a delay, discard these transient values by setting `MeasurementDelay` to the group delay of the filter, which is four symbol durations or 4/Rs seconds. Type the following at the MATLAB command line:

```
groupDelay = (hFilDesign.NumberOfSymbols/2);
hScope.MeasurementDelay = groupDelay /hScope.SymbolRate;
```

7 Update the scatter plot with transmitted signal by typing the following at the MATLAB command line:

```
update(hScope, xmt)
```
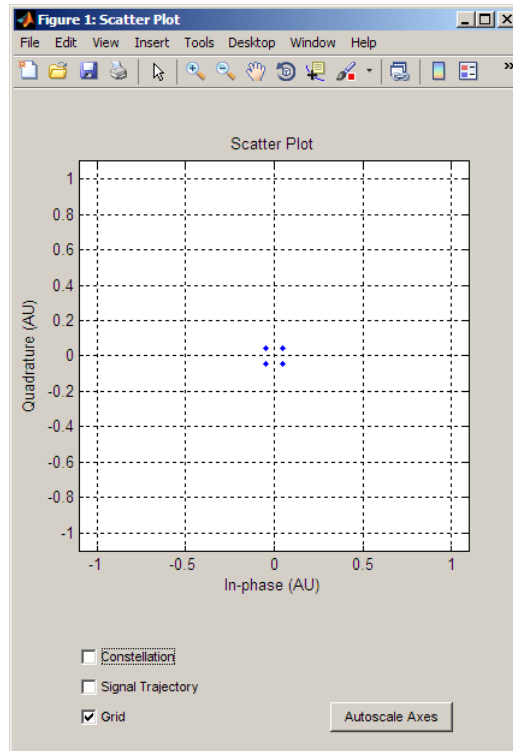
The Figure window updates, displaying the transmitted signal.

8 Display the ideal constellation and evaluate how closely it matches the transmitted signal. To display the ideal constellation, type the following at the MATLAB command line:

```
hScope.PlotSettings.Constellation = 'on';
```
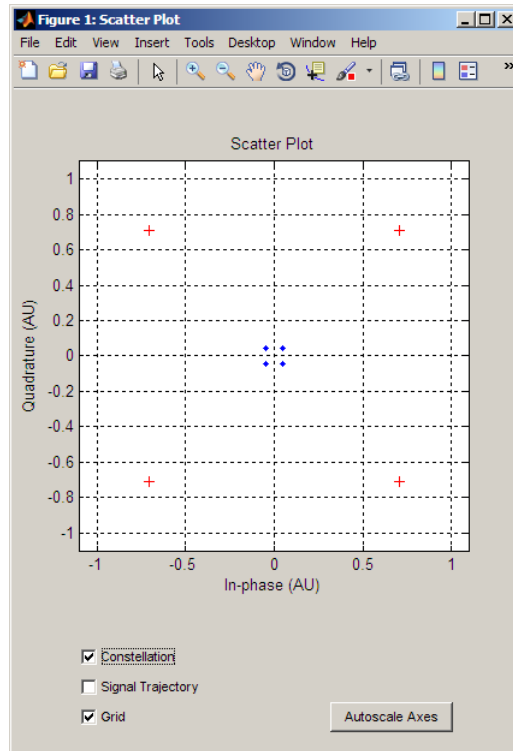
The Figure window updates, displaying the ideal constellation and the transmitted signal.

9 One way to create a better match between the two signals is to normalize the filter. Normalize the filter by typing the following at the MATLAB command line:

```
hFil.Numerator = hFil.Numerator / max(hFil.Numerator);
```

10 Refilter the signal using a normalized filter.

```
xmt = filter(hFil, upsample(sym, Rup));
```

11 Reset the scope before displaying the transmitted signal. Resetting the scope also resets the counter for measurement delay, discarding the transient filter values. To reset the scope, type the following at the MATLAB command line:

```
reset(hScope)
```

12 Update the scatter plot so it displays the signal.

```
update(hScope, xmt)
```
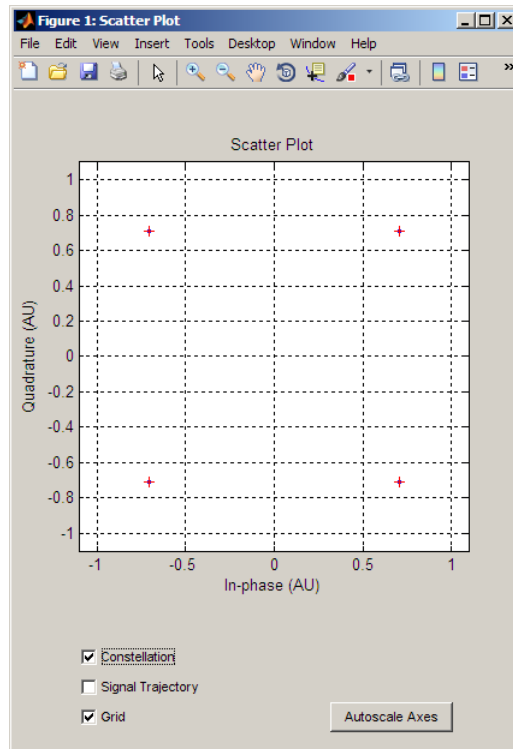

The match between the ideal constellation points and the transmitted signal is nearly identical.

13 To view the transmitted signal more clearly, turn off the ideal constellation by clicking **Constellation** in the Figure window.
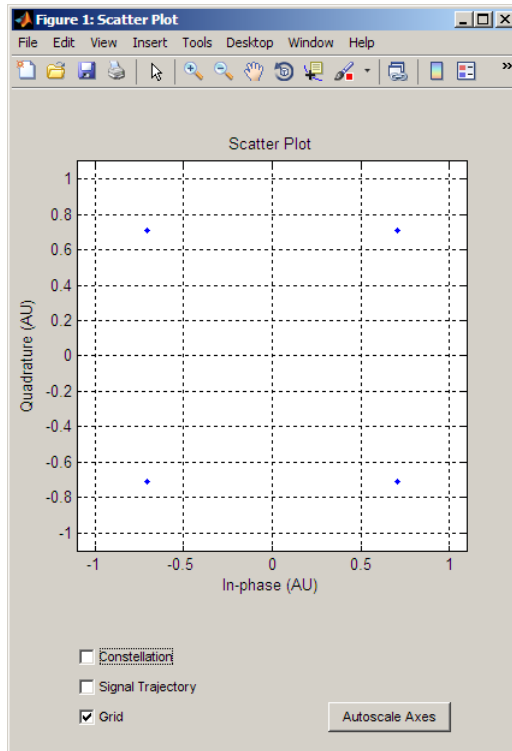
The Figure window updates, displaying only the transmitted signal.

14 View the signal trajectory. Type the following at the MATLAB command line:

```
hScope.PlotSettings.SignalTrajectory = 'on';
```
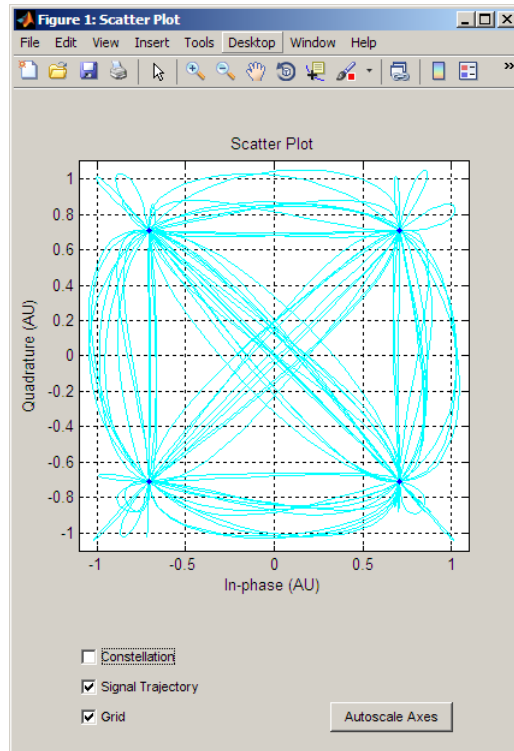
The Figure window updates, displaying the trajectory. An alternate way to display the signal trajectory is to click the **Signal Trajectory**.

15 Change the line style. Type the following at the MATLAB command line:

```
hScope.PlotSettings.SignalTrajectoryStyle = ':m';
```
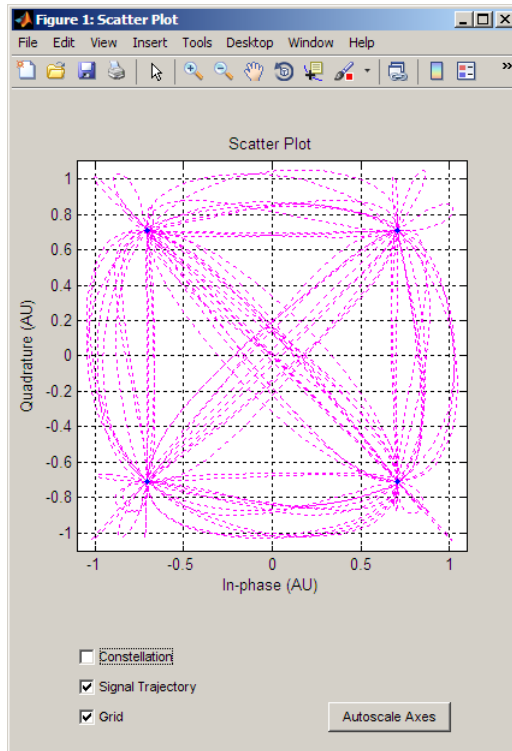
The Figure window updates, changing the line style making up the signal trajectory.

16 Autoscale the scatter plot display to fit the entire plot. Type the following at the MATLAB command line:

```
autoscale(hScope)
```
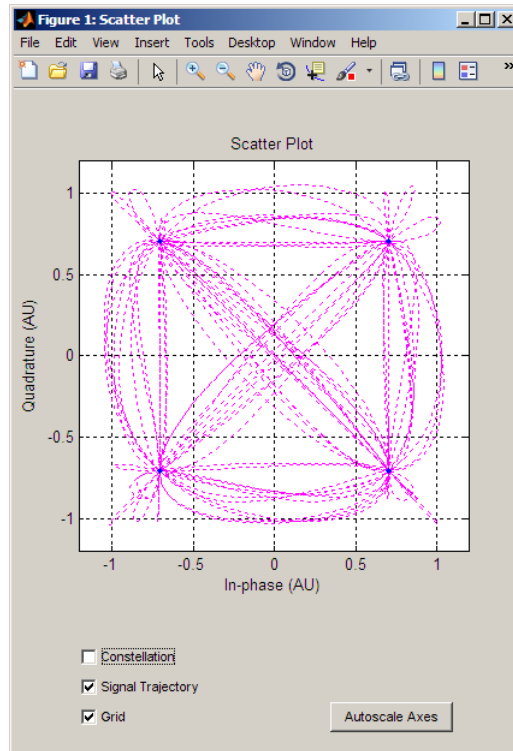
The Figure window updates. An alternate way to autoscale the fit is to click the **Autoscale Axes** button.

17 Create a noisy signal by Passing xmt through an AWGN channel. Type the following at the MATLAB command line:

```
rcv = awgn(xmt, 20, 'measured');    % Add AWGN
```

18 Send the received signal to the scatter plot. Before sending the signal, reset the scatter plot to remove the old data. Type the following at the MATLAB command line:

```
reset(hScope)
update(hScope, rcv)
```
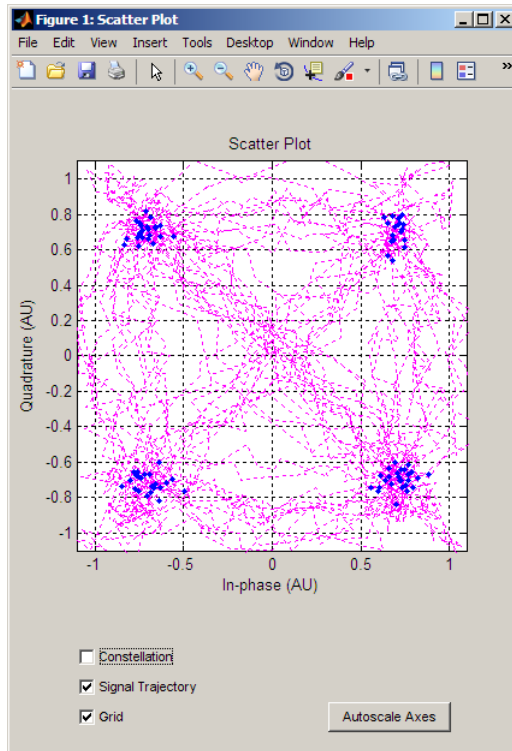
The Figure window updates, displaying the noisy signal.

19 Turn off the signal trajectory by clicking **Signal Trajectory** in the Figure window.
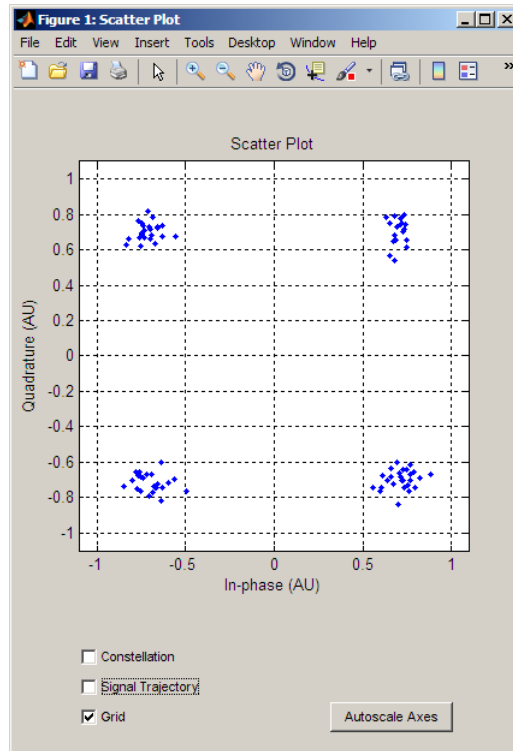
The Figure window updates, displaying the signal plot without the signal trajectory. An alternate way to turn off the signal trajectory is typing the following at the MATLAB command line:

```
hScope.PlotSettings.SignalTrajectory = 'off';
```

20 View the constellation by clicking **Constellation** in the Figure window.
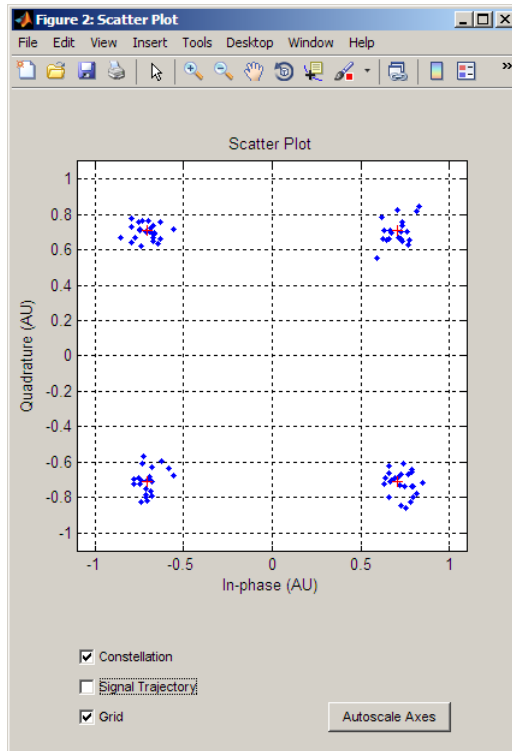
The Figure window updates, displaying both the ideal constellation and the transmitted signal. An alternate way to view the constellation is by typing the following at the MATLAB command line:

```
hScope.PlotSettings.Constellation = 'on';
```

21 Print the scatter plot by making the following selection in the Figure window: **File > Print**
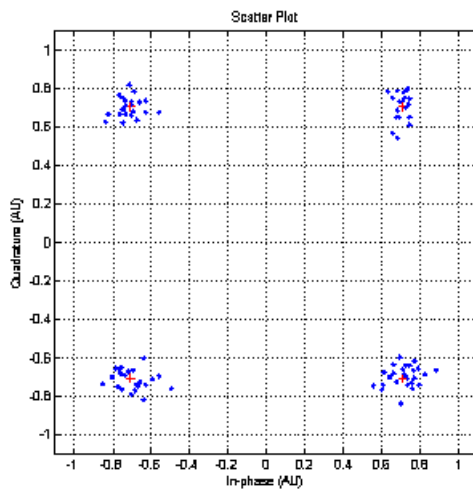
When you print the scatter plot, you print the axes, not the entire GUI.

Adjacent Channel Power Ratio (ACPR) Measurements

Adjacent channel power ratio (ACPR) calculations characterize *spectral regrowth* in a communications system component, such as a modulator or an analog front end. Amplifier nonlinearity causes spectral regrowth. ACPR calculations determine the likelihood that a given system causes interference with an adjacent channel.

Many transmission standards, such as IS-95, CDMA, WCDMA, 802.11, and Bluetooth, contain a definition for ACPR measurements. Most standards define ACPR measurements as the ratio of the average power in the main channel and any adjacent channels. The specific offset frequencies and measurement bandwidths (BWs) you use depends on the specific industry standard you are using. For instance, measurements of CDMA amplifiers involve two offsets (from the carrier frequency) of 885 kHz and 1.98 MHz, and a measurement BW of 30 KHz.

For more information, see the commmeasure.ACPR help page.

Overview of ACPR Measurement Tutorial

In this tutorial, you obtain ACPR measurements using a WCDMA communications signal, according to the 3GPP™ TS 125.104 standard.

This example uses baseband WCDMA sample signals at the input and output of a nonlinear amplifier. The WCDMASignal.mat file contains sample data for use with the tutorial. This file divides the data into 10e3 sample blocks and stores them in the columns of data matrices, dataBeforeAmplifier and dataAfterAmplifier.

The WCDMA specification requires that you obtain all measurements using a 3.84 MHz sampling frequency.

Creating the ACPR Object and Setting Up Measurements

1 Define the sampling frequency, load the WCDMA file, and get the data by typing the following at the MATLAB command line:

```
% System sampling frequency, 3.84 MHz chip rate, 8 samples per chip
```

```
Fs = 3.84e6*8;
load WCDMASignal.mat
txSignalBeforeAmplifier = dataBeforeAmplifier(:,1);
txSignalAfterAmplifier = dataAfterAmplifier(:,1);
```

2 Create the ACPR object and specify the sampling frequency.

```
hACPR = commmeasure.ACPR('Fs',Fs)
```

The object outputs the following:

```
hACPR =

                         Type: 'ACPR Measurement'
          NormalizedFrequency: 0
                           Fs: 30720000
          MainChannelFrequency: 0
             MainChannelMeasBW: 1536000
          AdjacentChannelOffset: [-3072000 3072000]
          AdjacentChannelMeasBW: 1536000
             MeasurementFilter: [1x1 dfilt.dffir]
        SpectralEstimatorOption: 'Default'
      FrequencyResolutionOption: 'Inherit from input dimensions'
                FFTLengthOption: 'Next power of 2'
                        MaxHold: 'Off'
                     PowerUnits: 'dBm'
                     FrameCount: 0
```

3 Specify the *main channel* center frequency and measurement bandwidth.

Specify the main channel center frequency using the
MainChannelFrequency property. Then, specify the main
channel measurement bandwidth using the MainChannelMeasBW
property.

For the baseband data you are using, the main channel center frequency
is at 0 Hz. The WCDMA standard specifies that you obtain main channel
power using a 3.84-MHz measurement bandwidth. Specify these by typing
the following.

```
hACPR.MainChannelFrequency = 0;
```

```
hACPR.MainChannelMeasBW = 3.84e6;
```

4 Specify *adjacent channel* offsets and measurement bandwidths.

The WCDMA standard specifies ACPR limits for four adjacent channels, located at 5, -5, 10, -10 MHz away from the main channel center frequency. In all cases, you obtain adjacent channel power using a 3.84-MHz bandwidth. Specify the adjacent channel offsets and measurement bandwidths using the AdjacentChannelOffset and AdjacentChannelMeasBW properties.

```
hACPR.AdjacentChannelOffset = [-10 -5 5 10]*1e6;
hACPR.AdjacentChannelMeasBW = 3.84e6;
```

Notice that if the measurement bandwidths for all the adjacent channels are equal, you specify a scalar value. If measurement bandwidths are different, you specify a vector of measurement bandwidths with a length equal to the length of the offset vector.

Obtaining the ACPR Measurements

You obtain ACPR measurements by calling the run method of the ACPR object. You can also obtain the power measurements for the main and adjacent channels. The PowerUnits property specifies the unit of measure. The property value defaults to dBm (power ratio referenced to one milliwatt (mW)).

1 Obtain the ACPR measurements at the amplifier input by typing the following at the MATLAB command line:

```
[ACPR mainChannelPower adjChannelPower] = ...,
run(hACPR,txSignalBeforeAmplifier)
```

The ACPR object produces the following input measurement data:

```
ACPR =

  -68.7815  -54.9298  -54.9667  -68.5011

mainChannelPower =
```

```
    29.7412

adjChannelPower =

  -39.0404  -25.1886  -25.2256  -38.7599
```

2 Obtain the ACPR measurements at the amplifier output:

```
[ACPR mainChannelPower adjChannelPower] = ...,
run(hACPR,txSignalAfterAmplifier)
```

The ACPR object produces the following input measurement data:

```
ACPR =

  -42.9373  -28.1873  -27.9864  -43.0322

mainChannelPower =

    41.0451

adjChannelPower =

  -1.8922   12.8579   13.0587   -1.9870
```

Notice the increase in ACPR values at the output of the amplifier. This increase reflects distortion due to amplifier nonlinearity. The WCDMA standard specifies that ACPR values be below -45 dB at +/- 5 MHz offsets, and below -50 dB at +/- 10 MHz offsets. In this example, the signal at the amplifier input meets the specifications while the signal at the amplifier output does not.

Specifying a Measurement Filter
The WCDMA standard specifies that you obtain ACPR measurements using a root-raised-cosine filter. It also states that you measure *both* the main channel power and adjacent channel powers using a matched root-raised-cosine (RRC) filter with a roll-off factor of 0.22. You specify the measurement filter using the MeasurementFilter property. This property value defaults to an all-pass filter with unity gain.

The filter must be an FIR filter, contained in a `dfilt` object, and its response must center at 0 Hz. The ACPR object automatically shifts and applies the filter at each of the specified main and adjacent channel bands. (The power measurement still falls within the bands specified by the MainChannelMeasBW, and AdjacentChannelMeasBW properties.)

The WCDMASignal.mat file contains data that was obtained using a 96 tap filter with a rolloff factor of 0.22.

1 Create the filter (using fdesign.pulseshaping, from the Signal Processing Toolbox software) and obtain measurements by typing the following at the MATLAB command line:

```
PulseShapeFdesign  = fdesign.pulseshaping(8,...,
'Square Root Raised Cosine', 'Nsym,Beta',16,0.22);
hRRCFilter = design(PulseShapeFdesign);
```

2 Set this filter as the measurement filter for the ACPR object.

```
hACPR.MeasurementFilter = hRRCFilter;
```

3 Obtain the ACPR power measurements at the amplifier input.

```
ACPR = run(hACPR,txSignalBeforeAmplifier)
```

The ACPR object produces the following measurement data:

```
ACPR =
  -71.7014  -55.6953  -55.9930  -71.4599
```

Controlling the Power Spectral Estimator

By default, the ACPR object measures power using a Welch power spectral estimator with a Hamming window and zero percent overlap. The object uses a rectangle approximation of the integral for the power spectral density estimates in the measurement bandwidth of interest. If you set SpectralEstimatorOption to 'User defined' several properties become available, providing you control of the resolution, variance, and dynamic range of the spectral estimates.

1 Enable the SegmentLength, OverlapPercentage, and WindowOption properties by typing the following at the MATLAB command line:

```
hACPR.SpectralEstimatorOption = 'User defined'
```

This change allows you to customize the spectral estimates for obtaining power measurements. For example, you can set the spectral estimator segment length to 1024 and increase the overlap percentage to 50%, reducing the consequent variance increase. You can also choose a window with larger side lobe attenuation (compared to the default Hamming window). For more information, see spectrum.welch in the Signal Processing Toolbox™ User's Guide.

2 Create a spectral estimator with a 'Chebyshev' window and a side lobe attenuation of 200 dB.

```
hACPR.SegmentLength = 1024;
hACPR.OverlapPercentage = 50;
% Choosing a Chebyshev window enables a SidelobeAtten property
% you can use to set the side lobe attenuation of the window.
hACPR.WindowOption = 'Chebyshev';
hACPR.SidelobeAtten = 200;
```

3 Obtain the ACPR power measurements at the amplifier output.

```
ACPR = run(hACPR,txSignalAfterAmplifier)
```

The ACPR object produces the following measurement data:

```
ACPR =
  -43.9446  -29.7173  -29.8086  -43.7798
```

Controlling the Resolution of Power Spectral Density Measurements

When the SpectralEstimatorOption property of the ACPR object is 'Default', you control the power spectral measurement resolution using the FrequencyResolutionOption property. Set this property to 'Inherit from input dimensions' to obtain the maximum resolution according to the input data length. Setting the FrequencyResolutionOption property to 'Specify via property' sets the resolution to the value you specified for the

FrequencyResolution property. Reducing the frequency resolution reduces the variance of the power spectral density estimates used to compute average power. The tradeoff is an increase in power leakage from frequency components outside of the measurement bandwidth.

1 Set SpectralEstimatorOption back to default to make the FrequencyResolutionOption relevant, by typing the following at the MATLAB command line:

```
hACPR.SpectralEstimatorOption = 'Default';
hACPR.FrequencyResolutionOption = 'Specify via property'
```

Now that the property is relevant, you can specify a frequency value in Hertz. (The larger this value, the smaller the resolution).

2 Specify 20.4 kHz for the frequency value.

```
hACPR.FrequencyResolution = 20.4e3;
```

3 Measure ACPR for the 50 signal snapshots at the amplifier output and compute the variance of the measurements.

```
for idx = 1:50
    ACPR(idx,:) = run(hACPR,dataAfterAmplifier(:,idx));
end
var(ACPR)
```

The ACPR object produces the following output data:

```
ans =

    0.4733    0.6926    0.7824    0.4596
```

4 For comparison, set the resolution to 650 kHz, measure ACPR for 50 signal snapshots at the amplifier output, and compute the variance of the measurements:

```
hACPR.FrequencyResolution = 650e3;
for idx = 1:50
    ACPR(idx,:) = run(hACPR,dataAfterAmplifier(:,idx));
end
```

```
var(ACPR)
```

The ACPR object produces the following output data:

```
ans =
    0.3298    0.5394    0.5389    0.4168
```

In this example, lowering the resolution (from 20.4 to 650 KHz) also lowers the ACPR measurement variance.

Measure Power Using the Max-Hold Option.

Some communications standards specify using max-hold spectrum power measurements when computing ACPR values. This calculation compares the current power spectral density vector estimation to the previous max-hold accumulated power spectral density vector estimation. The object obtains the power spectral density vector estimation using the current input data. It obtains the previous max-hold accumulated power spectral density vector from the previous call to the run method. The object uses the maximum values at each frequency bin for calculating average power measurements. A call to the reset method clears the max-hold spectrum.

1 Accumulate max-hold spectra for 50 amplifier output data snapshots and get ACPR measurements by typing the following at the MATLAB command line:

```
hACPR.MaxHold = 'On';
for idx = 1:49
    run(hACPR,dataAfterAmplifier(:,idx));
end
ACPR = run(hACPR,dataAfterAmplifier(:,50))
```

The ACPR object produces the following output data:

```
ACPR =

  -42.4644  -26.6522  -27.0012  -42.3082
```

2 The frame count property of the ACPR object reflects the number of signal snapshots the object processes.

```
hACPR.FrameCount
%Reset the ACPR object
reset(hACPR)
hACPR.FrameCount
```

The ACPR object produces the following output data:

```
ans =

    50

ans =

     0
```

Plotting the Signal Spectrum

Use the MATLAB software to plot the power spectral density of the WCDMA signals at the input and output of the nonlinear amplifier. The plot allows you to visualize the spectral regrowth effects intrinsic to amplifier nonlinearity. Notice how the measurements reflect the spectral regrowth. (Note: the following code is just for visualizing signal spectra; it has nothing to do with obtaining the ACPR measurements).

```
hPsd = spectrum.welch('Hamming',1024);
hopts = psdopts(hPsd);
set(hopts,'SpectrumType','twosided','NFFT',1024,'Fs',Fs,...,
'CenterDC',true)
PSD1 = psd(hPsd,txSignalBeforeAmplifier,hopts);
PSD2 = psd(hPsd,txSignalAfterAmplifier,hopts);
data = dspdata.psd([PSD1.Data PSD2.Data],PSD1.Frequencies,'Fs',Fs);
plot(data)
legend('Amplifier input', 'Amplifier output')
```
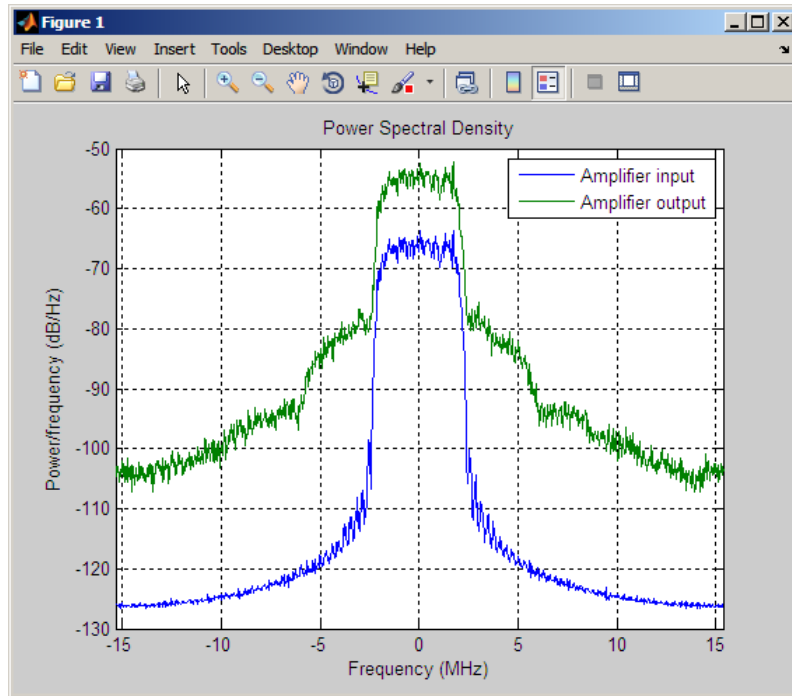
EVM Measurements

Section Overview

Error Vector Magnitude (EVM) is a measurement of modulator or demodulator performance in the presence of impairments. Essentially, EVM is the vector difference at a given time between the ideal (transmitted) signal and the measured (received) signal. If used correctly, these measurements can help in identifying sources of signal degradation, such as: phase noise, I-Q imbalance, amplitude non-linearity and filter distortion

These types of measurements are useful for determining system performance in communications applications. For example, determining if an EDGE system conforms to the 3GPP radio transmission standards requires accurate RMS, EVM, Peak EVM, and 95th percentile for the EVM measurements.

Users can create the EVM object in two ways: using a default object or by defining parameter-value pairs. As defined by the 3GPP standard, the unit of measure for RMS, Maximum, and Percentile EVM measurements is percentile (%). For more information, see the `commmeasure.EVM` help page.

MER Measurements

Section Overview

Communications Toolbox can perform Modulation Error Ratio (MER) measurements. MER is a measure of the signal-to-noise ratio (SNR) in a digital modulation applications. These types of measurements are useful for determining system performance in communications applications. For example, determining if an EDGE system conforms to the 3GPP radio transmission standards requires accurate RMS, EVM, Peak EVM, and 95th percentile for the EVM measurements.

The MER object is part of the commmeasure package. As defined by the DVB standard, the unit of measure for MER is decibels (dB). For consistency, the unit of measure for Minimum MER and Percentile MER measurements is also in decibels. For more information, see the `commmeasure.MER` help page.

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, NJ, 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, NJ, Prentice-Hall, 1977.

4

Error Rate Test Console

Introduction to Error Rate Test Console

The Error Rate Test Console is an object capable of running simulations for communications systems to measure error rate performance.

The ERTC is compatible with communications systems created with a specific API defined by the testconsole.SystemBasicAPI class. Within this class definition you define the functionality of a communications system.

You attach a system to the Error Rate Test Console to run simulations and obtain error rate data.

You obtain error rate results at different locations in the system under test, by defining unique test points. Each test point contains a pair of probes that the system uses to log data to the test console. The information you register with the test console specifies how each pair of test probes compares data. For example, in a frame based system, the Error Rate Test Console can compare transmitted and received header bits or transmitted and received data bits. Similarly, it can compare CRC error counts to obtain frame error rates at different points in the system. You can also configure the Error Rate Test Console to compare data in multiple pairs of probes, obtaining multiple error rate results.

You can run simulations with as many test parameters as desired, parse the results, and obtain parametric or surface plots by specifying which parameters act as independent variables.

There are two main tasks associated with using the Error Rate Test Console: Creating a System and Attaching a System to the Error Rate Test Console.

When you run a system that is not attached to an Error Rate Test Console, the system is running in debug mode. Debug mode is useful when evaluating or debugging the code for the system you are designing.

To see a full-scale example on creating a system and running simulations, see Running Simulations Using the Error Rate Test Console in the Communications Toolbox *Getting Started Guide*.

Creating a System

You attach a system to the Error Rate Test Console to run simulations and obtain error rate data. When you attach the system under test, you also register specific information to the test console in order to define the system's test inputs, test parameters, and test probes.

Creating a communications system for use with the Error Rate Test Console, involves the following steps.

- Writing a system class, extending the testconsole.SystemBasicAPI class.
- Writing a registration method
  - Registration is test related
  - Defines items such as test parameters, test probes, and test inputs
- Writing a setup method
- Writing a reset method
- Writing a run method

  Methods allows the system to communicate with the test console.

You can see an example of a system file by opening MPSKSYSTEM.m, which resides in the following location:

matlab\toolbox\comm\comm\+commtest

Writing A Register Method

Using the `register` method, you register test inputs, test parameters, and test probes to the Error Rate Test Console. You register these items to the Error Rate Test Console using the `registerTestInput`, `registerTestParameter`, and `registerTestProbe` methods.

- Write a `register` method for every communication system you create.
- If you do not implement a `register` method for a system, you can still attach the system to the Error Rate Test Console. While the test console runs the specified number of iterations on the system, you cannot control simulation parameters or retrieve results from the simulation.

Registering Test Inputs

In order to run simulations, the system under test requests test inputs from the Error Rate Test Console. These test inputs provide data, driving simulations for the system under test.

A system under test cannot request a specific input type until you attach it to the Error Rate Test Console. Additionally, the specific input type must be registered to the test console.

Inside the `register` method, you call the `registerTestInput(sys,inputName)` method to register test inputs.

- `sys` represents the handle to a user-defined system object.

- `inputName` represents the name of the input that the system registers. This name must coincide with the name of an available test input in the Error Rate Test Console or an error occurs.

  - 'NumTransmissions' - calling the `getInput` method returns the frame length. The system itself is responsible for generating a data frame using a data source.

  - 'RandomIntegerSource' - calling the `getInput` method returns a vector of symbols with a length the Error Rate Test Console `FrameLength` property specifies. If the system registers this source type, then it must also register a test parameter named 'M' that corresponds to the modulation order.

Registering Test Parameters

Test parameters are the system parameters for which the Error Rate Test Console obtains simulation results. You specify the sweep range of these parameters using the Error Rate Test Console and obtain simulation results for different system conditions.

The system under test registers a system parameter to the Error Rate Test Console, creating a test parameter. You register a test parameter to the Error Rate Test Console using the `registerTestParameter(sys,name,default,validRange)` method.

- `sys` represents the handle to the user-defined system object

- `name` represents the parameter name that the system registers to the Error Rate Test Console

- `default` specifies the default value of the test parameter – it can be a numeric value or a string

- `validRange` specifies a range of input values for the test parameter — it can be a 1x2 vector of numeric values with upper and lower ranges or a cell array of chars (an Enum).

A parameter of type char becomes useful when defining system conditions. For example, in a communications system, a Channel parameter may be defined so that it takes values such as 'Rayleigh', 'Rician', or 'AWGN'. Depending on the Channel char value, the system may filter transmitted data through a different channel. This allows the simulation of the system over different channel scenarios.

If the system registers a test parameter named 'X' then the system must also contain a readable property named 'X'. If not, the registration process issues an error. This process ensures that calling the `getTestParameter` method in debug mode returns the value held by the corresponding property.

Registering Test Probes

Test probes log the simulation data the Error Rate Test Console uses for computing test metrics, such as: number of errors, number of transmissions, and error rate. To log data into a probe, your communications system must register the probe to the Error Rate Test Console.

You register a test probe to the Error Rate Test Console using the `registerTestProbe(sys,name)` method.

- `sys` represents the handle to the user-defined system object

- `name` represents the name of the test probe

You can define an arbitrary number of probes to log test data at several points within the system.

Writing a Setup Method

The Error Rate Test Console calls the `setup` method at the beginning of simulations for each new sweep point. A sweep point is one of several sets of simulation parameters for which the system will be simulated. Using the `getTestParameter` method of the system under test, the `setup` method requests the current simulation sweep values from the Error Rate Test Console and sets the various system components accordingly. The `setup` method sets the system to the conditions the current test parameter sweep values generate.

Writing a `setup` method for each communication system you create is not necessary. The `setup` method is optional.

Writing a Reset Method

Use the `reset` method to reset states of various system components, such as: objects, data buffers, or system flags. The Error Rate Test Console calls the `reset` method of the system:

- at the beginning of simulations for a new sweep point. (This condition occurs when you set the `ResetMode` of the Error Rate Test Console to "Reset at new simulation point'.
- at each simulation iteration. (This condition occurs when you set the `ResetMode` of the Error Rate Test Console to 'Reset at every iteration'.)

Writing a `reset` method for each communication system you create is not mandatory. The `reset` method is optional.

Writing a Run Method

Write a `run` method for each communication system you create. The `run` method includes the core functionality of the system under test. At each simulation iteration, the Error Rate Test Console calls the `run` method of the system under test.

When designing a communication system, ensure at run time that your system sets components to the current simulation test parameter sweep values. Depending on your unique design, at run time, the communication system:

- requests test inputs from the test console using the `getInput` method

- logs test data to its test probes using the `setTestProbeData` method

- logs user-data to the test console using the `setUserData` method

- Although it is recommended you do this at setup time, the system can also request the current simulation sweep values using the `getTestParameter` method.

Methods Allowing You to Communicate with the Error Rate Test Console at Simulation Run Time

In this section...
"Getting Test Inputs From the Error Rate Test Console" on page 4-8
"Getting the Current Simulation Sweep Value of a Registered Test Parameter" on page 4-9
"Logging Test Data to a Registered Test Probe" on page 4-9
"Logging User-Defined Data To The Test Console" on page 4-9

Getting Test Inputs From the Error Rate Test Console

At simulation time, the communications system you design can request input data to the Error Rate Test Console. To request a particular type of input data, the system under test must register the specific input type to the Error Rate Test Console. The system under test calls getInput(obj,inputName) method to request test inputs to the test console.

- obj represents the handle of the Error Rate Test Console

- inputName represents the input that the system under test gets from the Error Rate Test Console

For an Error Rate Test Console, 'NumTransmissions' or 'RandomDiscreetSource' are acceptable selections for `inputName`.

The system under test provides the following inputs:

- 'NumTransmissions' - calling the `getInput` method returns the frame length. The system itself is responsible for generating a data frame using a data source.

- 'RandomIntegerSource' - calling the `getInput` method returns a vector of symbols with a length the Error Rate Test Console `FrameLength` property specifies. If the system registers this source type, then it must also register a test parameter named 'M' that corresponds to the modulation order.

Getting the Current Simulation Sweep Value of a Registered Test Parameter

For each simulation iteration, the system under test may require the current simulation sweep values from the registered test parameters. To obtain these values from the Error Rate Test console, the system under test calls the `getTestParameter(sys,name)` method.

Logging Test Data to a Registered Test Probe

At simulation time, the system under test may log data to a registered test probe using the `setTestProbeData(sys,name,data)` method.

- `sys` represents the handle to the system

- `name` represents the name of a registered test probe

- `data` represents the data the probe logs to the Error Rate Test Console.

Logging User-Defined Data To The Test Console

At simulation time, the system under test may log user-data to the Error Rate Test Console by calling the `setUserData` method. This user-data passes directly to the specific user-defined metric calculator functions. Log user-data to the Error Rate Test Console as follows:

```
setUserData(sys,data)
```

- `sys` represents the handle to the system

- `data` represents the data the probe logs to the Error Rate Test Console.

Debug Mode

When you run a system that is not attached to an Error Rate Test Console, the system is running in debug mode. Debug mode is useful when evaluating or debugging the code for the system you are designing.

A system that extends the testconsole.SystemBasicAPI class can run by itself, without the need to attach it to a test console. This scenario is referred to as debug mode. Debug mode is useful when evaluating or debugging the code for the system you are designing. For example, if you define break points when designing your system, you can run the system in debug mode and confirm that the system runs without errors or warnings.

Implementing A Default Input Generator Function For Debug Mode

If your system registers a test input and calls the `getInput` method at simulation run time then for it to run in debug mode, the system must implement a default input generator function. This method should return an input congruent to the test console.

```
input = generateDefaultInput(obj)
```

Attaching a System to the Error Rate Test Console and Running Simulations

In this section...
"Attaching a System to the Error Rate Test Console" on page 4-11
"Defining Simulation Conditions" on page 4-12
"Registering a Test Point" on page 4-14
"Getting Test Information" on page 4-15
"Running a simulation" on page 4-16
"Getting Results and Plotting Data" on page 4-16

Running simulations with the Error Rate Test Console involves the following tasks:

- Attaching a system
- Defining simulation conditions
  - Specifying stop criterion
  - Specifying iteration mode
  - Specifying reset mode
  - Specifying sweep values
- Registering test points
- Running simulations
- Getting results and plotting

Attaching a System to the Error Rate Test Console

You attach a system to the Error Rate Test Console to run simulations and obtain error rate data. There are two ways to attach a system to the Error Rate Test Console.

- To attach a system to the Error Rate Test Console, type the following at the MATLAB command line:

```
attachSystem(testConsole, mySystem)
```

- To attach a system at construction time of an Error Rate Test Console, use the following:

```
testconsole = commtest.ErrorRate(mySystem)
```

- `mySystem` is the name of the system under test

If system under test A is currently attached to the Error Rate Test Console H1, and you call `attachSystem(H2,A)`, then A detaches from H1 and attaches to Error Rate Test Console H2. This throws a warning stating that the system has detached from H1 to be attached to H2.

Defining Simulation Conditions

Stop Criterion

The Error Rate Test Console controls the simulation stop criterion using the `SimulationLimitOption` property. You define the criterion to stop a simulation when reaching either a specific number of transmissions or a specific number of errors.

- Setting `SimulationLimitOption` property to 'Number of transmissions' stops the simulation for each sweep parameter point when the Error Rate Test Console counts the number of transmissions specified in `MaxNumTransmissions`

- Setting `SimulationLimitOption` property to 'Number of errors' stops the simulation for a sweep parameter point when the Error Rate Test Consols counts the number of errors specified in `MinNumErrors`. The `ErrorCountTestPoint` property should be set to the name of the registered test point containing the error count being compared to the `MinNumErrors` property to control the simulation length.

- Setting `SimulationLimitOption` property to 'Number of errors or transmissions' stops the simulation for each sweep parameter point when the Error Rate Test Console completes the number of transmissions specified in `MaxNumTransmissions` or when obtaining the number of errors specified in `MinNumErrors`, whichever happens first.

Iteration Mode

The iteration mode defines the way that the Error Rate Test Console combines test parameter sweep values to perform simulations. The `IterationMode` property of the test console controls this behavior.

- Setting `IterationMode` to 'Combinatorial' performs simulations for all possible combinations of registered test parameter sweep values.

- Setting `IterationMode` to 'Indexed' performs simulations for all indexed sweep value sets. The ith sweep value set consists of the ith element from every sweep value vector for each registered test parameter. All sweep value vectors must be of equal length, with the exception of those that are unit length.

Specifying and Obtaining Sweep Values

The Error Rate Test Console performs simulations for a set of sweep points, which consist of combinations of sweep values specified for each registered test parameter. The was the test console forms sweep points depends on the `IterationMode` settings. The iteration mode defines the way in which sweep values for different test parameters combine to produce simulation results.

Using the `setTestParameterSweepValues` method, you specify sweep values for each test parameter that the system under test registers to the Error Rate Test Console.

```
setTestParameterSweepValues(obj,name,value)
```

where

- `obj` represents handle to the Error Rate Test Console.

- `name` represents the name of the registered test parameter (this name must correspond to a test parameter registered by the system under test or an error occurs)

- `value` represents the sweep values you specify for the test parameter named 'name'. Depending on the application, sweep values may be a vector with numeric values or a cell array of characters. The test console issues an error if you attempt to set sweep values that are out of the specified

valid range for a test parameter (valid ranges are defined by the system when attaching to a test console).

You obtain the list of test parameters registered by the system under test using the `info` method of the Error Rate Test Console.

You obtain the sweep values for a specific registered test parameter using the `getTestParameterSweepValues` method of the Error Rate Test Console. You obtain the valid ranges of a specific registered test parameter using the `getTestParameterValidRanges` method of the Error Rate Test Console.

If you do not specify sweep values for a particular test parameter, the Error Rate Test Console. always uses the parameter's default value to run simulations. (Default values for test parameters are defined by the system when attaching to a test console at registration time.)

Reset Mode

You control the reset criteria for the system under test using the `SystemResetMode` property of the Error Rate Test Console.

- Setting `SystemResetMode` to 'Reset at new simulation point' resets the system under test resets at the beginning of iterations for a new simulation sweep point.

- Setting `SystemResetMode` to 'Reset at every iteration' resets the system under test at every simulation.

Registering a Test Point

You obtain error rate results at different points in the system under test, by defining unique test points. Each test point groups a pair of probes that the system under test uses to log data and the Error Rate Test Console uses to obtain data. In order to create a test point for a pair of probes, the probes must be registered to the Error Rate Test Console.

The Error Rate Test Console calculates error rates by comparing the data available in a pair of probes.

Test points hold error and transmission counts for each sweep point simulation.

The info method displays which test points are registered to the test console.

registerTestPoint(h, name , actprobe,expprobe) registers a new test point with name, name, to the error rate test console, h.

The test point must contain a pair of registered test probes actprobe and expprobe whose data will be compared to obtain error rate values. actprobe contains actual data, and ,expprobe contains expected data. Error rates will be calculated using a default error rate calculator function that simply performs one-to-one comparisons of the data vectors available in the probes.

registerTestPoint(h, name, actprobe,expprobe, handle) adds a handle, handle, that points to a user-defined error calculator function that will be used in stead of the default function to compare the data in probes actprobe and exprobe, to obtain error rate results.

Writing a user-defined error calculator function

A user-defined error calculator function must comply with the following syntax:

[ecnt tcnt] = functionName(act, exp, udata) where ecnt output corresponds to the error count, and TCNT output is the number of transmissions used to obtain the error count. Inputs act, and exp correspond to actual and expected data. The error rate test console will set these inputs to the data available in the pair of test point probes actprobe and expprobe previously mentioned. udata is a user data input that the system under test may pass to the test console at run time using the setUserData method. udata may contain data necessary to compute errors such as delays, data buffers, and so on. The error rate test console will pass the same user data logged by the system under test to the error calculator functions of all the registered test points.

You call the info method to see the names of the registered test points and the error rate calculator functions associated with them, and to see the names of the registered test probes.

Getting Test Information

Returns a report of the current test console settings.

info(h) displays:

- Test console name
- System under test name
- Available test inputs
- Registered test inputs
- Registered test parameters
- Registered test probes
- Registered test points
- Metric calculator functions
- Test metrics

Running a simulation

You run simulations by calling the `run` method of the Error Rate Test Console.

`run(testConsole)` runs a specified number of iterations of an attached system under test for a specified set of parameter values. If a Parallel Computing Toolbox license is available and a matlabpool is open, then the iterations will be distributed among the available number of workers.

Getting Results and Plotting Data

Call the `getResults` method of the error rate test console to obtain test results.

r = getResults(testConsole) returns the simulation results, r, for the test console, h. r is an object of type testconsole.Results and contains all the simulation data for all the registered test points.

You call the getData method of results.object r to get simulation results data. You call the plot and semilogy method of the results object r to plot results data. See `testconsole.Results` for more information.

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 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), convert the SER to a BER before using the simulation with BERTool.

Opening BERTool

To open BERTool, type

```
bertool
```
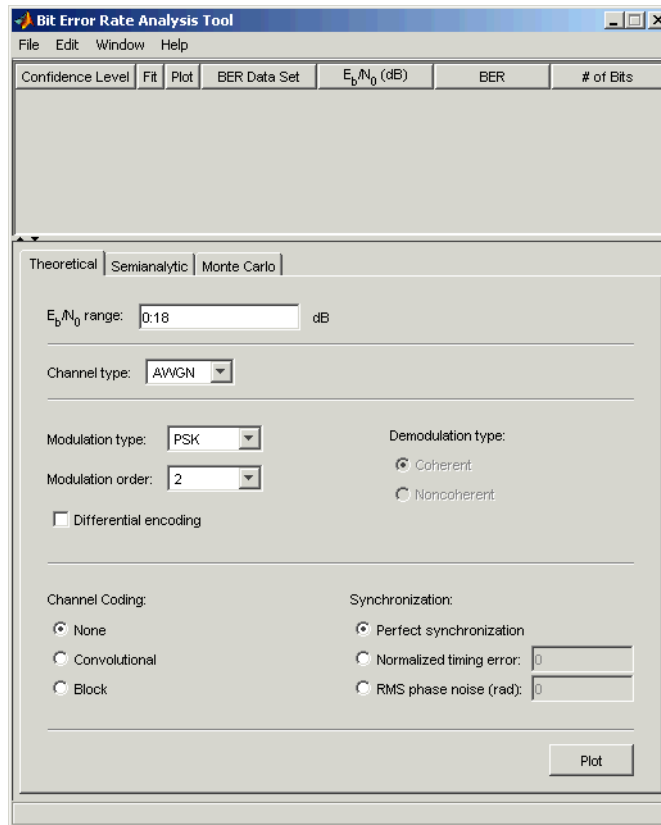
The BERTool Environment

Components of BERTool

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

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits

  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 5-22.

- A set of tabs on the bottom. Labeled **Theoretical**, **Semianalytic**, and **Monte Carlo**, the tabs 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 5-8

- "Using the Semianalytic Technique to Compute BERs" on page 5-16

- "Running MATLAB Simulations" on page 5-22 or "Running Simulink Simulations" on page 5-37

• 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 Tab in BERTool" on page 5-9.

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, BERTool reconfigures the tabs 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, BERTool reconfigures the tabs to reflect the parameters associated with that data and also highlights the corresponding data set in the data viewer.

Note You cannot click on a data point while BERTool is generating Monte Carlo simulation results. You must wait until the tool generates all data points before clicking for more information.

• If you configure the **Semianalytic** or **Theoretical** tab in a way that is already reflected in an existing data set, 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, 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 Tab in BERTool" on page 5-9.

Note If you save the BER Figure window using the window's **File** menu,
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 5-55.

Computing Theoretical BERs

In this section...

"Section Overview" on page 5-8

"Example: Using the Theoretical Tab in BERTool" on page 5-9

"Available Sets of Theoretical BER Data" on page 5-11

Section Overview

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 the following procedure:

1 Open BERTool, and go to the **Theoretical** tab.

2 Set the 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 5-11 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 Tab in BERTool"
on page 5-9.

Also, "Available Sets of Theoretical BER Data" on page 5-11 indicates which
combinations of parameters are available on the **Theoretical** tab and which
underlying functions perform computations.

Example: Using the Theoretical Tab 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** tab.

2 Set the parameters as shown in the following figure.

3 Click **Plot**.

BERTool creates an entry in the data viewer and plots the data in the BER Figure window. Even though the parameters request that E_b/N_0 go up to 18, BERTool plots only those BER values that are at least 10^{-8}. The following figures illustrate this step.

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, as shown in the following figures.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	theoretical-exact0	0:18	[0.0786 0.0...	
		☑	theoretical-exact1	0:18	[0.1409 0.1...	
		☑	theoretical-exact2	0:18	[0.1998 0.1...	
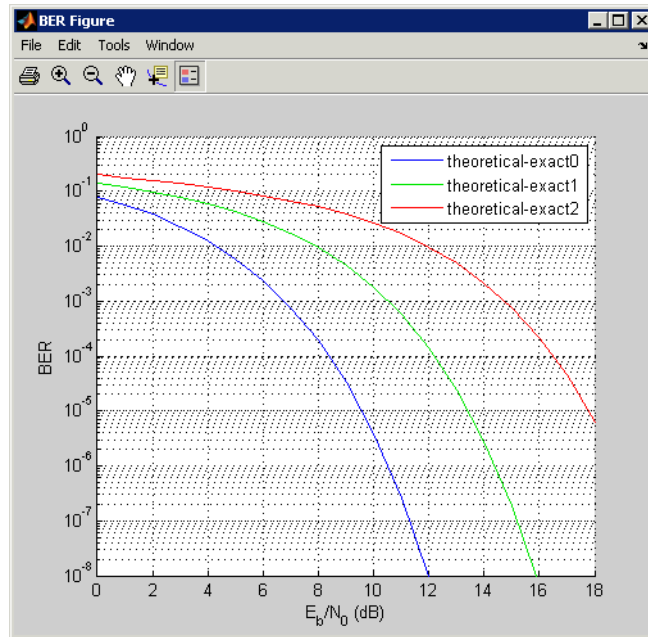
6 To recall which value of **Modulation order** corresponds to a given curve, click the curve. BERTool responds by adjusting the parameters in the **Theoretical** tab to reflect the values that correspond to that curve.

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, select the check box again.

Available Sets of Theoretical BER Data

BERTool can generate a large set of theoretical bit-error rates, but not all combinations of parameters are currently supported. The **Theoretical** tab adjusts itself to your choices, so that the combination of parameters is always valid. 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.

BERTool assumes that Gray coding is used for all modulations.

For QAM, when $\log_2 M$ is odd (*M* being the modulation order), a rectangular constellation is assumed.

Combinations of Parameters for AWGN Channel Systems

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

Modulation	Modulation Order	Other Choices
PSK	2, 4	Differential or nondifferential encoding.
	8, 16, 32, 64, or a higher power of 2	
OQPSK	4	Differential or nondifferential encoding.
DPSK	2, 4, 8, 16, 32, 64, or a higher power of 2	
PAM	2, 4, 8, 16, 32, 64, or a higher power of 2	
QAM	4, 8, 16, 32, 64, 128, 256, 512, 1024, or a higher power of 2	
FSK	2	Orthogonal or nonorthogonal; `Coherent` or `Noncoherent` demodulation.
	4, 8, 16, 32, or a higher power of 2	Orthogonal; `Coherent` demodulation.
	4, 8, 16, 32, or 64	Orthogonal; `Noncoherent` demodulation.

Modulation	Modulation Order	Other Choices
MSK	2	`Coherent` conventional or precoded MSK; `Noncoherent` precoded MSK.
CPFSK	2, 4, 8, 16, or a higher power of 2	`Modulation index` > 0.

BER results are also available for the following:

- block and convolutional coding with hard-decision decoding for all modulations except CPFSK

- block coding with soft-decision decoding for all binary modulations (including 4-PSK and 4-QAM) except CPFSK, noncoherent non-orthogonal FSK, and noncoherent MSK

- convolutional coding with soft-decision decoding for all binary modulations (including 4-PSK and 4-QAM) except CPFSK

- uncoded nondifferentially-encoded 2-PSK with synchronization errors

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

- `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 and Rician Channel Systems

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

When diversity is used, the SNR on each diversity branch is derived from the SNR at the input of the channel (`EbNo`) divided by the diversity order.

Modulation	Modulation Order	Other Choices
PSK	2	Differential or nondifferential encoding Diversity order ≥ 1 In the case of nondifferential encoding, diversity order being 1, and Rician fading, a value for RMS phase noise (in radians) can be specified.
	4, 8, 16, 32, 64, or a higher power of 2	Diversity order ≥ 1
OQPSK	4	Diversity order ≥ 1
DPSK	2, 4, 8, 16, 32, 64, or a higher power of 2	Diversity order ≥ 1
PAM	2, 4, 8, 16, 32, 64, or a higher power of 2	Diversity order ≥ 1
QAM	4, 8, 16, 32, 64, 128, 256, 512, 1024, or a higher power of 2	Diversity order ≥ 1
FSK	2	Correlation coefficient $\in [-1,1]$. Coherent or Noncoherent demodulation Diversity order ≥ 1 In the case of a nonzero correlation coefficient and noncoherent demodulation, the diversity order is 1 only.
	4, 8, 16, 32, or a higher power of 2	Noncoherent demodulation only. Diversity order ≥ 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.

Using the Semianalytic Technique to Compute BERs

In this section...
"Section Overview" on page 5-16
"Example: Using the Semianalytic Tab in BERTool" on page 5-17
"Procedure for Using the Semianalytic Tab in BERTool" on page 5-19
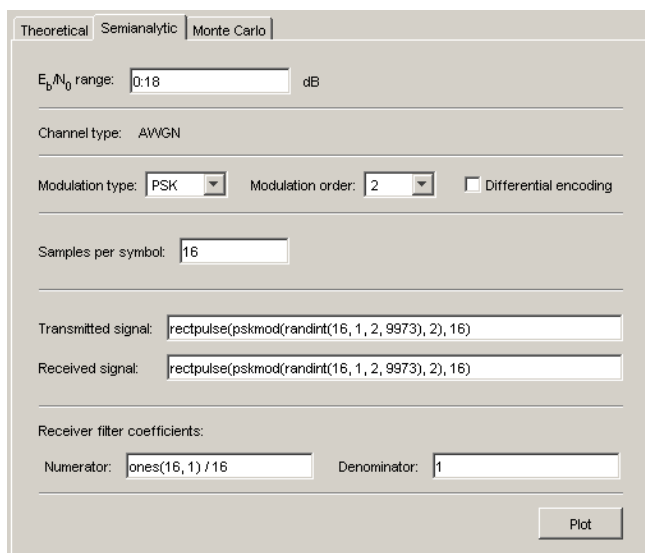
Section Overview

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** tab.

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 Tab 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, but it is 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(1i*pi/180); % Static phase offset of 1 degree
```

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

3 Set parameters as shown in the following figure.

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.
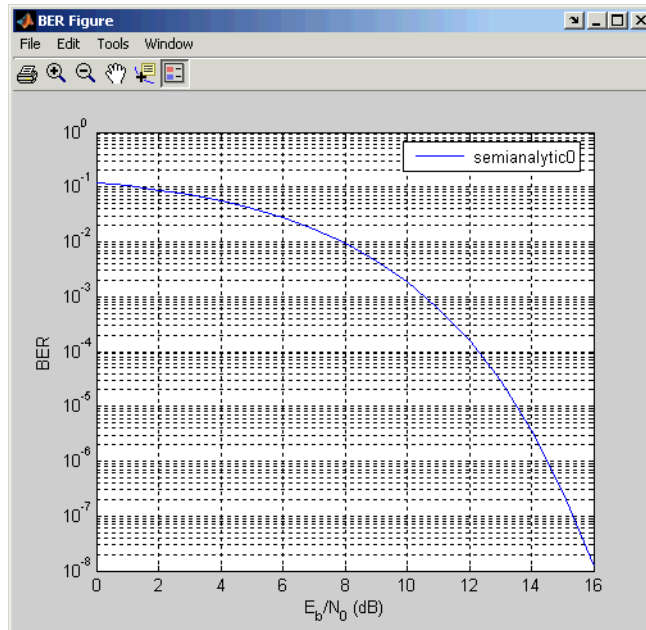
Procedure for Using the Semianalytic Tab in BERTool

The procedure below describes how you 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. Shape the resultant signal with rectangular pulse shaping, using the oversampling factor that you will later use to filter the modulated signal. Store the result of this step as txsig for later use.

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. If you use a square-root raised cosine filter, use it on the nonoversampled modulated signal and specify the oversampling factor in the filtering function. If you use another filter type, you can apply it to the rectangularly pulse shaped signal.

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** tab 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 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

Parameter Name	Meaning
Numerator	Coefficients of the receiver filter that BERTool
Denominator	applies to the received signal

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, 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, 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

In this section...
"Section Overview" on page 5-22
"Example: Using a MATLAB Simulation with BERTool" on page 5-22
"Varying the Stopping Criteria" on page 5-25
"Plotting Confidence Intervals" on page 5-26
"Fitting BER Points to a Curve" on page 5-28

Section Overview

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.

To learn how to make your own simulation functions compatible with BERTool, see "Preparing Simulation Functions for Use with BERTool" on page 5-29.

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 Communications Toolbox software.

To run this example, follow these steps:

1 Open BERTool and go to the **Monte Carlo** tab. (The default parameters depend on whether you have Communications Blockset™ software installed. Also note that the **BER variable name** field applies only to Simulink models.)

2 Set parameters as shown in the following figure.

3 Click **Run**.

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

Then BERTool creates a 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 ...

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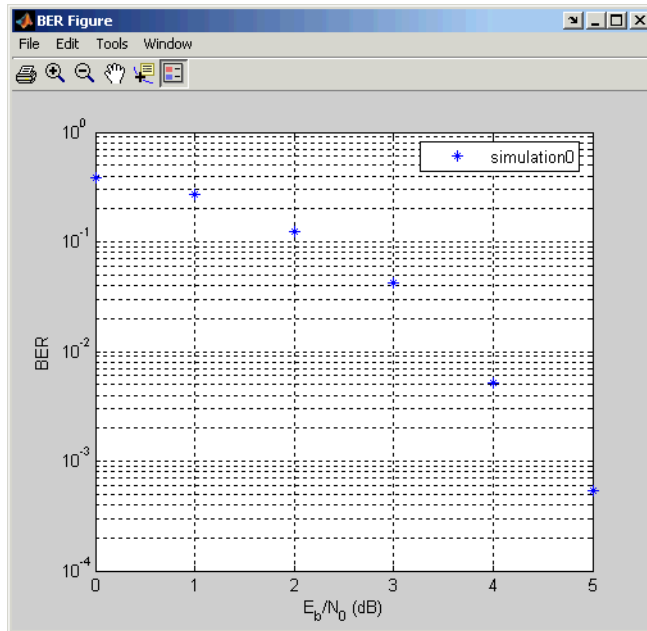
4 To change the range of E_b/N_0 while reducing the number of bits processed in each case, type [5 5.2 5.3] in the **Eb/No range** field, type 1e5 in the **Number of bits** field, and click **Run**.

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.
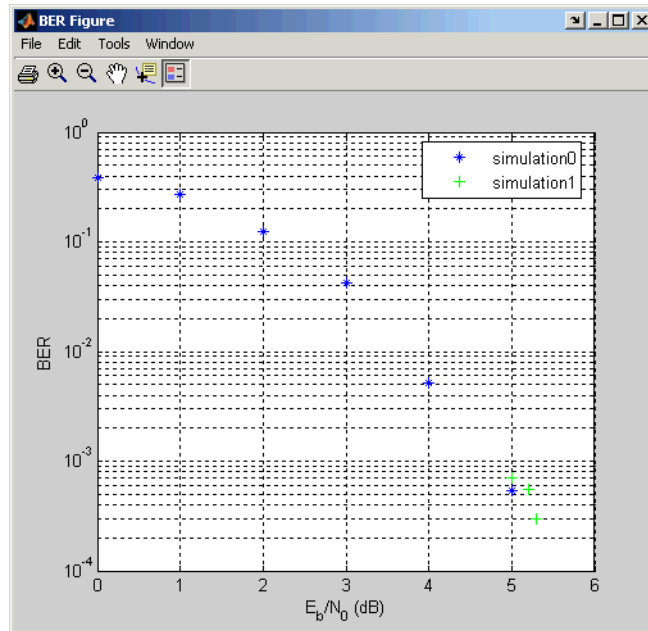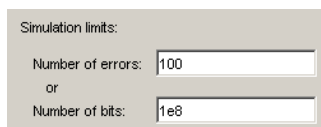
The two points corresponding to 5 dB from the two data sets are different because the smaller value of **Number of bits** in the second simulation caused the simulation to end before observing many errors. To learn more about the criteria that BERTool uses for ending simulations, see "Varying the Stopping Criteria" on page 5-25.

For another example that uses BERTool to run a MATLAB simulation function, see "Example: Preparing a Simulation Function for Use with BERTool" on page 5-33.

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 5-29; for an example, see "Example: Preparing a Simulation Function for Use with BERTool" on page 5-33.

After creating your function, set the target number of errors and the maximum number of bits in the **Monte Carlo** tab 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, 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 5-22 and set **Confidence Level** to 95 for each of the two data sets. 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%.
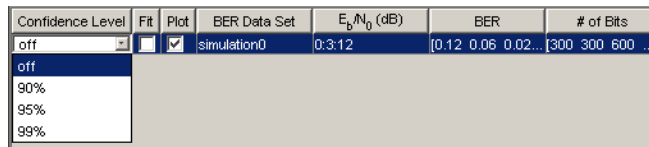
The plot in the BER Figure window responds immediately to your choice. A sample plot is below.
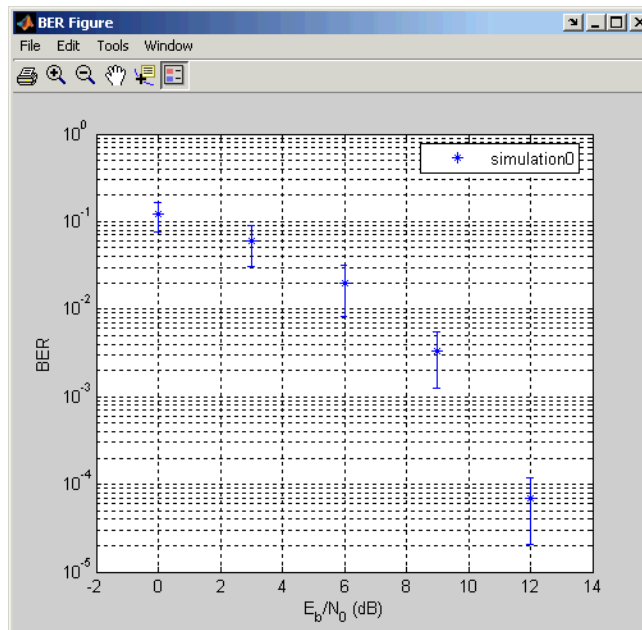
For an example that plots confidence intervals for a Simulink simulation, see "Example: Using a Simulink Model with BERTool" on page 5-38.

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, select 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.
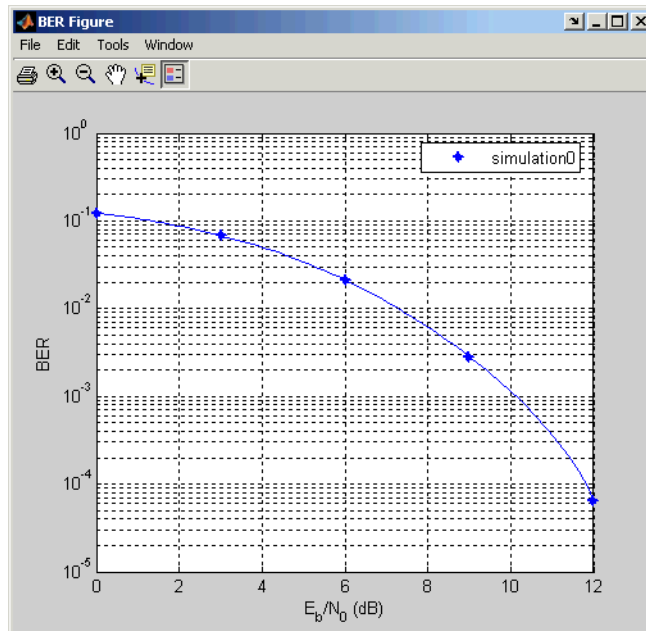
For an example that performs curve fitting for data from a Simulink simulation and generates the plot shown above, see "Example: Using a Simulink Model with BERTool" on page 5-38. For an example that performs curve fitting for data from a MATLAB simulation function, see "Example: Preparing a Simulation Function for Use with BERTool" on page 5-33.

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

Preparing Simulation Functions for Use with BERTool

In this section...
"Requirements for Functions" on page 5-29
"Template for a Simulation Function" on page 5-30
"Example: Preparing a Simulation Function for Use with BERTool" on page 5-33

Requirements for Functions

When you create a MATLAB function for use with BERTool, ensure 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 **Stop** in BERTool and abort the simulation in that case.

Template for a Simulation Function

Use the following template 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 5-31 explains the template's key sections, while "Using the Template" on page 5-32 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 it satisfies the requirements described in "Requirements for Functions" on page 5-29.

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 5-33.

```
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, observe how the function either satisfies the requirements listed in "Requirements for Functions" on page 5-29 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, clicking **Stop** is the only way to terminate the loop.

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

Using the Template

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

1 Determine the setup tasks 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, click **Stop** 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`, you must change one of the names so they match. As another example, if your original code uses SNR instead of E_b/N_0, 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 5-30 to use simulation code from the documentation example in "Example: Curve Fitting for an Error Rate Plot" on page 3-15.

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 5-30 into a new M-file in the MATLAB Editor. Save it in a folder on your MATLAB path using the file name `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.
hMod = modem.dpskmod('M', M); % Create a DPSK modulator
hDemod = modem.dpskdemod(hMod); % Create a DPSK
                    % demodulator using the modulator object
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 = modulate(hMod, msg); % Modulate.
rxsig = awgn(txsig, snr, 'measured'); % Add noise.
decodmsg = demodulate(hDemod, rxsig); % 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 to 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** tab.

2 Set parameters on the **Monte Carlo** tab as shown in the following figure.

3 Click **Run**.

BERTool spends some time computing results and then plots them. They do not appear to fall along a smooth curve because the simulation required only five errors for each value in EbNo.
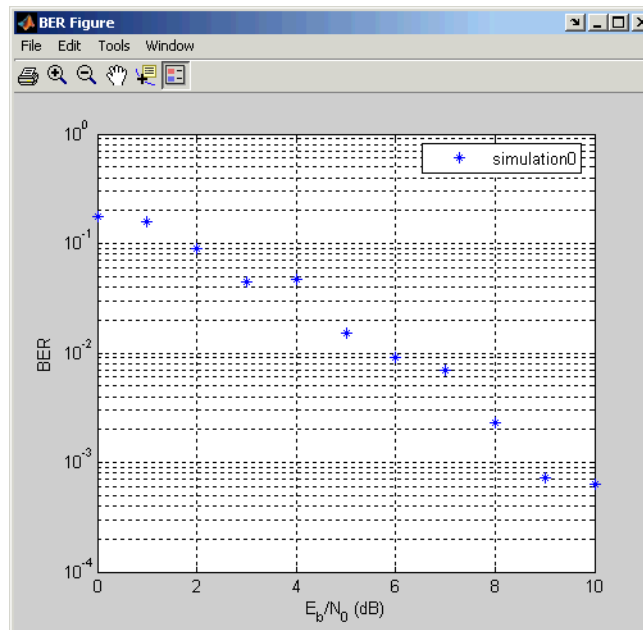
4 To fit a curve to the series of points in the BER Figure window, select the box next to **Fit** in the data viewer.

BERTool plots the curve, as shown in the following figure.
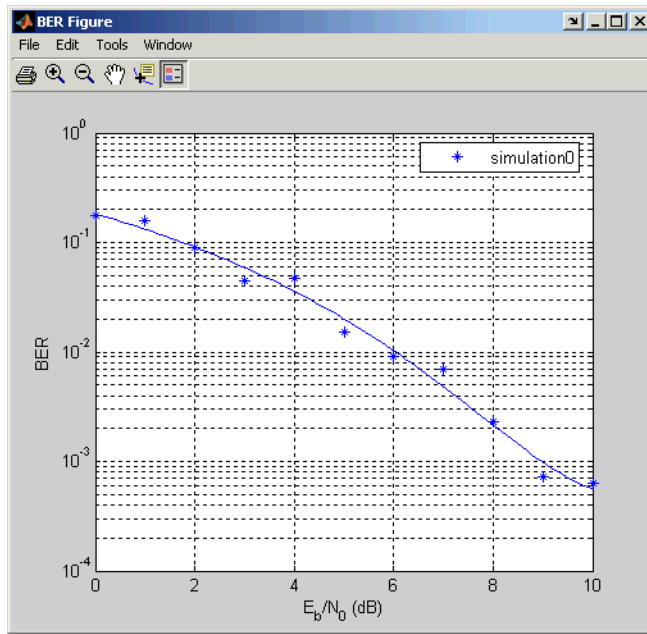
Running Simulink Simulations

In this section...
"Section Overview" on page 5-37
"Example: Using a Simulink Model with BERTool" on page 5-38
"Varying the Stopping Criteria" on page 5-41

Section Overview

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. Communications Blockset software is highly recommended. The rest of this section assumes you have a license for both Simulink and Communications Blockset applications.

To access the capabilities of BERTool related to Simulink models, open the **Monte Carlo** tab.
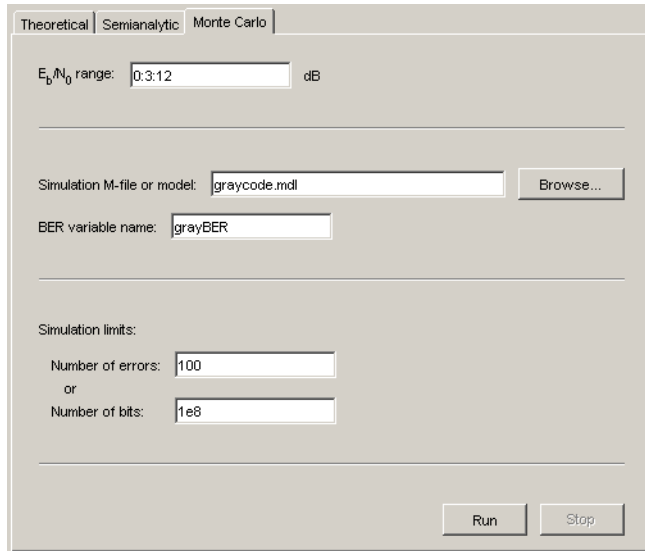
For further details about confidence intervals and curve fitting for simulation data, see "Plotting Confidence Intervals" on page 5-26 and "Fitting BER Points to a Curve" on page 5-28, respectively.

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 commgraycode, one of the demonstration models included with Communications Blockset software. The example assumes that you have Communications Blockset software installed.

To run this example, follow these steps:

1 Open BERTool and go to the **Monte Carlo** tab. The model's file name, commgraycode.mdl, appears as the **Simulation M-file or model** parameter. (If viterbisim.m appears there, select to indicate that Communications Blockset software 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. BERTool creates a listing in the data viewer.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☐	☑	simulation0	0:3:12	[0.1233 0.068 ...	[900 1500 480...

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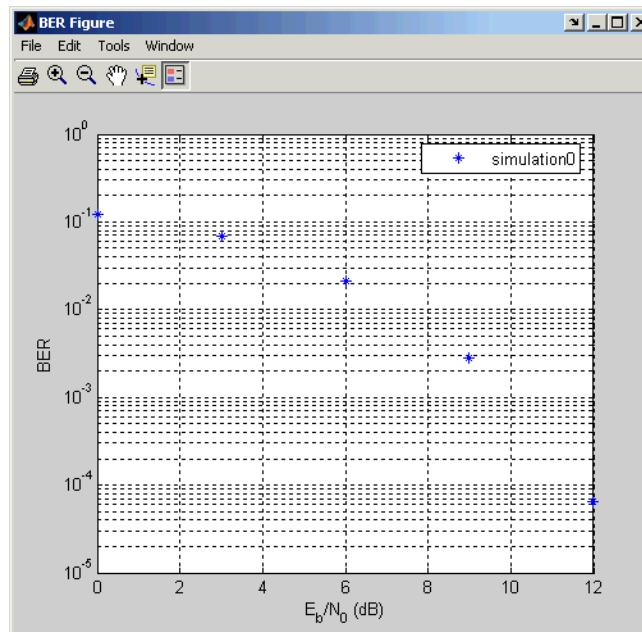
3 To fit a curve to the series of points in the BER Figure window, select the box next to **Fit** in the data viewer.

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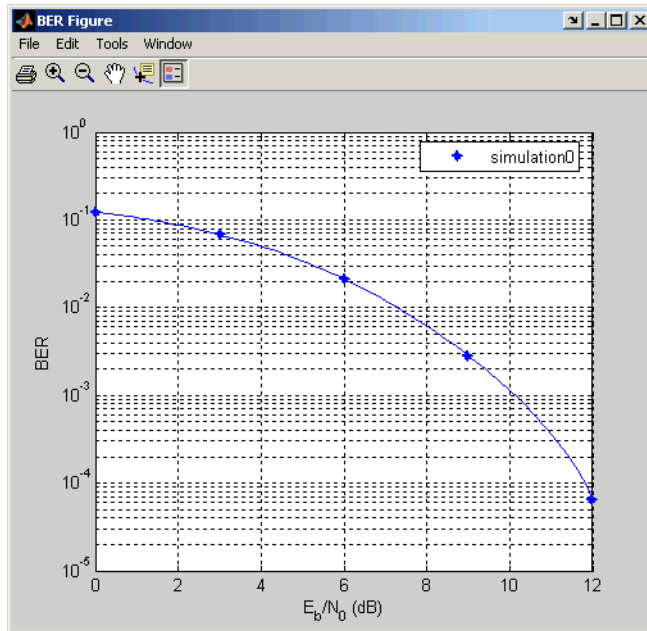
4 To indicate the 99% confidence interval around each point in the simulation data, set **Confidence Level** to 99% in the data viewer.

BERTool displays error bars to represent the confidence intervals, as below.
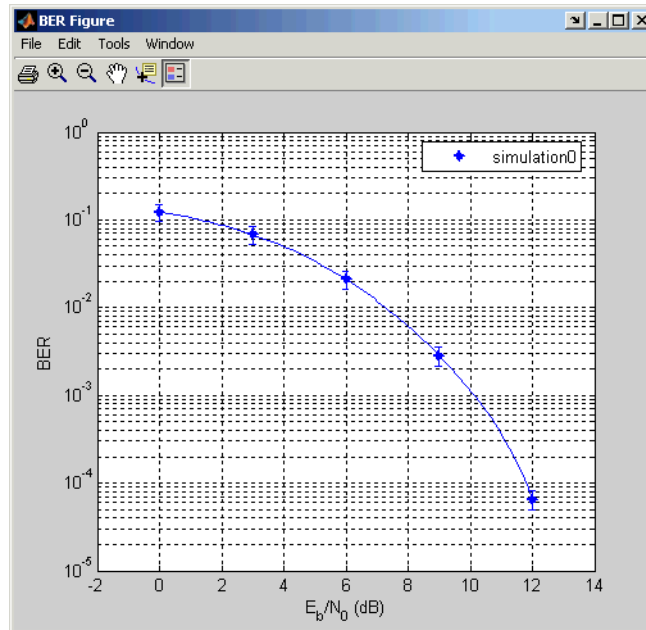
Another example that uses BERTool to manage a series of Simulink simulations is in "Example: Preparing a Model for Use with BERTool" on page 5-46.

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 5-43; for an example, see "Example: Preparing a Model for Use with BERTool" on page 5-46.

After creating your Simulink model, set the target number of errors and the maximum number of bits in the **Monte Carlo** tab 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, 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 click **Stop** in BERTool to stop a series of simulations prematurely.

Preparing Simulink Models for Use with BERTool

In this section...

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 Communications Blockset software 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 Communications Blockset software with these parameters:

  - **Mode** = `Signal to noise ratio (Eb/No)`

  - **Eb/No** = `EbNo`
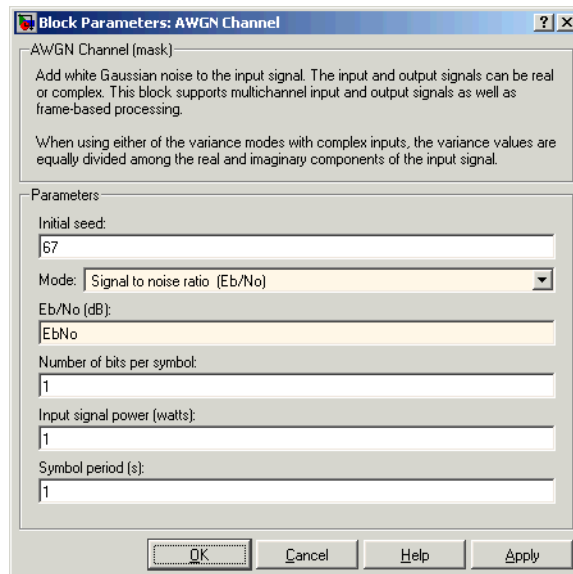
- To compute the error rate, use the Error Rate Calculation block in Communications Blockset software with these parameters:

  - Check **Stop simulation**.

  - **Target number of errors** = `maxNumErrs`

  - **Maximum number of symbols** = `maxNumBits`
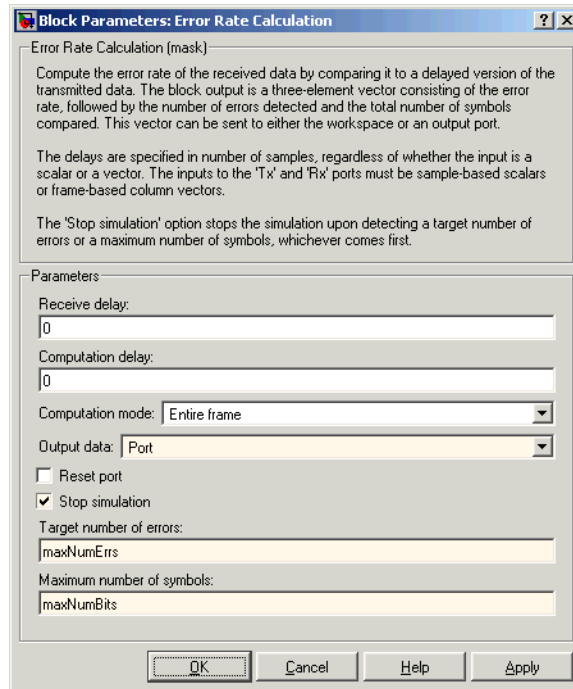
- To send data from the Error Rate Calculation block to the MATLAB workspace, set **Output data** to Port, attach a Signal to Workspace block from Signal Processing Blockset™ software, and set the latter block's **Limit data points to last** parameter to 1. The **Variable name** parameter in the Signal 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, use the Integer to Bit Converter block in Communications Blockset software to convert symbols to bits.

- Frame-based simulations often run faster than sample-based simulations for the same number of bits processed. The number of errors or number of processed bits might exceed the values 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), 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, 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, 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 click **Run**. Similarly, if you refresh your workspace by issuing a `clear all` or `clear variables` command, 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 Communications Blockset software 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
```
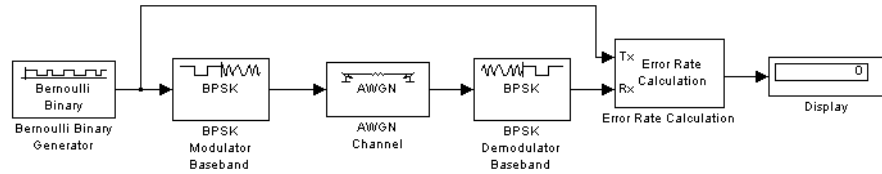
2 To initialize parameters in the MATLAB workspace and avoid using undefined variables as block parameters, enter the following command in the MATLAB Command Window.

```
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 Signal Processing Blockset software and is different from the To Workspace block in Simulink.
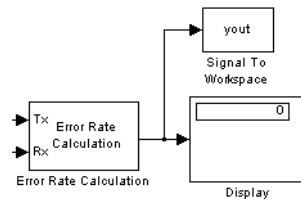
6 To configure the newly added Signal to Workspace block, open its dialog box. Set **Variable name** to BER, set **Limit data points to last** to 1, and click **OK**.
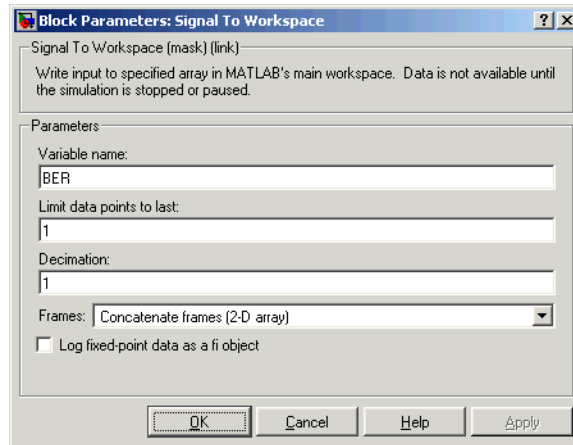
7 (*Optional*) To make the simulation run faster, especially at high values of E_b/N_0, open the dialog box for the Bernoulli Binary Generator block. Select **Frame-based outputs** and set **Samples per frame** to 1000.

8 Save the model in a folder on your MATLAB path using the file name 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 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** tab.

11 Set parameters on the **Monte Carlo** tab as shown in the following figure.

12 Click **Run**.

BERTool spends some time computing results and then plots them.
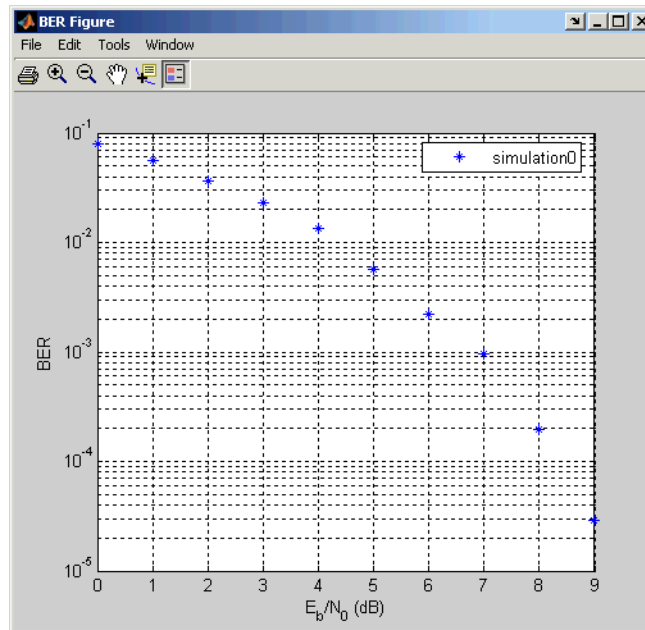
13 To compare these simulation results with theoretical results, go to the **Theoretical** tab in BERTool and set parameters as shown below.

14 Click **Plot**.

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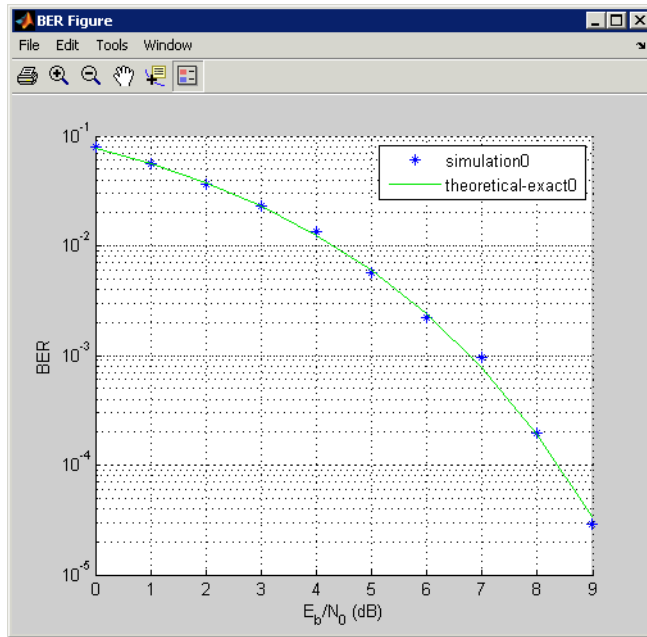
Managing BER Data

In this section...
"Exporting Data Sets or BERTool Sessions" on page 5-52
"Importing Data Sets or BERTool Sessions" on page 5-55
"Managing Data in the Data Viewer" on page 5-57

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:

- "Exporting Data Sets" on page 5-52
- "Examining an Exported Structure" on page 5-54
- "Saving a BERTool Session" on page 5-55

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 **File > Export Data**.
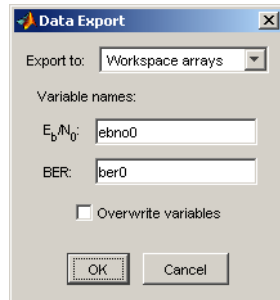
3 Set **Export to** to indicate the format and destination of the data.

    a If you want to reimport the data into BERTool later, 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, select **Overwrite variables**.

    b If you do *not* need to reimport the data into BERTool later, 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 array 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, select **Overwrite variables**.

4 Click **OK**. If you selected `MAT-file structure`, 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 5-56.

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, `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, 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 5-55.

Data Field. If your exported workspace variable is called ber0, 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 **File > Save Session**.

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** tab), 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:

- "Importing Data Sets" on page 5-56

- "Opening a Previous BERTool Session" on page 5-56

To learn more about exporting data sets or saving sessions from BERTool, see "Exporting Data Sets or BERTool Sessions" on page 5-52.

Importing Data Sets

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

1 Choose **File > Import Data**.
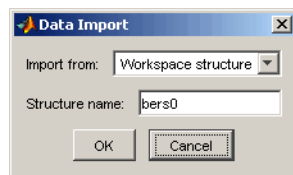
2 Set **Import from** to either `Workspace structure` or `MAT-file structure`. If you select `Workspace structure`, type the name of the workspace variable in the **Structure name** field.

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

After you dismiss the **Data Import** dialog box (and the file selection dialog box, 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 **File > Open Session**.

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

2 When BERTool prompts you, enter the path to the file 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** tab) that you saved separately in a MAT-file, 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 **Edit > Delete**.

Note If the data set originated from the **Semianalytic** or **Theoretical** tab, 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.

6

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 Communications Toolbox software in the sections listed below.

- "Quantizing a Signal" on page 6-2
- "Optimizing Quantization Parameters" on page 6-6
- "Differential Pulse Code Modulation" on page 6-8
- "Optimizing DPCM Parameters" on page 6-11
- "Companding a Signal" on page 6-13
- "Huffman Coding" on page 6-15
- "Arithmetic Coding" on page 6-17
- "Selected Bibliography for Source Coding" on page 6-19

This toolbox does not support vector quantization.

Quantizing a Signal

Section Overview

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 the MATLAB environment, 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];
```

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])
```
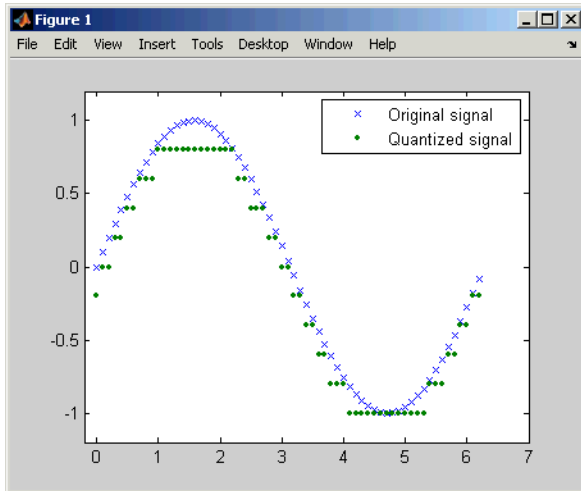

Determining Which Interval Each Input Is In

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

In this section...
"Section Overview" on page 6-6
"Example: Optimizing Quantization Parameters" on page 6-6

Section Overview

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 you use should be typical of the kinds of signals 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. 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

Section Overview

The quantization in the section "Quantizing a Signal" on page 6-2 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 6-2 and "Representing Codebooks" on page 6-3, 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 kth 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
```

Optimizing DPCM Parameters

Section Overview

The section "Optimizing Quantization Parameters" on page 6-6 describes
how to 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 you use with `dpcmopt` should be typical of the kinds
of signals 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, where 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 this section...
"Section Overview" on page 6-13
"Example: μ-Law Compander" on page 6-13

Section Overview

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: 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 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

Section Overview

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 6-17.

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

Section Overview

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, NJ, Prentice-Hall, 1988.

7

Error Detection and Correction

Error detection and correction 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" on page 7-2
- "Convolutional Coding" on page 7-32
- "Cyclic Redundancy Check Coding" on page 7-46

Block Coding

In this section...

Section Overview

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 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 7-4 • "Block Coding Terminology" on page 7-5 • "Performing Other Block Code Tasks" on page 7-27 • "Selected Bibliography for Block Coding" on page 7-30
Reed-Solomon	• "Representing Words for Reed-Solomon Codes" on page 7-5 • "Parameters for Reed-Solomon Codes" on page 7-6 • "Creating and Decoding Reed-Solomon Codes" on page 7-8
Bose-Chaudhuri-Hocquenghem (BCH)	• "Representing Words for BCH Codes" on page 7-12 • "Parameters for BCH Codes" on page 7-13 • "Creating and Decoding BCH Codes" on page 7-13
Low-density parity-check (LDPC)	• "LDPC Codes" on page 7-15
Cyclic, Hamming, and generic linear block	• "Representing Words for Linear Block Codes" on page 7-16 • "Parameters for Linear Block Codes" on page 7-20 • "Creating and Decoding Linear Block Codes" on page 7-24

Block Coding Features of the Toolbox

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

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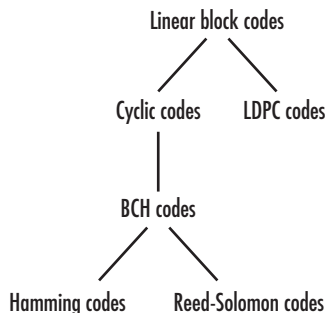
Communications Toolbox supports general linear block codes. It also includes functions to process cyclic, LDPC, BCH, Hamming, and Reed-Solomon codes (which are all special kinds of linear block codes). Functions in the toolbox can accomplish these tasks:

- 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 and Objects
Linear block	`encode`, `decode`, `gen2par`, `syndtable`
Cyclic	`encode`, `decode`, `cyclpoly`, `cyclgen`, `gen2par`, `syndtable`
BCH	`bchenc`, `bchdec`, `bchgenpoly`
LDPC	`fec.ldpcenc`, `fec.ldpcdec`
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 13-4 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

Symbol	Meaning	Value or Range
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. This 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, 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 7-5 and "Parameters for Reed-Solomon Codes" on page 7-6.

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

- "Example: Reed-Solomon Coding Syntaxes" on page 7-8
- "Example: Detecting and Correcting Errors in a Reed-Solomon Code" on page 7-9
- "Excessive Noise in Reed-Solomon Codewords" on page 7-10
- "Creating Shortened Reed-Solomon Codes" on page 7-11

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.

This 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.
```

The array of noise values contains integers between 1 and 2^m, and 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, that means 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 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. 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 7-12 and "Parameters for BCH Codes" on page 7-13. This section illustrates how to use bchenc and bchdec.

The topics are

- "Example: BCH Coding Syntaxes" on page 7-13

- "Example: Detecting and Correcting Errors in a BCH Code" on page 7-14

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 7-10.

LDPC Codes

Low-Density Parity-Check (LDPC) codes are linear error control codes with:

• Sparse parity-check matrices

- Long block lengths that can attain performance near the Shannon limit (see `fec.ldpcenc` and `fec.ldpcdec`)

The decoding process is done iteratively. If the number of iterations is too small, the algorithm may not converge. You may need to experiment with the number of iterations to find an appropriate value for your model. For details on the decoding algorithm, see Decoding Algorithm.

Unlike some other codecs, you cannot connect an LDPC decoder directly to the output of an LDPC encoder, because the decoder requires log-likelihood ratios (LLR). Thus, you may use a demodulator to compute the LLRs.
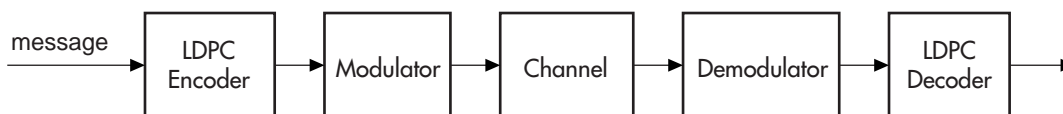
Also, unlike other decoders, it is possible (although rare) that the output of the LDPC decoder does not satisfy all parity checks.

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:

- "Binary Vector Format" on page 7-16
- "Binary Matrix Format" on page 7-18
- "Decimal Vector Format" on page 7-19

To learn how to represent words for BCH or Reed-Solomon codes, see "Representing Words for BCH Codes" on page 7-12 or "Representing Words for Reed-Solomon Codes" on page 7-5.

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, 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, 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 7-20	Generic linear block
"Parity-Check Matrix" on page 7-21	Generic linear block
"Generator Polynomial" on page 7-23	Cyclic
"Decoding Table" on page 7-23	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, 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 7-21.

Parity-Check Matrix

Decoding an [n,k] linear block code requires an (n-k)-by-n parity-check matrix H. It satisfies $GH^{tr} = 0$ (mod 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 $[n,k] = [2^3-1, n-3] = [7,4]$. `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 7-23.

```
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, 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 7-16 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 7-23 and 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 7-16 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 7-25 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 7-23 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" on page 7-28

• "Error Correction Versus Error Detection for Linear Block Codes" on page 7-29

• "Finding the Error-Correction Capability" on page 7-29

- "Finding Generator and Parity-Check Matrices" on page 7-30
- "Converting Between Parity-Check and Generator Matrices" on page 7-30

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 13-8 and "Polynomials over Galois Fields" on page 13-33.

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.

Error Correction Versus Error Detection for Linear Block Codes

You can use a liner block code to detect $d_{\min}$ -1 errors or to correct $t =$

$\left\lceil \frac{1}{2}(d_{\min} - 1) \right\rceil$ errors. If you compromise the error correction capability of a code, you can detect more than t errors. For example, a code with $d_{\min} = 7$ can correct $t = 3$ errors or it can detect up to 4 errors and correct up to 2 errors.

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 three 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. The reference page for `gen2par` contains examples to illustrate this.

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, NJ, Prentice-Hall, 1983.

[4] Peterson, W. Wesley, and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed., Cambridge, MA, 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, NJ, Prentice Hall, 1995.

[7] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

[8] Ryan, William E., "An introduction to LDPC codes," *Coding and Signal Processing for Magnetic Recoding Systems* (Vasic, B., ed.), CRC Press, 2004.

Convolutional Coding

In this section...

Section Overview

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.

Convolutional Coding Features of the Toolbox

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 7-45.

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.
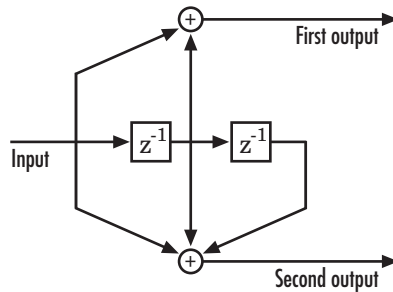
A polynomial description of a convolutional encoder has either two or three components, depending on whether the encoder is a feedforward or feedback type:

- 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, 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 7-34 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, 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, so 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 7-30, especially that book's appendices.

Feedback Connection Polynomials

If you are representing a feedback encoder, 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, 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.

First output (systematic)

1   1   1   1   1

Input

Z⁻¹   Z⁻¹   Z⁻¹   Z⁻¹

1   1   0   1   1

Second output
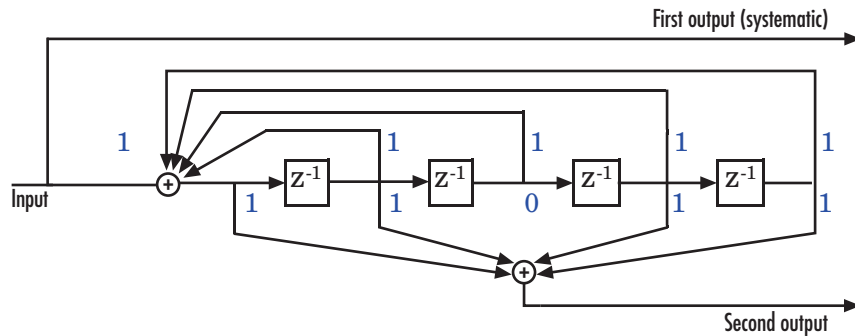
This encoder has a constraint length of 5, a generator polynomial matrix of [37 33], and a feedback connection polynomial of 37.

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. 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 7-32.

```
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, and 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.

——— State transition when input is 0
– – – State transition when input is 1
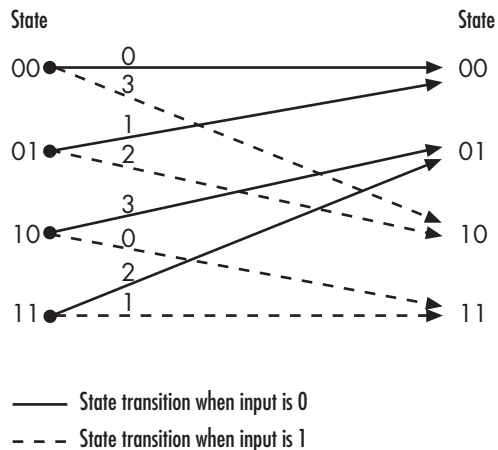
As an example of interpreting this trellis diagram, if the encoder is in the 10 state and receives an input of zero, it outputs the code symbol 3 and changes to the 01 state. If it is in the 10 state and receives an input of one, it outputs the code symbol 0 and changes to the 11 state.

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 octal) 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 7-32.

To check whether your structure is a valid trellis structure, use the `istrellis` function.

Example: A MATLAB Trellis Structure

Consider the trellis shown below.

State transition when input is 0
- - - State transition when input is 1
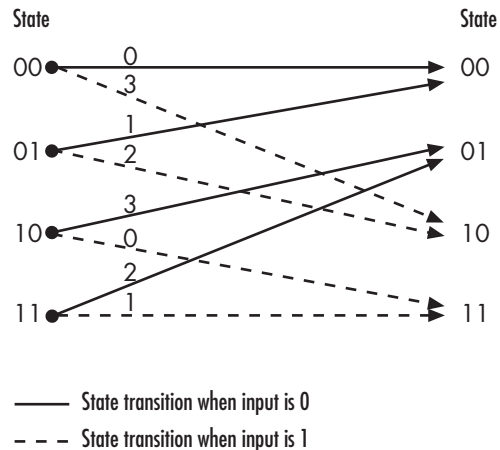
To build a trellis structure that describes it, use the command below.

```
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'`. 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. 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 this schematic. 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.
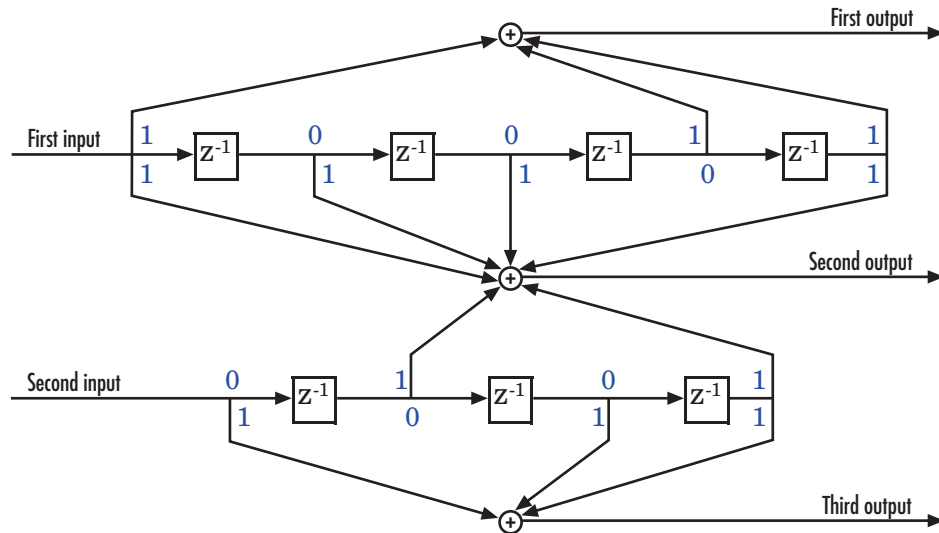
Determining Coding Parameters. The `convenc` and `vitdec` functions can implement this code if their parameters have the appropriate values.

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, 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));
```

`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 30,000 random bits and encoding them using a rate-3/4
convolutional encoder with a puncture pattern of [1 1 1 0 0 1]. The resulting
vector contains 40,000 bits, which are mapped to values of -1 and 1 for
transmission. The punctured code, `punctcode`, passes through an additive
white Gaussian noise channel. Then `vitdec` decodes the noisy vector using
the `'unquant'` decision type.

Finally, the example computes the bit error rate and the number of bit errors.

```
len = 30000; msg = randi([0 1], len, 1); % Random data
t = poly2trellis(7, [133 171]); % Define trellis.
punctcode = convenc(msg, t, [1 1 1 0 0 1]); % Length is (2*len)*2/3.
tcode = 1-2*punctcode; % Map "0" bit to 1 and "1" bit to -1
ncode = awgn(tcode, 3, 'measured'); % Add noise.

% Decode the punctured code
decoded = vitdec(ncode, t, 96, 'trunc', ...
'unquant', [1 1 1 0 0 1]); % Decode.
[numErrP, berP] = biterr(decoded, msg); % Bit error rate
```

```
% Erase the least reliable 100 symbols, then decode
[dummy idx] = sort(abs(ncode));
erasures =  zeros(size(ncode)); erasures(idx(1:100)) = 1;
decoded = vitdec(ncode, t, 96, 'trunc', 'unquant', ...
[1 1 1 0 0 1], erasures); % Decode.
[numErrPE, berPE] = biterr(decoded, msg); % Bit error rate

fprintf('Number of errors with puncturing: %d\n', numErrP)
fprintf('Number of errors with puncturing and erasures: %d\n', numErrPE)
```

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.

Cyclic Redundancy Check Coding

In this section...
"Overview" on page 7-46
"CRC Algorithm" on page 7-46
"Selected Bibliography for CRC Coding" on page 7-48

Overview

Cyclic redundancy check (CRC) coding is an error-control coding technique for detecting errors that occur when a message is transmitted. Unlike block or convolutional codes, CRC codes do not have a built-in error-correction capability. Instead, when an error is detected in a received message word, the receiver requests the sender to retransmit the message word.

In CRC coding, the transmitter applies a rule to each message word to create extra bits, called the checksum, or syndrome, and then appends the checksum to the message word. After receiving a transmitted word, the receiver applies the same rule to the received word. If the resulting checksum is nonzero, an error has occurred, and the transmitter should resend the message word.

Open the Error Detection and Correction library by double-clicking its icon in the main Communications Toolbox library. Open the CRC sublibrary by double-clicking on its icon in the Error Detection and Correction library.

CRC Algorithm

The CRC algorithm accepts a binary data vector, corresponding to a polynomial M, and appends a checksum of r bits, corresponding to a polynomial C. The concatenation of the input vector and the checksum then corresponds to the polynomial $T = M \star x^r + C$, since multiplying by x^r corresponds to shifting the input vector r bits to the left. The algorithm chooses the checksum C so that T is divisible by a predefined polynomial P of degree r, called the *generator polynomial*.

The algorithm divides T by P, and sets the checksum equal to the binary vector corresponding to the remainder. That is, if $T = Q \star P + R$, where R is a polynomial of degree less than r, the checksum is the binary vector

corresponding to *R*. If necessary, the algorithm prepends zeros to the checksum so that it has length r.

The CRC generation feature, which implements the transmission phase of the CRC algorithm, does the following:

1 Left shifts the input data vector by *r* bits and divides the corresponding polynomial by *P*.

2 Sets the checksum equal to the binary vector of length *r*, corresponding to the remainder from step 1.

3 Appends the checksum to the input data vector. The result is the output vector.

The CRC detection feature computes the checksum for its entire input vector, as described above.

The CRC algorithm uses binary vectors to represent binary polynomials, in descending order of powers. For example, the vector [1 1 0 1] represents the polynomial $x^3 + x^2 + 1$.

Note The implementation described in this section is one of many valid implementations of the CRC algorithm. Different implementations can yield different numerical results.
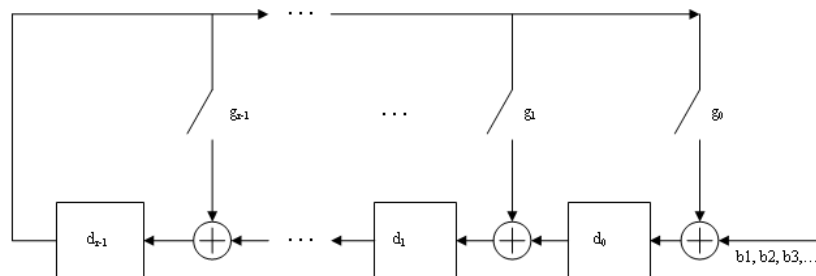
Bits enter the linear feedback shift register (LFSR) from the lowest index bit to the highest index bit. The sequence of input message bits represents the coefficients of a message polynomial in order of decreasing powers. The message vector is augmented with r zeros to flush out the LFSR, where r is the degree of the generator polynomial. If the output from the leftmost register stage $d(1)$ is a 1, then the bits in the shift register are XORed with the coefficients of the generator polynomial. When the augmented message sequence is completely sent through the LFSR, the register contains the checksum $[d(1)\ d(2)\ .\ .\ .\ d(r)]$. This is an implementation of binary long division, in which the message sequence is the divisor (numerator) and the polynomial is the dividend (denominator). The CRC checksum is the remainder of the division operation.

Selected Bibliography for CRC Coding

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

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

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 Communications Toolbox software in the sections listed below.

- "Block Interleavers" on page 8-2
- "Convolutional Interleavers" on page 8-5
- "Selected Bibliography for Interleaving" on page 8-14

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

In this section...
"Section Overview" on page 8-2
"Block Interleaving Features of the Toolbox" on page 8-2
"Example: Block Interleavers" on page 8-3

Section Overview

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.

Type of Interleaver	Interleaver Function	Description
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 follows.

```
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

In this section...

Section Overview

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.
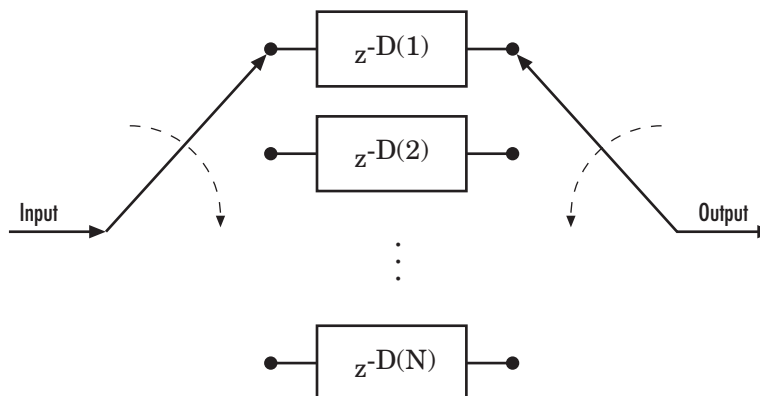
Convolutional Interleaving Features of the Toolbox

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]

Current Input	Current Shift Register	Current Output	Contents of Shift Registers
6	3	0	[ ] [5] [3 6]
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 8-9.

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 following code 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.

9

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

Modulation Techniques

The available methods of modulation depend on whether the input signal is analog or digital. The tables below show the modulation techniques that Communications Toolbox software supports for analog and digital signals, respectively.

Analog Modulation Method	Acronym	Function or Method
Amplitude modulation (suppressed or transmitted carrier)	AM	`ammod, amdemod`
Frequency modulation	FM	`fmmod, fmdemod`
Phase modulation	PM	`pmmod, pmdemod`
Single sideband amplitude modulation	SSB	`ssbmod, ssbdemod`

Digital Modulation Method	Acronym	Function or Method
Differential phase shift keying modulation	DPSK	`modulate` method on `modem.dpskmod` object, `demodulate` method on `modem.dpskdemod` object
Frequency shift keying modulation	FSK	`fskmod, fskdemod`

Digital Modulation Method	Acronym	Function or Method
General Quadrature amplitude modulation	General QAM	`modulate` method on `modem.genqammod` object, `demodulate` method on `modem.genqamdemod` object
Minimum shift keying modulation	MSK	`modulate` method on `modem.mskmod` object, `demodulate` method on `modem.mskdemod` object
Offset quadrature phase shift keying modulation	OQPSK	`modulate` method on `modem.oqpskmod` object, `demodulate` method on `modem.oqpskdemod` object
Phase shift keying modulation	PSK	`modulate` method on `modem.pskmod` object, `demodulate` method on `modem.pskdemod` object
Pulse amplitude modulation	PAM	`modulate` method on `modem.pammod` object, `demodulate` method on `modem.pamdemod` object
Quadrature amplitude modulation	QAM	`modulate` method on `modem.qammod` object, `demodulate` method on `modem.qamdemod` object

Baseband vs. 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

In this section...
"Representing Analog Signals" on page 9-5
"Analog Modulation Example" on page 9-6

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.
```
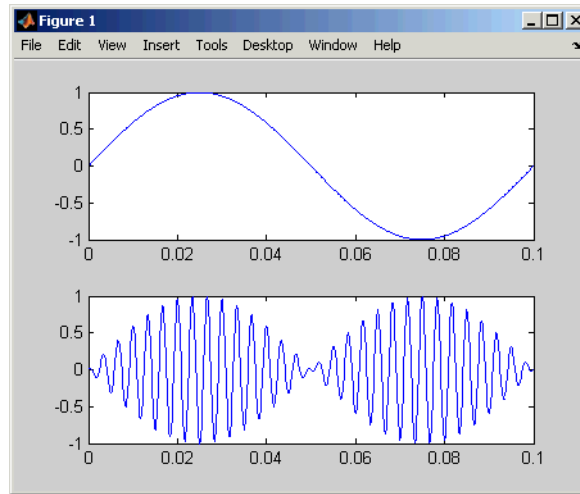
As a multichannel example, the code below defines a two-channel signal in which one channel is a sinusoid with zero initial phase and the second channel is a sinusoid with an initial phase of `pi/8`.

```
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 reference pages for ammod, amdemod, ssbdemod, and fmmod.

Digital Modulation

Section Overview

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 9-2. You can also plot signal constellations. Modulation functions output the complex envelope of the modulated signal.

Note The modulation and demodulation functions do not perform pulse shaping or filtering. See Chapter 10, "Special Filters" or "Combining Pulse Shaping and Filtering with Modulation" on page 9-13 for more information about filtering.

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 from 0 to M-1. 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 eight 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.

Gray Encoding a Modulated Signal

For the PSK, DPSK, FSK, QAM, and PAM modulation types, Gray
constellations are obtained by selecting the Gray parameter in the
corresponding modulation function or method.

For modulation objects, you can set the symbol order property to Gray to
obtain Gray-encoded modulation.

The following example demonstrates use of the symbol order property. The
Scatter plot shows the modulated symbols are Gray-encoded.

```
% Create 8-PSK Gray encoded modulator
hMod = modem.pskmod('M',8,'SymbolOrder','Gray');
% Create a scatter plot
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',hMod.Constellation);
% Show constellation
scatterPlot.PlotSettings.Constellation = 'on';
scatterPlot.PlotSettings.ConstellationStyle = 'rd';
% Add symbol labels
hold on;
k=log2(hMod.M);
for jj=1:hMod.M
        text(real(hMod.Constellation(jj))-0.15,imag(hMod.Constellation(j
        dec2base(hMod.SymbolMapping(jj),2,k));
end
hold off;
```
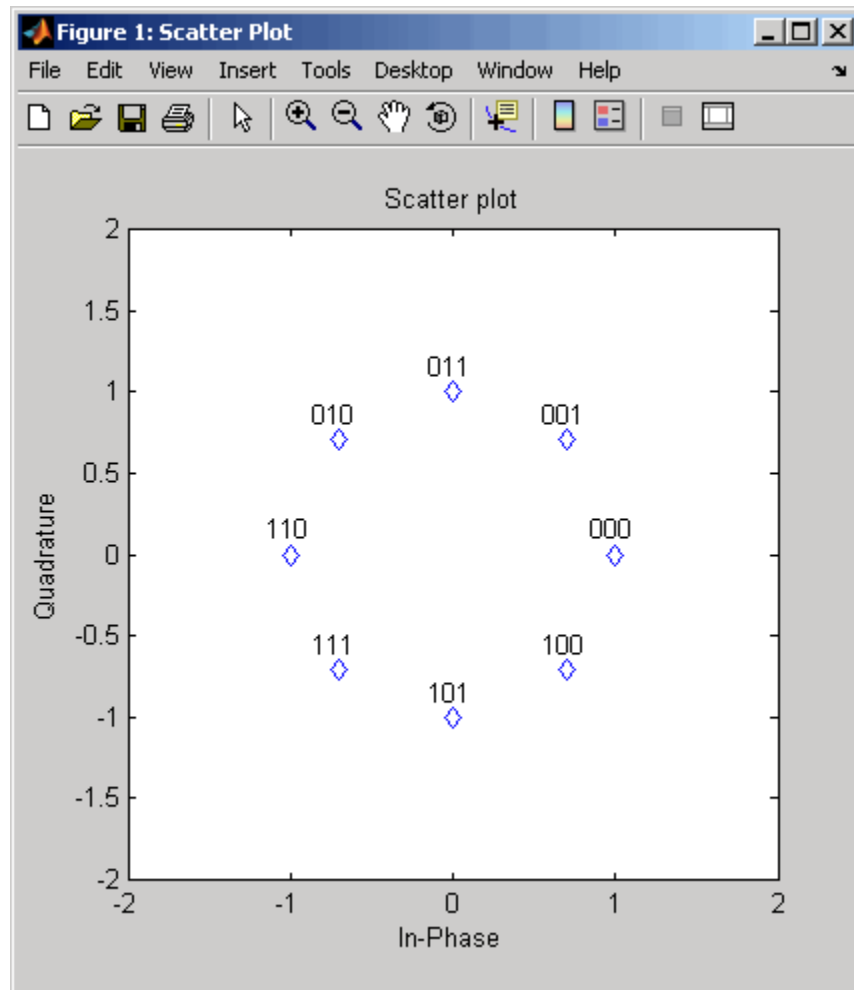
For modulation functions, set the symbol order argument to Gray.

Looking at the map above, notice that this is indeed a Gray-encoded map; all adjacent elements differ by only one bit.

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 = randi([0 M-1],5000,1);  % Random symbols

% Use 16-QAM modulation.
hMod = modem.qammod(M);
hDemod = modem.qamdemod(hMod);

% Create a scatter plot and show constellation
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',hMod.Constellation);
scatterPlot.PlotSettings.Constellation = 'on';

% Modulate
y = modulate(hMod,x);

% Transmit signal through an AWGN channel.
ynoisy = awgn(y,15,'measured');

% Create scatter plot from noisy data.
update(scatterPlot,ynoisy);

% Demodulate ynoisy to recover the message.
z=demodulate(hDemod,ynoisy);

% Check symbol error rate.
[num,rt] = symerr(x,z)
```

The output and scatter plot follow. Your numerical results and plot might vary, because the example uses random numbers.

```
num =

    83


rt =

    0.0166
```
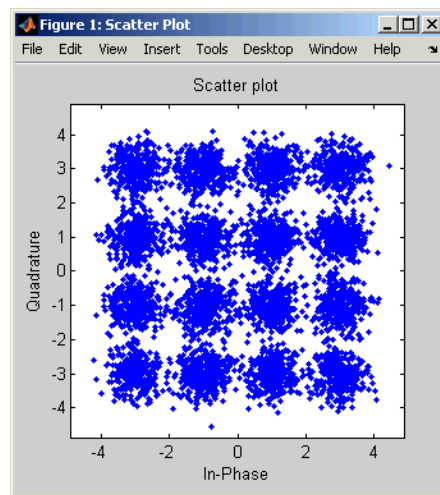

The scatter plot does not look exactly like a signal constellation. Where the signal constellation has 16 precisely located points, the noise causes the scatter plot to have a small cluster of points approximately where each constellation point would be.

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 1-15.

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 = randi([0 M-1],5000,1);  % Message signal
Nsamp = 4;                     % Oversampling rate

% Use 16-QAM modulation.
hMod = modem.qammod(M);
hDemod = modem.qamdemod(hMod);

% Modulate
y = modulate(hMod,x);

% 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 = demodulate(hDemod,ydownsamp);
```

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 9-15
- "Constellation for 32-QAM" on page 9-16
- "Gray-Coded Signal Constellation" on page 9-17
- "Customized Constellation for QAM" on page 9-18

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.

```
% Use 16-PSK modulation.
hMod = modem.pskmod(16);

% Create a scatter plot
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',hMod.Constellation);
% Show constellation
scatterPlot.PlotSettings.Constellation = 'on';
scatterPlot.PlotSettings.ConstellationStyle = 'rd';
% Add symbol labels
hold on;
k=log2(hMod.M);
for jj=1:hMod.M
        text(real(hMod.Constellation(jj))-0.15,...,
        imag(hMod.Constellation(jj))+0.15,...
        dec2base(hMod.SymbolMapping(jj),2,k));
end
hold off;
```
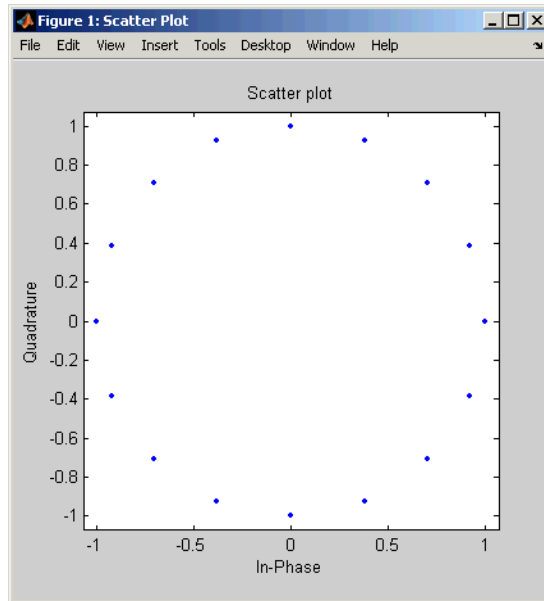
Constellation for 32-QAM. The code below plots a QAM constellation having 32 points and a peak power of 1 watt. The example also illustrates how to label the plot with the numbers that form the input to the modulator.

```
% Create 32-QAM modulator
hMod = modem.qammod(32);
% Create a scatter plot
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',hMod.Constellation);
% Show constellation
scatterPlot.PlotSettings.Constellation = 'on';
scatterPlot.PlotSettings.ConstellationStyle = 'rd';
% Add symbol labels
hold on;
for jj=1:hMod.M
   text(real(hMod.Constellation(jj)),imag(hMod.Constellation(jj)),...
   [' ' num2str(hMod.SymbolMapping(jj))]);
end
hold off;
```
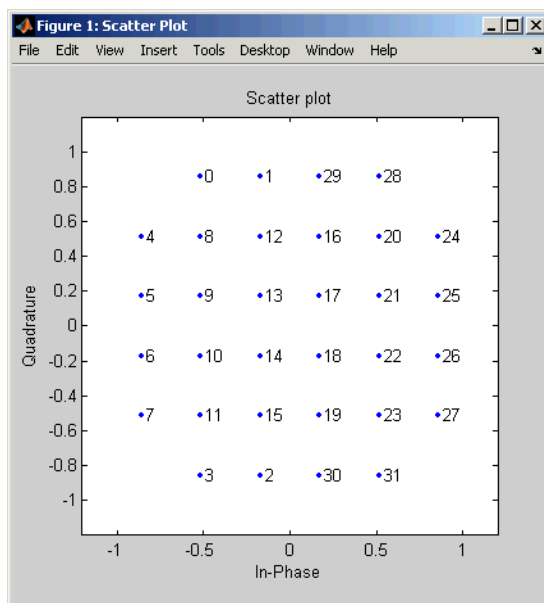
Gray-Coded Signal Constellation. The example below plots an 8-QAM signal Gray-coded constellation, labeling the points using binary numbers so you can verify visually that the constellation uses Gray coding.

```
% Create 8-QAM Gray encoded modulator
hMod = modem.qammod('M',8,'SymbolOrder','Gray');
% Create a scatter plot
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',hMod.Constellation);
% Show constellation
scatterPlot.PlotSettings.Constellation = 'on';
scatterPlot.PlotSettings.ConstellationStyle = '.';
% Add symbol labels
hold on;
k=log2(hMod.M);
for jj=1:hMod.M
        text(real(hMod.Constellation(jj))+0.15,...,
        imag(hMod.Constellation(jj)),...
        dec2base(hMod.SymbolMapping(jj),2,k));
end
```

```
hold off;
```
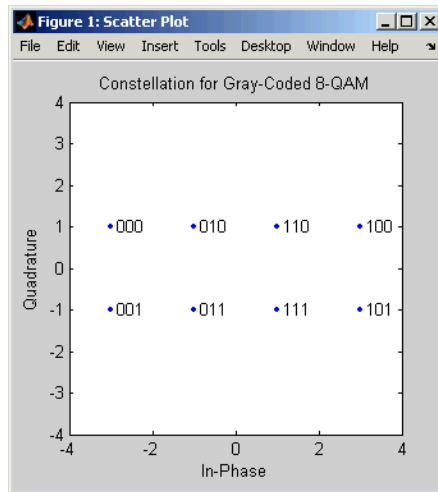

Customized Constellation for QAM. The code below describes and plots a constellation with a customized structure.

```
% 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 + 1i*quadr;

% Create a scatter plot
scatterPlot = commscope.ScatterPlot('SamplesPerSymbol',1,...
    'Constellation',const);
% Show constellation
scatterPlot.PlotSettings.Constellation = 'on';
scatterPlot.PlotSettings.ConstellationStyle = '*';
title('Customized Constellation for QAM');
```
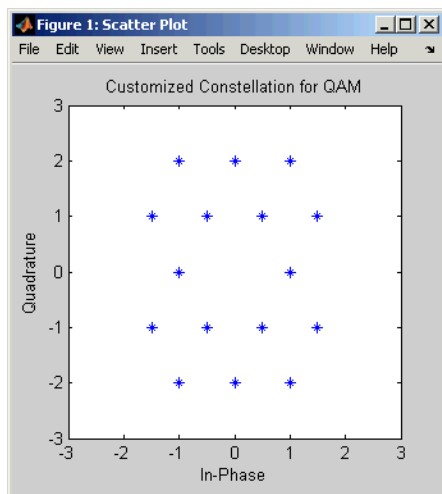
Using Modem Objects

Section Overview

Signal modulation generally requires the use of functions, such as `fskmod` or `ssbmod`. For DPSK, General QAM, MSK, OQPSK, PAM, PSK, and QAM, however, you modulate signals through the use of modem objects. This section gives an overview of how you use these objects.

A *modem object* is a type of MATLAB variable that contains information about the modulation algorithm, such as the name of the modulation class, M-ary number, and the constellation mapping. The object can be operated upon using specific methods to perform certain tasks.

Constructing a Modem Object

To construct modulator and demodulator objects, use the functions (constructors) shown in the following table.

Modulation Type	**Constructors**
DPSK	`modem.dpskmod` and `modem.dpskdemod`

Modulation Type	Constructors
General QAM	`modem.genqammod` and `modem.genqamdemod`
MSK	`modem.mskmod` and `modem.mskdemod`
OQPSK	`modem.oqpskmod` and `modem.oqpskdemod`
PAM	`modem.pammod` and `modem.pamdemod`
PSK	`modem.pskmod` and `modem.pskdemod`
QAM	`modem.qammod` and `modem.qamdemod`

See individual reference pages for details.

Managing Object Properties

To view the properties of a modem object, use its `disp` method, as shown in the following example:

```
h=modem.pskmod; % Construct a PSK modulator object.
h.disp          % Display object properties.
```

You can directly assign a value to a property as follows:

```
h=modem.pskmod(8);    % Construct a PSK modulator object.
% Set the 'symbolorder' property of the object to 'gray'.
h.symbolorder='gray';
```

The properties can also be set to specific values when constructing the object. See reference pages of individual objects for details.

Copying a Modem Object

The syntax `h = copy(refobj)` creates a new instance of an object, `h`, of the same type as `refobj`, and copies the properties of `refobj` into `h`.

Setting another variable equal to an object just copies its handle, and is not creating an independent copy of it. Thus, in the previous example, if you set `a = h`, then `a` points to the same object `h` and any changes made to `h` are also reflected in `a`.

Displaying a Modem Object

The syntax `disp(h)` displays relevant properties of object *h*.

If a property is not relevant to the object's configuration, it does not display. For example, for a MODEM.PSKDEMOD object, NoiseVariance property is not relevant when DecisionType property is set to 'Hard decision', hence NoiseVariance property does not display.

The following is an example of using `disp`:

```
h = modem.pskmod; % create an object with default properties
      disp(h); % display object properties
```

The output for this example looks like:

```
          Type: 'PSK Modulator'
             M: 2
   PhaseOffset: 0
 Constellation: [1 -1+i*1.22464679914735e-16]
   SymbolOrder: 'Binary'
 SymbolMapping: [0 1]
     InputType: 'Integer'
```

The following is an example of using `disp`:

```
h = modem.qamdemod('M', 32) % note the absence of semicolon
```

The output for this example looks like:

```
          Type: 'QAM Demodulator'
             M: 32
   PhaseOffset: 0
 Constellation: [1x32 double]
   SymbolOrder: 'Binary'
 SymbolMapping: [1x32 double]
    OutputType: 'Integer'
  DecisionType: 'Hard decision'
```

Resetting a Modem Object

The MSK, OQPSK, and DPSK modem objects (i.e., only those with memory) have a `reset` method that resets the internal states of the object.

It assumes that the number of channels of the input signal to the `modulate` or `demodulate` methods are one (i.e., the input is a column vector).

`reset(h,nchan)` resets the internal states of the object, `h`, assuming `nchan` number of channels, where the input to the modulator is a matrix of `nchan` columns. If the `modulate` or `demodulate` method is called with an input with number of channels different from `nchan`, the object automatically resets itself with the correct number of channels.

The following is an example of using `reset`:

```
h = modem.mskmod; % create an object with default properties
x = randint(100, 1, 2); % generate input bits
y = modulate(h, x); % modulate x
x = randint(100, 1, 2); % generate new input bits
reset(h); % reset the modulator
y = modulate(h, x); % modulate x with the same initial state
                    % as the first call
```

Modulating a Signal

The basic procedure for modulating a signal with DPSK, MSK, OQPSK, PAM, PSK, QAM, or general QAM involves these steps:

1 Construct a modulator object as shown in "Constructing a Modem Object" on page 9-20, depending on your modulation type.

2 Adjust properties of the modulator object, if necessary, to tailor it to your needs. For example, you can change the phase offset or symbol order.

3 Modulate your signal by applying the `modulate` method of the modulator object, as described in the following section.

Modem Modulation Method

Modulator objects have a method `modulate` that is used to modulate signals.

The syntax is `y = modulate(h, x)`, where `h` is the handle to a modulator object and `x` is a signal. This syntax outputs the baseband signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

When mapping input bits to symbols, the first bit is interpreted as the most significant bit.

For `h.inputtype = `bit'` (i.e., `x` represents binary input), $nBits$ consecutive elements in each channel or column represent a symbol, where $nBits = \log_2($h.M$)$. The number of elements in each channel must be an integer multiple of $nBits$, and elements of `x` must be 0 or 1. For an input `x` of size $R \times C$, an output `y` of size $(R/nBits) \times C$ is computed.

For `h.inputtype = `integer'` (i.e., `x` represents symbol input), elements of `x` must be in the range [0, `h.M-1`]. For an input `x` of size $R \times C$, an output `y` of size $R \times C$ is computed.

Demodulating a Signal

The basic procedure for demodulating a signal with DPSK, MSK, OQPSK, PAM, PSK, QAM, or general QAM involves these steps:

1 Construct a demodulator object as shown in "Constructing a Modem Object" on page 9-20, depending on your modulation type.

2 Adjust properties of the demodulator object, if necessary, to tailor it to your needs. For example, you can change the phase offset or symbol order.

3 Demodulate your signal by applying the `demodulate` method of the demodulator object, as described in the following section.

Modem Demodulation Method

Demodulator objects have a method `demodulate` that is used to demodulate signals.

The syntax is `y = demodulate(h, x)`, where `h` is the handle to a demodulator object and `x` is a signal. This syntax processes the binary words (bits) or symbols (integers) in signal `x` with the PSK or QAM demodulator object and output the baseband signal `y`.

`x` can be a multichannel signal. The columns of `x` are considered individual channels, while the rows are time steps.

The demodulator object's property `DecisionType` should be set depending on whether you want hard or soft (LLR or approximate LLR) decisions. To allow for soft decisions, the demodulator object's property `OutputType` must be set to `'bit'`.

For `h.outputtype = `bit'`, an output `y` of size $R \times (nBits \times C)$ is computed for an input `x` of size $R \times C$, where $nBits = \log_2(\text{h.M})$.

For `h.outputtype = `integer'`, an output `y` of size $R \times C$ is computed for an input `x` of size $R \times C$.

Example of Basic Modulation and Demodulation

This code briefly illustrates the steps in modulation and demodulation.

```
x = randint(10,1,8);    % Create a signal source.
h = modem.qammod(8)     % Create a modulator object
                        % and display its properties.
y = modulate(h,x);      % Modulate the signal x.
g = modem.qamdemod(h)   % Create a demodulator object
                        % from a modem.qammod object
                        % and display its properties.
z = demodulate(g,y);    % Demodulate the signal y.
```

Exact LLR Algorithm

The log-likelihood ratio (LLR) is the logarithm of the ratio of probabilities of a 0 bit being transmitted versus a 1 bit being transmitted for a received signal. The LLR for a bit b is defined as:

$$L(b) = \log\left(\frac{\Pr(b = 0 \mid r = (x, y))}{\Pr(b = 1 \mid r = (x, y))}\right)$$

Assuming equal probability for all symbols, the LLR for an AWGN channel can be expressed as:

$$L(b) = \log\left(\frac{\displaystyle\sum_{s \in S_0} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}{\displaystyle\sum_{s \in S_1} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}\right)$$

where the variables represent the values shown in the following table.

Variable	What the Variable Represents
r	Received signal with coordinates (x, y).
b	Transmitted bit (one of the K bits in an M-ary symbol, assuming all M symbols are equally probable.

Variable	What the Variable Represents
S_0	Ideal symbols or constellation points with bit 0, at the given bit position.
S_1	Ideal symbols or constellation points with bit 1, at the given bit position.
s_x	In-phase coordinate of ideal symbol or constellation point.
s_y	Quadrature coordinate of ideal symbol or constellation point.
σ^2	Noise variance of baseband signal.
σ_x^2	Noise variance along in-phase axis.
σ_y^2	Noise variance along quadrature axis.

Note Noise components along the in-phase and quadrature axes are assumed to be independent and of equal power (i.e., $\sigma_x^2 = \sigma_y^2 = \sigma^2/2$).

Approximate LLR Algorithm

Approximate LLR [4] is computed by taking into consideration only the nearest constellation point to the received signal with a 0 (or 1) at that bit position, rather than all the constellation points as done in exact LLR. It is defined as:

$$L(b) = -\frac{1}{\sigma^2}\left(\min_{s \in S_0}\left((x - s_x)^2 + (y - s_y)^2\right) - \min_{s \in S_1}\left((x - s_x)^2 + (y - s_y)^2\right)\right)$$

Selected Bibliography for Modulation

[1] Jeruchim, M. C., P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[2] Proakis, J. G., *Digital Communications*, 3rd ed., New York, McGraw-Hill, 1995.

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

[4] Viterbi, A. J., "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, No. 2, pp. 260–264, Feb. 1998.

Special Filters

Communications Toolbox software includes several functions that can help you design and use filters. Other filtering capabilities are in Signal Processing Toolbox software. The sections of this chapter are as follows.

For a demonstration involving raised cosine filters, type `showdemo rcosdemo`.

# Noncausality and the Group Delay Parameter

Section Overview

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 one second. In the figure, the impulse response near time 0 is small and the large impulse response values occur near time 1.
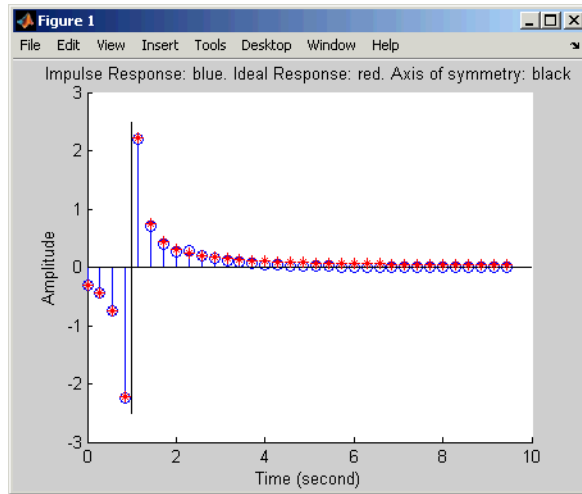
Example: Compensating for Group Delays in Data Analysis

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);
```

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 `showdemo rcosdemo`.
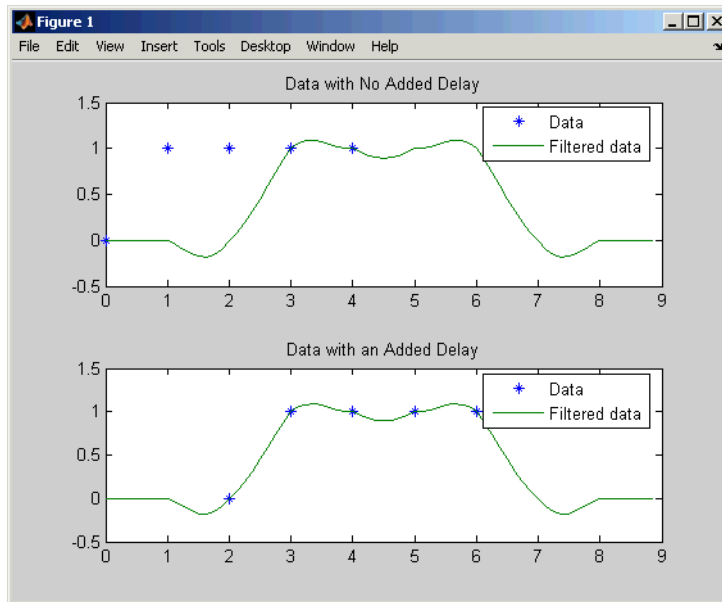
Designing Hilbert Transform Filters

Section Overview

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

```
hilbiir
```

plots the impulse response of a fourth-order digital Hilbert transform filter having a one-second group delay. The sample time is 2/7 seconds. In this particular design, the tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter having a one-second group delay. The plot is in the figure in "Noncausality and the Group Delay Parameter" on page 10-2.

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
```

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 in "Noncausality and the Group Delay Parameter" on page 10-2.

Filtering with Raised Cosine Filters

In this section...
"Section Overview" on page 10-7
"Sampling Rates" on page 10-7
"Designing Filters Automatically" on page 10-8
"Specifying Filters Using Input Arguments" on page 10-9
"Controlling the Rolloff Factor" on page 10-10
"Controlling the Group Delay" on page 10-10
"Combining Two Square-Root Raised Cosine Filters" on page 10-12

Section Overview

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 `showdemo 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 a sampling rate of `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 a sampling rate of `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 10-12.

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.
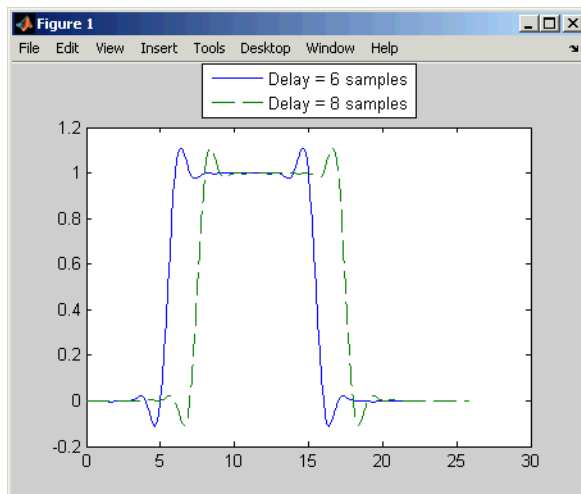
Delays of Six Samples (Solid Line) and Eight Samples (Dashed Line)

10-11

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. 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.

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

In this section...
"Section Overview" on page 10-14
"Sampling Rates" on page 10-14
"Example Designing a Square-Root Raised Cosine Filter" on page 10-14
"Other Options in Filter Design" on page 10-15

Section Overview

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 10-7.

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 page for rcosine explains these parameters. The group delay is also mentioned above in "Noncausality and the Group Delay Parameter" on page 10-2.

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, NJ, 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 Communications Toolbox software in the sections listed below.

Channel Features of the Toolbox

This toolbox supports these types of channels:

- Additive white Gaussian noise (AWGN) channel
- Multiple-Input Multiple-Output (MIMO) channel
- Fading channel
- Binary symmetric channel, for binary signals

These channels accept MATLAB arrays as inputs and output a channel impaired version of the input.

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

In this section...
"Section Overview" on page 11-3
"Describing the Noise Level of an AWGN Channel" on page 11-3

Section Overview

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:

$$E_s/N_0 \text{ (dB)} = 10\log_{10}\left(T_{sym}/T_{samp}\right) + SNR \text{ (dB)} \quad \text{for complex input signals}$$

$$E_s/N_0 \text{ (dB)} = 10\log_{10}\left(0.5T_{sym}/T_{samp}\right) + SNR \text{ (dB)} \text{ for real input signals}$$

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 Hertz
- F_s = Sampling frequency, in Hertz

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

11-5
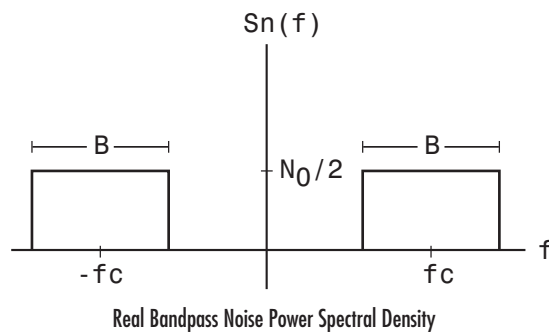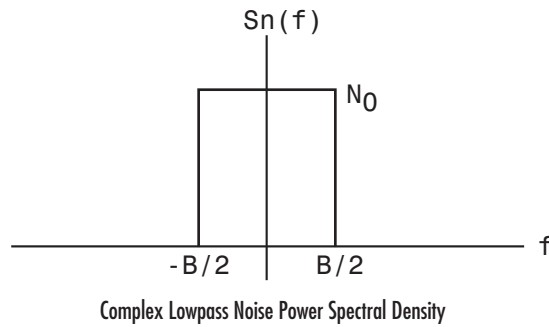
MIMO Channels

The new MIMO channel object supports multiple-input multiple output simulations. You can specify correlated or uncorrelated fading between channels. For more information, see the `mimochan` help page.

Fading Channels

In this section...

Section Overview

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.

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.
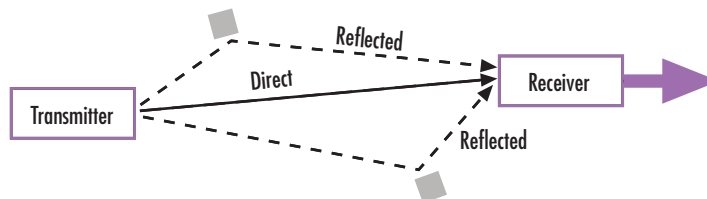
The major paths result in the arrival of delayed versions of the signal at the receiver. In addition, the radio signal undergoes scattering on a *local* scale for each major path. Such local scattering is typically characterized by a large number of reflections by objects near the mobile. These irresolvable 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 nonline-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

- 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 each path.

- By default, each path of the channel is modeled with a Jakes Doppler spectrum, with a maximum Doppler shift that can be specified. Other types of Doppler spectra are allowed (identical or different for all paths): flat, restricted Jakes, asymmetrical Jakes, Gaussian, bi-Gaussian, and rounded.

  If the maximum Doppler shift is set to 0 or omitted during the construction of a channel object, then the channel is modeled as static (i.e., the fading does not evolve with time), and the Doppler spectrum specified has no effect on the fading process.

Some additional information about typical values for delays and gains is in "Choosing Realistic Channel Property Values" on page 11-21.

Simulation of Multipath Fading Channels: Methodology

The Rayleigh and Rician multipath fading channel simulators of this toolbox use the band-limited discrete multipath channel model of section 9.1.3.5.2 in [1]. It is assumed that the delay power profile and the Doppler spectrum of the channel are separable [1]. The multipath fading channel is therefore modeled as a linear finite impulse-response (FIR) filter. Let $\{s_i\}$ denote the set of samples at the input to the channel. Then the samples $\{y_i\}$ at the output of the channel are related to $\{s_i\}$ through:

$$y_i = \sum_{n=-N_1}^{N_2} s_{i-n} g_n$$

where $\{g_n\}$ is the set of tap weights given by:

$$g_n = \sum_{k=1}^{K} a_k \mathrm{sinc}\left[\frac{\tau_k}{T_s} - n\right], \ -N_1 \le n \le N_2$$

In the equations above:

- T_s is the input sample period to the channel.

- $\{\tau_k\}$, where $1 \le k \le K$, is the set of path delays. K is the total number of paths in the multipath fading channel.

- $\{a_k\}$, where $1 \le k \le K$, is the set of complex path gains of the multipath fading channel. These path gains are uncorrelated with each other.

- 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.

Each path gain process a_k is generated by the following steps:

1 A complex uncorrelated (white) Gaussian process with zero mean and unit variance is generated in discrete time.

2 The complex Gaussian process is filtered by a Doppler filter with frequency response $H(f) = \sqrt{S(f)}$, where $S(f)$ denotes the desired Doppler power spectrum.

3 The filtered complex Gaussian process is interpolated so that its sample period is consistent with that of the input signal. A combination of linear and polyphase interpolation is used.

4 The resulting complex process z_k is scaled to obtain the correct average path gain. In the case of a Rayleigh channel, the fading process is obtained as:

$$a_k = \sqrt{\Omega_k}\, z_k$$

where

$$\Omega_k = E\left[\left|a_k\right|^2\right]$$

In the case of a Rician channel, the fading process is obtained as:

$$a_k = \sqrt{\Omega_k}\left[\frac{z_k}{\sqrt{K_{r,k}+1}} + \sqrt{\frac{K_{r,k}}{K_{r,k}+1}}e^{j\left(2\pi\, f_{d,LOS,k}\, t\, +\, \theta_{LOS,k}\right)}\right]$$

where $K_{r,k}$ is the Rician K-factor of the k-th path, $f_{d,LOS,k}$ is the Doppler shift of the line-of-sight component of the k-th path (in Hz), and $\theta_{LOS,k}$ is the initial phase of the line-of-sight component of the k-th path (in rad).

At the input to the band-limited multipath channel model, the transmitted symbols must be oversampled by a factor at least equal to the bandwidth expansion factor introduced by pulse shaping. For example, if sinc pulse shaping is used, for which the bandwidth of the pulse-shaped signal is equal to the symbol rate, then the bandwidth expansion factor is 1, and, in the ideal case, at least one sample-per-symbol is required at the input to the channel. If a raised cosine (RC) filter with a factor in excess of 1 is used, for which the bandwidth of the pulse-shaped signal is equal to twice the symbol rate,

then the bandwidth expansion factor is 2, and, in the ideal case, at least two samples-per-symbol are required at the input to the channel.

References

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

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.

Note Setting the maximum path delay greater than 100 samples may generate an 'Out of memory' error.

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" on page 11-12
- "Viewing Object Properties" on page 11-12
- "Changing Object Properties" on page 11-14
- "Linked Properties of Channel Objects" on page 11-15

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 11-23.

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 to 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
           DopplerSpectrum: [1x1 doppler.jakes]
           MaxDopplerShift: 130
                PathDelays: 0
              AvgPathGaindB: 0
         NormalizePathGains: 1
               StoreHistory: 0
             StorePathGains: 0
                  PathGains: -0.0428 + 0.4732i
          ChannelFilterDelay: 0
       ResetBeforeFiltering: 1
        NumSamplesProcessed: 0

  g =

     -0.0428 + 0.4732i
```

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 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. In the second listing of properties, the `ResetBeforeFiltering` property has the value `0`.

```
c1 =

              ChannelType: 'Rayleigh'
        InputSamplePeriod: 1.0000e-005
          DopplerSpectrum: [1x1 doppler.jakes]
          MaxDopplerShift: 130
               PathDelays: 0
             AvgPathGaindB: 0
         NormalizePathGains: 1
              StoreHistory: 0
            StorePathGains: 0
                 PathGains: 0.5781 + 0.9020i
         ChannelFilterDelay: 0
       ResetBeforeFiltering: 1
        NumSamplesProcessed: 0

c1 =

              ChannelType: 'Rayleigh'
```

```
      InputSamplePeriod: 1.0000e-005
        DopplerSpectrum: [1x1 doppler.jakes]
        MaxDopplerShift: 130
             PathDelays: 0
           AvgPathGaindB: 0
      NormalizePathGains: 1
           StoreHistory: 0
          StorePathGains: 0
              PathGains: 0.5781 + 0.9020i
     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 samples 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.

Specifying the Doppler Spectrum of a Fading Channel

The Doppler spectrum of a channel object is specified through its DopplerSpectrum property. The value of this property must be either:

• A Doppler object. In this case, the same Doppler spectrum applies to each path of the channel object.

- A vector of Doppler objects of the same length as the `PathDelays` vector property. In this case, the Doppler spectrum of each path is given by the corresponding Doppler object in the vector.

A Doppler object contains all the properties used to characterize the Doppler spectrum, with the exception of the maximum Doppler shift, which is a property of the channel object. This section describes how to create and manipulate Doppler objects, and how to assign them to the `DopplerSpectrum` property of channel objects.

Creating Doppler Objects

The sole purpose of Doppler objects is to specify the value of the `DopplerSpectrum` property of channel objects. Doppler objects can be created using one of seven functions: `doppler.ajakes`, `doppler.bigaussian`, `doppler.jakes`, `doppler.rjakes`, `doppler.flat`, `doppler.gaussian`, and `doppler.rounded`. For a description of each of these functions and the underlying theory, refer to their corresponding reference pages.

For example, a Gaussian spectrum with a normalized (by the maximum Doppler shift of the channel) standard deviation of 0.1, can be created as:

```
d = doppler.gaussian(0.1);
```

Duplicating Doppler Objects

As in the case of channel objects, Doppler objects can be duplicated using the `copy` function. The command:

```
d2 = copy(d1);
```

creates a Doppler object `d2` with the same properties as that of `d1`. `d1` and `d2` are then separate instances of a Doppler object, in that modifying either one will not affect the other. Using `d1 = d2` instead will cause both `d1` and `d2` to reference the same instance of a Doppler object, in that modifying either one will cause the same modification to the other.

Viewing and Changing Doppler Object Properties

The syntax for viewing and changing Doppler object properties is the same as for the case of channel objects (see "Viewing Object Properties" on page 11-12

and "Changing Object Properties" on page 11-14). The function disp can be used with Doppler objects to display their properties.

In the following example, a rounded Doppler object with default properties is created and displayed, and the third element of its CoeffRounded property is modified:

```
dr = doppler.rounded

dr =

    SpectrumType: 'Rounded'
    CoeffRounded: [1 -1.7200 0.7850]

dr.CoeffRounded(3) = 0.8250

dr =

    SpectrumType: 'Rounded'
    CoeffRounded: [1 -1.7200 0.8250]
```

Note that the property SpectrumType, which is common to all Doppler objects, is read-only. It is automatically specified at object construction, and cannot be modified. If you wish to use a different Doppler spectrum type, you need to create a new Doppler object of the desired type.

Using Doppler Objects Within Channel Objects

The DopplerSpectrum property of a channel object can be changed by assigning to it a Doppler object or a vector of Doppler objects. The following example illustrates how to change the default Jakes Doppler spectrum of a constructed Rayleigh channel object to a flat Doppler spectrum:

```
>> h = rayleighchan(1/9600, 100)

h =

            ChannelType: 'Rayleigh'
      InputSamplePeriod: 1.0417e-004
        DopplerSpectrum: [1x1 doppler.jakes]
        MaxDopplerShift: 100
```

```
                        PathDelays: 0
                     AvgPathGaindB: 0
                NormalizePathGains: 1
                      StoreHistory: 0
                    StorePathGains: 0
                         PathGains: -0.4007 - 0.2748i
                ChannelFilterDelay: 0
             ResetBeforeFiltering: 1
              NumSamplesProcessed: 0

   >> dop_flat = doppler.flat

   dop_flat =

       SpectrumType: 'Flat'

   >> h.DopplerSpectrum = dop_flat

   h =

                       ChannelType: 'Rayleigh'
                 InputSamplePeriod: 1.0417e-004
                   DopplerSpectrum: [1x1 doppler.flat]
                   MaxDopplerShift: 100
                        PathDelays: 0
                     AvgPathGaindB: 0
                NormalizePathGains: 1
                      StoreHistory: 0
                    StorePathGains: 0
                         PathGains: -0.4121 - 0.2536i
                ChannelFilterDelay: 0
             ResetBeforeFiltering: 1
              NumSamplesProcessed: O
```

The following example shows how to change the default Jakes Doppler
spectrum of a constructed Rician channel object to a Gaussian Doppler
spectrum with normalized standard deviation of 0.3, and subsequently display
the DopplerSpectrum property, and change the value of the normalized
standard deviation to 1.1:

```
>> h = ricianchan(1/9600, 100, 2);
>> h.DopplerSpectrum = doppler.gaussian(0.3)

h =

                ChannelType: 'Rician'
          InputSamplePeriod: 1.0417e-004
            DopplerSpectrum: [1x1 doppler.gaussian]
            MaxDopplerShift: 100
                 PathDelays: 0
               AvgPathGaindB: 0
                    KFactor: 2
     DirectPathDopplerShift: 0
        DirectPathInitPhase: 0
          NormalizePathGains: 1
                StoreHistory: 0
              StorePathGains: 0
                   PathGains: 0.8073 - 0.0769i
           ChannelFilterDelay: 0
        ResetBeforeFiltering: 1
          NumSamplesProcessed: 0

>> h.DopplerSpectrum

ans =

     SpectrumType: 'Gaussian'
     SigmaGaussian: 0.3000

>> h.DopplerSpectrum.SigmaGaussian = 1.1;
```

The following example illustrates how to change the default Jakes Doppler spectrum of a constructed three-path Rayleigh channel object to a vector of different Doppler spectra, and then change the properties of the Doppler spectrum of the third path:

```
>> h = rayleighchan(1/9600, 100, [0 1e-4 2.1e-4]);
>> h.DopplerSpectrum = [doppler.flat doppler.flat doppler.rounded]

h =
```

```
            ChannelType: 'Rayleigh'
      InputSamplePeriod: 1.0417e-004
        DopplerSpectrum: [3x1 doppler.baseclass]
        MaxDopplerShift: 100
             PathDelays: [O 1.0000e-004 2.1000e-004]
           AvgPathGaindB: [O O O]
      NormalizePathGains: 1
            StoreHistory: O
          StorePathGains: O
               PathGains: [0.4233 - 0.1113i -0.0785 + 0.1667i
                           -0.2064 + 0.3531i]
      ChannelFilterDelay: 3
   ResetBeforeFiltering: 1
     NumSamplesProcessed: O

>> h.DopplerSpectrum(3).CoeffRounded = [1 -1.21 0.7];
```

If the `DopplerSpectrum` property of a channel object is a vector:

- If the length of the `PathDelays` vector property is increased, the length of `DopplerSpectrum` is automatically increased to match the length of `PathDelays`, by appending Jakes Doppler objects.

- If the length of the `PathDelays` vector property is decreased, the length of `DopplerSpectrum` is automatically decreased to match the length of `PathDelays`, by removing the last Doppler object(s).

Configuring Channel Objects

Before you filter a signal using a channel object, make sure 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 11-21

- "Configuring Channel Objects Based on Simulation Needs" on page 11-23

The syntaxes for viewing and changing values of properties of channel objects are described in "Specifying Fading Channels" on page 11-11.

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.

Note Setting the maximum path delay greater than 100 samples may generate an 'Out of memory' error.

• 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 calculated as follows, where *f* is the transmission carrier frequency in Hertz and *c* is the speed of light (3e8 m/s).

$$f_d = \frac{vf}{c}$$

- Based on this formula 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.

Doppler Spectrum Parameters

- See the reference pages for the respective Doppler objects for descriptions of the parameters and their significance.

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), 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, 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, 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, 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 11-11, 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

(`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 11-25.

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 is populated with plot information. See "Using the Channel Visualization Tool" on page 11-34 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 12, "Equalizers" to learn how to implement equalizers in this toolbox. See the `dpskmod` reference page or the example in "Comparing Empirical Results to Theoretical Results" on page 11-26 to learn how to implement differential modulation.

Examples Using Fading Channels

The following examples use fading channels:

• "Power of a Faded Signal" on page 11-25

• "Comparing Empirical Results to Theoretical Results" on page 11-26

• "Working with Delays" on page 11-28

• "Quasi-Static Channel Modeling" on page 11-29

• "Filtering Using a Loop" on page 11-32

• "Storing Channel State History" on page 11-33

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 = 1i*ones(2000,1);  % Generate signal
y = filter(c,sig);       % Pass signal through channel
c                        % Display all properties of the channel

% Plot power of faded signal, versus sample number.
plot(20*log10(abs(y)))
```

The output and the plot follow.

```
c =

              ChannelType: 'Rayleigh'
        InputSamplePeriod: 1.0000e-004
           DopplerSpectrum: [1x1 doppler.jakes]
           MaxDopplerShift: 100
                PathDelays: 0
              AvgPathGaindB: 0
         NormalizePathGains: 1
               StoreHistory: 0
             StorePathGains: 0
                  PathGains:  -0.8062 + 0.2648i
          ChannelFilterDelay: 0
        ResetBeforeFiltering: 1
         NumSamplesProcessed: 2000
```
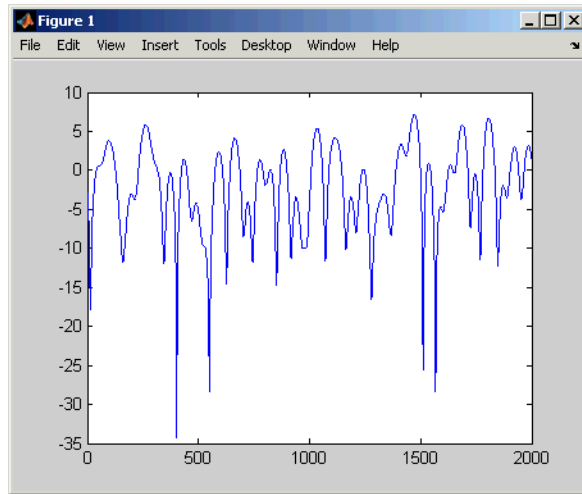
Comparing Empirical Results to Theoretical Results

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
hMod = modem.dpskmod('M', M);       % Create a DPSK modulator
hDemod = modem.dpskdemod(hMod);     % Create a DPSK demodulator
                                    % using the modulator
tx = randi([0 M-1],50000,1);        % Generate a random bit stream
dpskSig = modulate(hMod, tx);       % DPSK modulate the signal
fadedSig = filter(chan,dpskSig);    % Apply the channel effects

% Compute error rate for different values of SNR.
SNR = 0:2:20; % Range of SNR values, in dB.
numSNR = length(SNR);
```

```
BER = zeros(1, numSNR);
for n = 1:numSNR
   rxSig = awgn(fadedSig,SNR(n));   % Add Gaussian noise
   rx = demodulate(hDemod, rxSig);  % Demodulate
   reset(hDemod);
   % Compute error rate.
   [nErrors, BER(n)] = biterr(tx,rx);
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');
```

With the parameters in the preceding code, the fading is slow enough to be
considered the same across two consecutive samples.

The resulting plot shows that the simulation results are close to the
theoretical results computed by berfading.
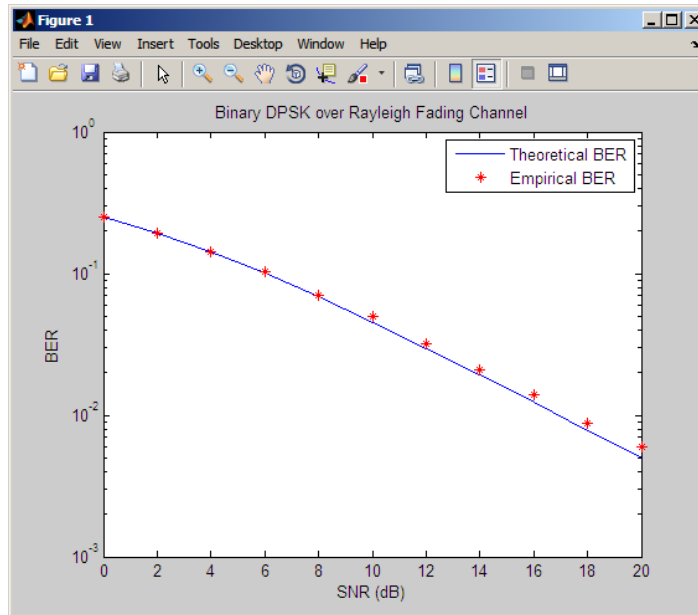
Working with Delays

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, 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;
hMod = modem.dpskmod('M', M);   % Create a DPSK modulator
hDemod = modem.dpskdemod(hMod); % Create a DPSK demodulator
                                  % using the modulator

% Create Rayleigh fading channel object.
ch = rayleighchan(1/bitRate,4,[0 0.5/bitRate],[0 -10]);
delay = ch.ChannelFilterDelay;
```

```
tx = randi([0 M-1],50000,1);      % Generate random bit stream
dpskSig = modulate(hMod,tx);       % DPSK modulate signal
fadedSig = filter(ch,dpskSig);     % Apply channel effects
rx = demodulate(hDemod,fadedSig);  % Demodulate signal

% Compute bit error rate, taking delay into account.
% 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, the error rate would have been close to 1/2.

```
num =

    845


ber =

    0.0170
```

**More Information About Working with Delays.** The discussion in "Effect of Delays on Recovery of Convolutionally Interleaved Data" on page 8-10 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 entails 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 these steps 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
hMod = modem.dpskmod('M', M);   % Create a DPSK modulator
hDemod = modem.dpskdemod(hMod); % Create a DPSK demodulator
                                    % using the modulator
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 Rician 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 = randi([0 M-1],numBits,1);     % Generate random bit stream
   dpskSig = modulate(hMod, tx);      % DPSK modulate signal
   fadedSig = filter(chan, dpskSig);  % Apply channel effects
   rxSig = awgn(fadedSig,20);         % Add Gaussian noise.
   rx = demodulate(hDemod,rxSig);     % 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, 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 11-23 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.
scatterPlot = commscope.ScatterPlot;

% Apply channel in a loop, maintaining continuity.
% Plot only the current data in each iteration.
for n = 1:numTrials
   tx = randi([0 M-1],500,1);    % Generate random bit stream
   pskSig = pskmod(tx,M);        % PSK modulate signal
   fadedSig = filter(ch, pskSig); % Apply channel effects
```

```
    % Plot the new data from this iteration.
    update(scatterPlot,fadedSig);
end
```

The scatter plot changes with each iteration of the loop, and the exact content varies because the fading process involves random numbers. Following are some snapshots of typical images that the example can produce.
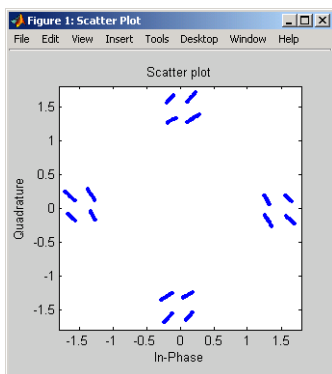
Sample Scatter Plot (a)
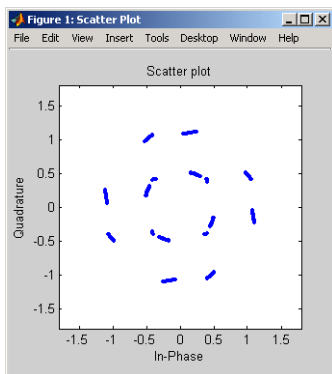
Sample Scatter Plot (b)

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` makes 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
```

This example generates the following output:

```
ans =

  -0.0460 - 1.1873i
  -0.0439 - 1.1881i
  -0.0418 - 1.1889i
  -0.0397 - 1.1897i
  -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.

Setting `StoreHistory` to `true` significantly slows down the execution speed of the channel's `filter` function.

Using the Channel Visualization Tool

Communications Toolbox software provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See "Fading Channels" on page 11-7 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
```
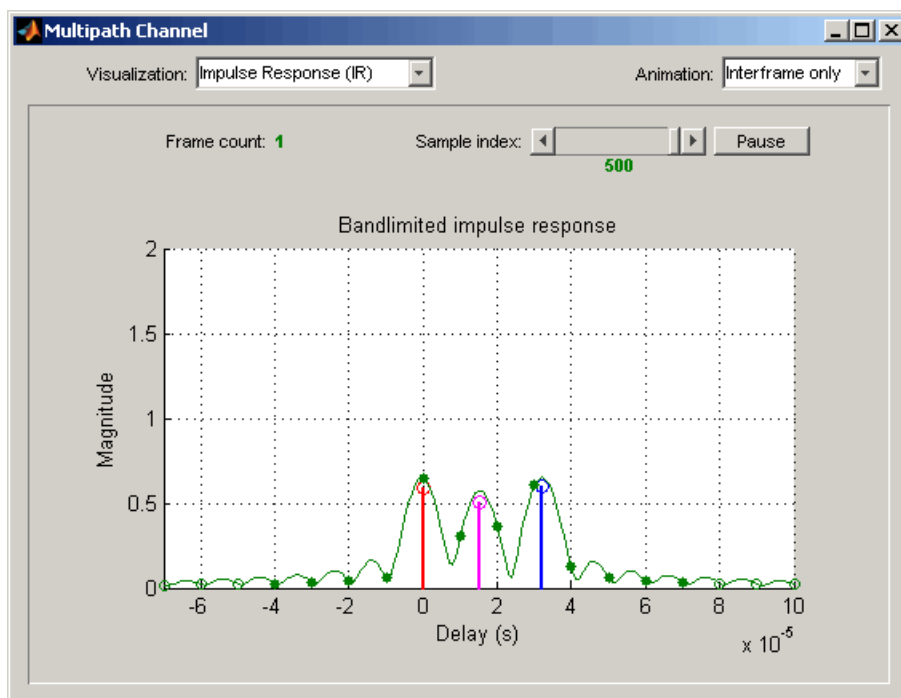

See "Examples of Using the Channel Visualization Tool" on page 11-46 for the basic usage cases of the channel visualization tool.

This tool can also be accessed from Communications Blockset software.

Parts of the GUI

The **Visualization** pull-down menu allows you to choose the visualization method. See "Visualization Options" on page 11-36 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 updates 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.
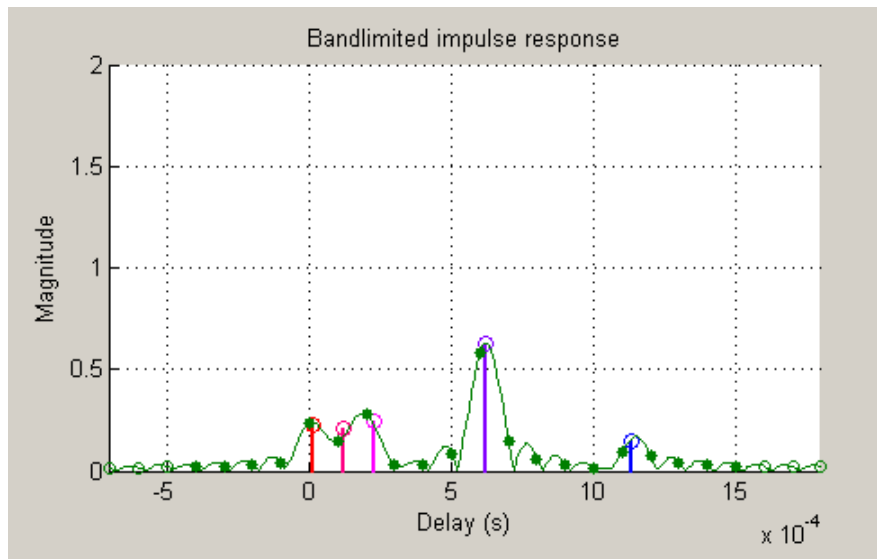
The multipath response is represented by stems, each corresponding to one multipath component. The component with the smallest delay value is shown in red, and the component with the largest delay value is shown in blue. Components with intermediate delay values are shades between red and blue, becoming 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.
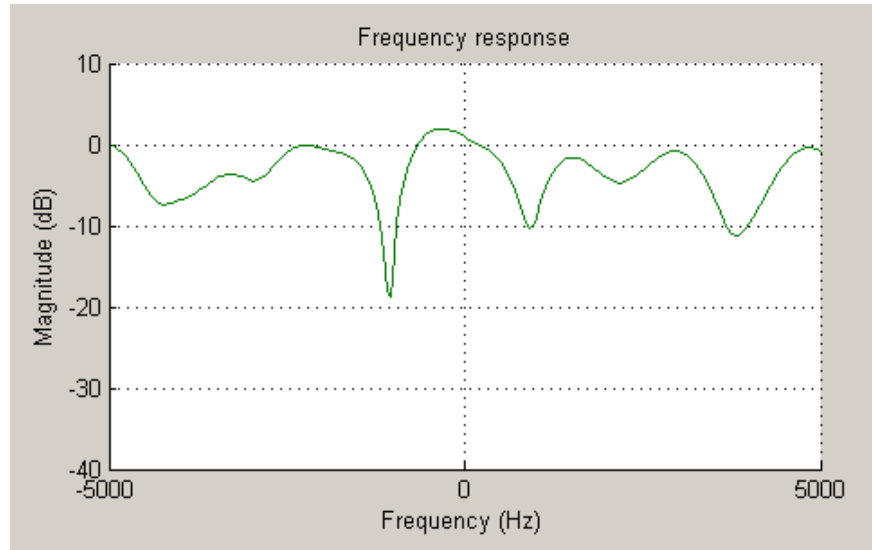
As with the impulse response visualization, you can visualize how this frequency response changes over time.

IR Waterfall. This plot shows the evolution of the magnitude impulse response over time.
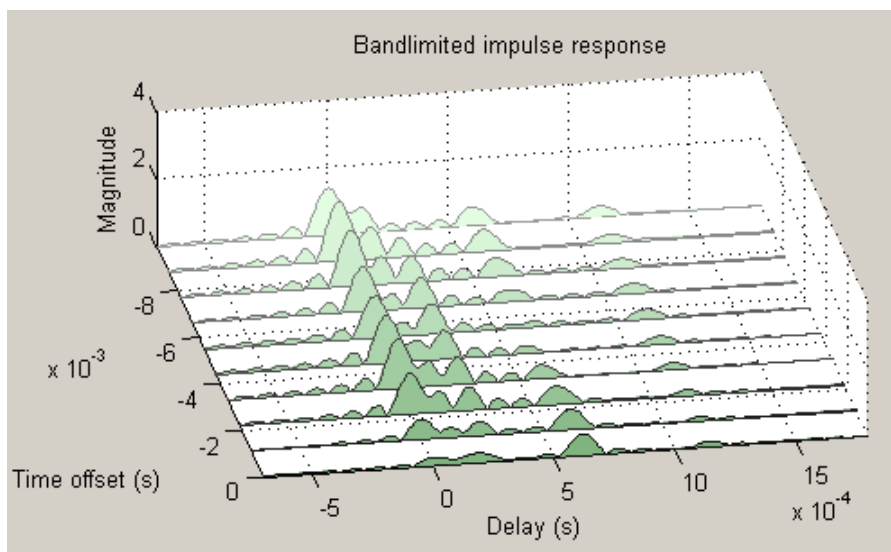
It shows 10 snapshots of the bandlimited channel impulse response within the last frame, with the darkest green curve showing the current response.

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.
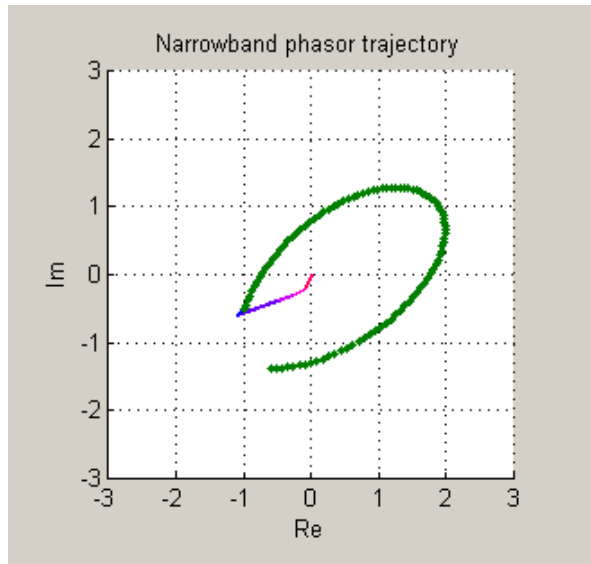
The phasors are connected end to end in order of path delay, and the trajectory of the resultant phasor is plotted as a green line. This resultant phasor is referred to as the *narrowband phasor*.

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.
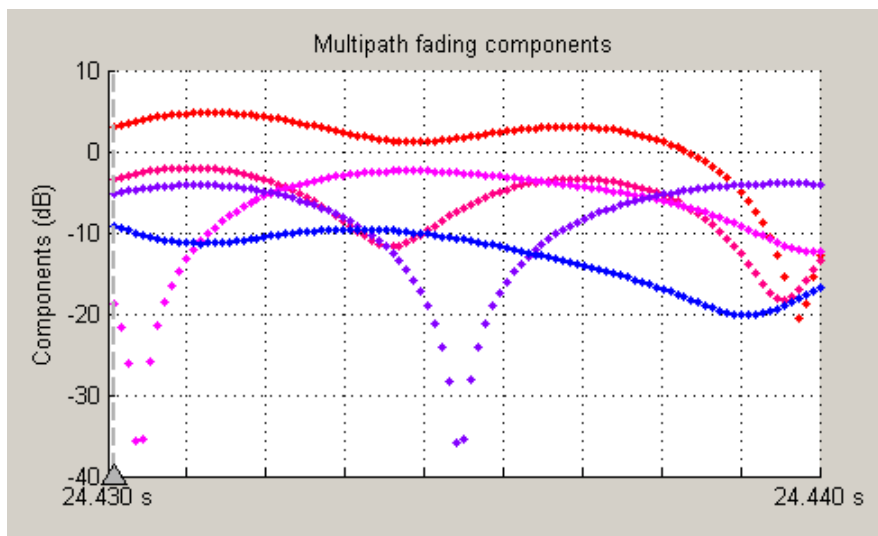
The triangle marker and vertical dashed line represent the start of the current frame. If a frame has been processed previously, its multipath gains may also be displayed.

Multipath Gain. This plot shows the collective gains for the multipath channel for three signal bandwidths.
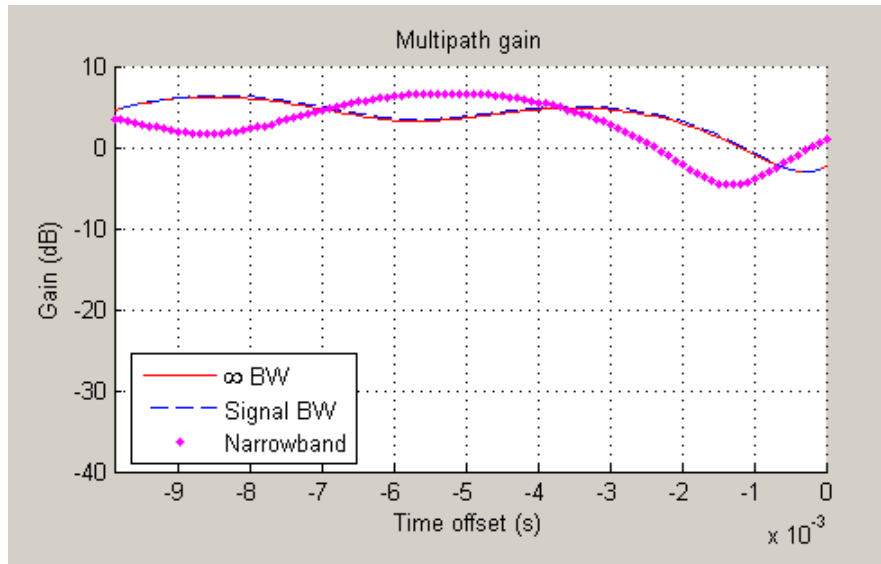
A collective gain is the sum of component magnitudes, as explained in the following:

- 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.
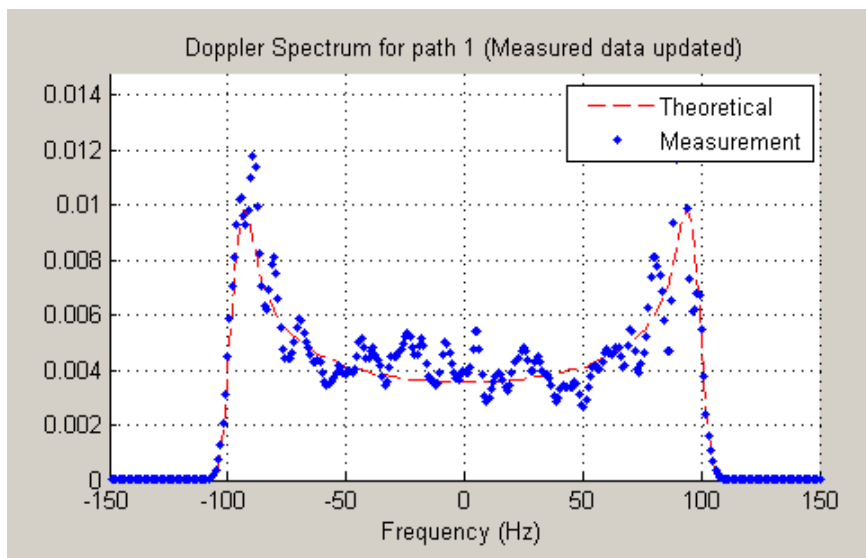
The first Doppler spectrum, represented by the dashed red line, is a theoretical spectrum based on the Doppler filter response used in the multipath channel model. In the preceding plot, the theoretical Doppler spectrum used for the multipath channel model is known as the *Jakes spectrum*. Note that the plotted Doppler spectrum is normalized to have a total power of 1. 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 need to be processed through the channel before either the first
Doppler spectrum or an updated spectrum can be plotted.

The **Path Number** edit box allows you to visualize the Doppler spectrum of
the specified path. The value entered in this box must be a valid path number,
i.e., between 1 and the length of the `PathDelays` vector property. Once you
change the value of this field, the new Doppler spectrum will appear as soon
as the processing of the current frame has ended.

If the measured Doppler spectrum is a good approximation of the theoretical
Doppler spectrum, 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.

It is possible that a multipath channel (e.g., a Rician channel) can have both
specular (line-of-sight) and diffuse components. In such a case, the Doppler
spectrum would have both a line component and a wideband component.
The channel visualization tool only shows the wideband component for the
Doppler spectrum.

Unlike other visualizations, the Doppler spectrum visualization does
not support animation. Because there is no intraframe data to plot, the
visualization tool only updates the channel statistics at the end of each frame
and therefore cannot pause in the middle of a frame. If you switch to the
Doppler spectrum visualization from a different visualization that is in pause
mode, the **Pause** button is subsequently disabled. Disabling pause avoids
interaction problems between the Doppler spectrum visualization and other
animation-style visualizations.

Scattering Function. This plot shows the Doppler spectra of each path
versus the path delays, using the same color code as that used for the
multipath impulse response.
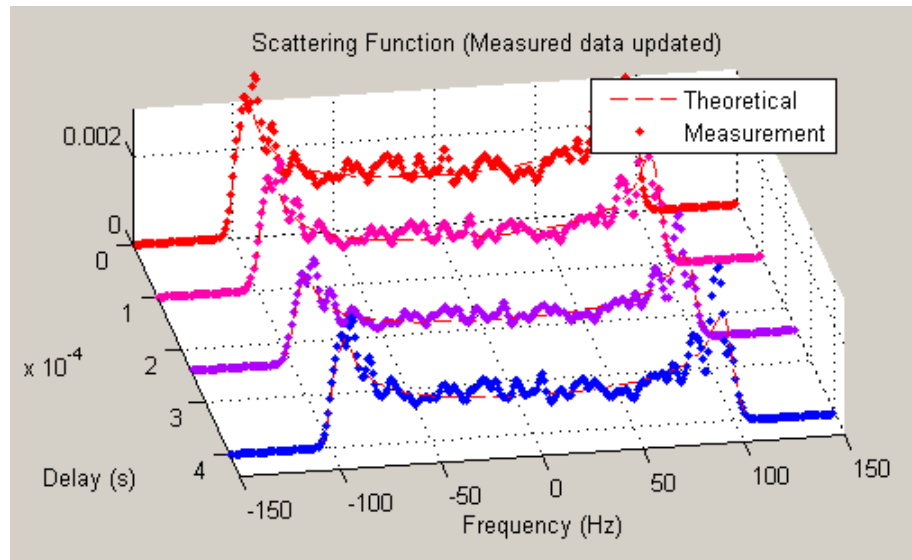
The principle of operation of the Scattering Function plot is similar to that of the Doppler Spectrum plot. The main difference is that the Doppler spectra on this plot are not normalized as they are on the Doppler Spectrum plot, in order to better visualize the power delay profile.

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 11-46
- "Animating Snapshots Across Frames" on page 11-46

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 = 1;        % 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 click **Resume**. 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 = 1;
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 clicking **Resume**, the plot continues to animate.

The property ResetBeforeFiltering needs to be set to false so that the state information in the channel is not reset after the processing of each frame.

Binary Symmetric Channel

Section Overview

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.

To model a binary channel whose statistical description involves the number of errors per codeword, 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 =

    158
```

```
numsyserrs =

    53


ber =

    0.0053
```

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, John Wiley & Sons, 1993.

12

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 Communications Toolbox software in the sections listed below.

Equalizer Features of Communications Toolbox Software

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 (FSEs)
- Decision-feedback equalizers (DFEs)
- 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 12-28.

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 12-8. For brief background material on the supported adaptive equalizer types, see "Overview of Adaptive Equalizer Classes" on page 12-3. For more detailed background material, see the works listed in "Selected Bibliography for Equalizers" on page 12-36.

Overview of Adaptive Equalizer Classes

In this section...
"Section Overview" on page 12-3
"Symbol-Spaced Equalizers" on page 12-3
"Fractionally Spaced Equalizers" on page 12-5
"Decision-Feedback Equalizers" on page 12-6

Section Overview

This section gives 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 12-36. 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.
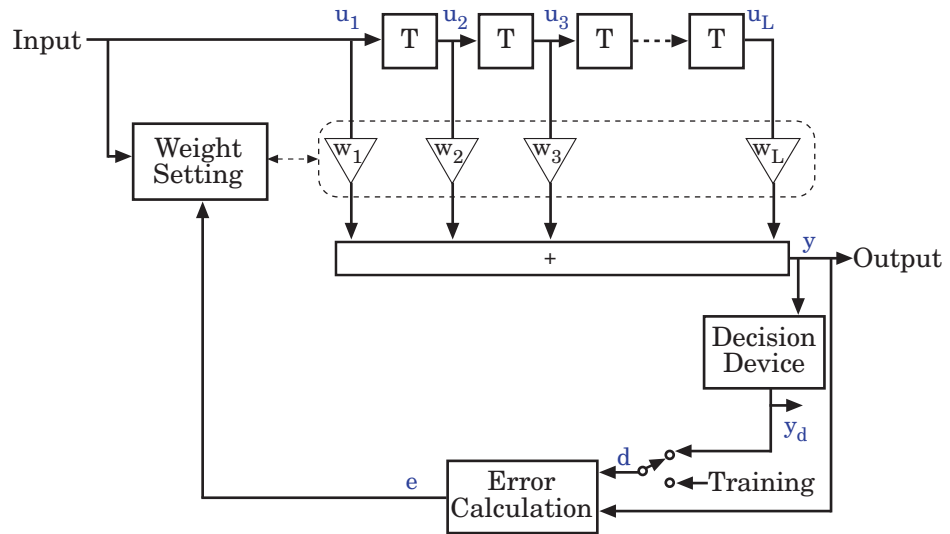
Updating the Set of Weights

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 Communications Toolbox Software" on page 12-2. 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 12-3. 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.
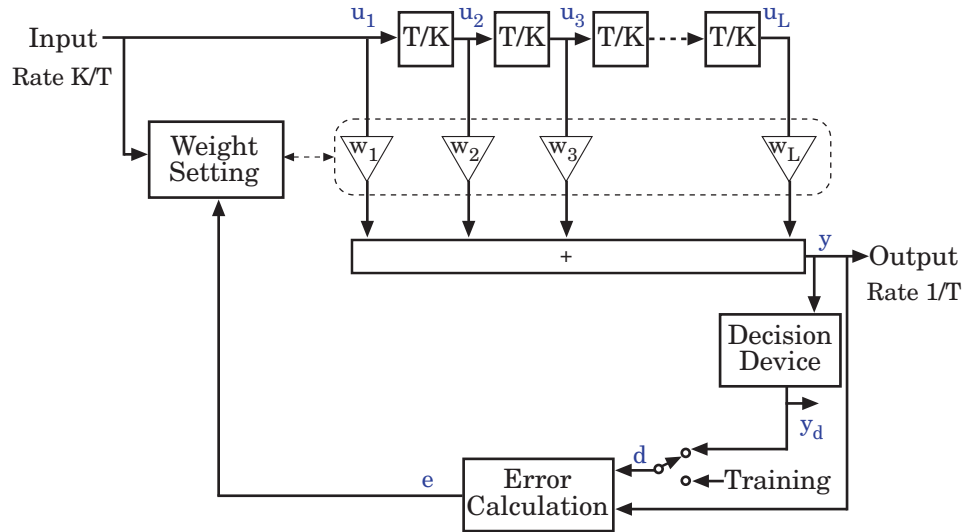
Decision-Feedback Equalizers

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 12-3, 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, the result is a symbol-spaced DFE instead of a fractionally spaced DFE.
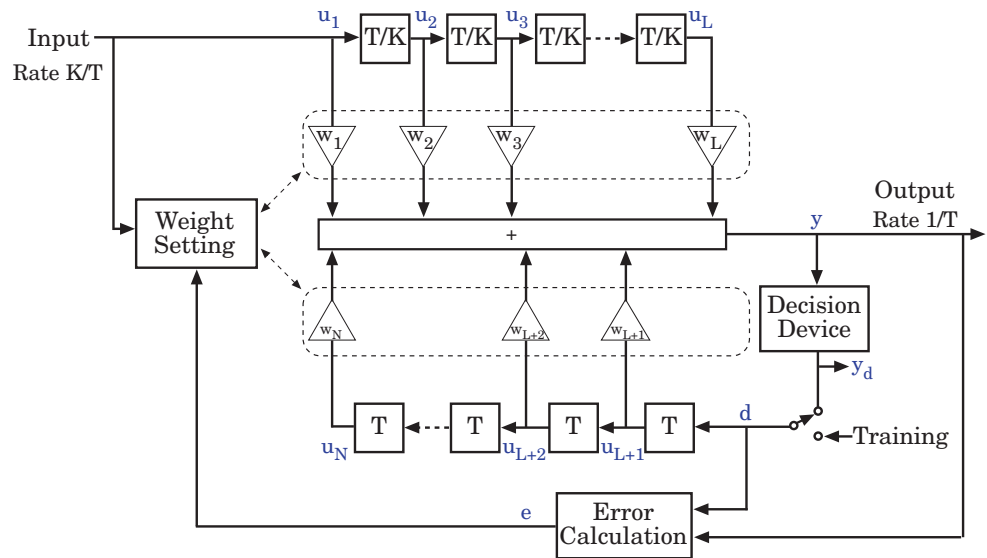
In each symbol period, the equalizer receives K input samples at the forward filter, as well as one decision or training sample at the feedback filter. The equalizer then outputs a weighted sum of the values in the forward and feedback delay lines, and updates the weights to prepare for the next symbol 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

In this section...
"Section Overview" on page 12-8
"Basic Procedure for Equalizing a Signal" on page 12-8
"Example Illustrating the Basic Procedure" on page 12-8
"Learning More About Adaptive Equalizer Functions" on page 12-9

Section Overview

This section gives an overview of the process you typically use in the MATLAB environment to take advantage of the adaptive equalizer capabilities. The MLSE equalizer has a different interface, described in "Using MLSE Equalizers" on page 12-28.

Basic Procedure for Equalizing a Signal

Equalizing a signal using Communications Toolbox software 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 you want to equalize, using the `equalize` method of the equalizer object.

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 eight 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, 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

In this section...
"Choosing an Adaptive Algorithm" on page 12-10
"Indicating a Choice of Adaptive Algorithm" on page 12-11
"Accessing Properties of an Adaptive Algorithm" on page 12-12

Choosing an Adaptive Algorithm

Configuring an equalizer involves choosing an adaptive algorithm and indicating your choice when creating an equalizer object in the MATLAB environment.

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 12-36.

Indicating a Choice of Adaptive Algorithm

After you have chosen the adaptive algorithm you want to use, indicate your choice when creating the equalizer object mentioned in "Basic Procedure for Equalizing a Signal" on page 12-8. 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** To create an adaptive algorithm object by duplicating an existing one and then changing its properties, see the important note in "Duplicating and Copying Objects" on page 12-14 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 they also let you specify certain properties of the algorithm. For information about what each property of an adaptive algorithm object means, see the reference pages for the `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma` functions.

To view or change any properties of an adaptive algorithm, use the syntax described for channel objects in "Viewing Object Properties" on page 11-12 and "Changing Object Properties" on page 11-14.

Specifying an Adaptive Equalizer

In this section...
"Defining an Equalizer Object" on page 12-13
"Accessing Properties of an Equalizer" on page 12-14

Defining an Equalizer Object

As mentioned in "Basic Procedure for Equalizing a Signal" on page 12-8, you must create an equalizer object before you can equalize a signal.

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 three weights in the feedforward filter and two 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 12-10.

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 12-17.

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 11-12 and "Changing Object Properties" on page 11-14.

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 fails 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`, 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. 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

In this section...

Section Overview

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 12-13.

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, you should not include a training sequence as an input vector.

The following code 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 on the plot that the points of the equalized signal are clustered more closely around the points of the signal constellation.
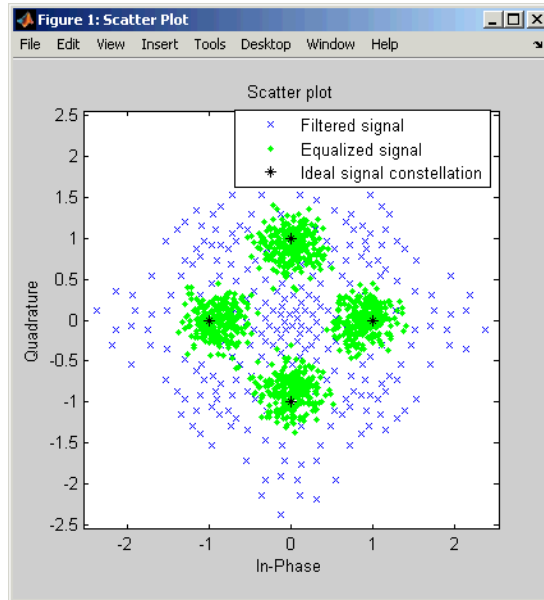
Equalizing in Decision-Directed Mode

Decision-directed mode means 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 12-17) 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 12-17 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, 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, and 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 8-9.

```
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), 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 12-23
- "Simulating the System Using a Loop" on page 12-24

If you want to equalize iteratively while potentially changing equalizers between iterations, see "Changing the Equalizer Between Iterations" on page 12-26 for help generalizing 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 this example uses random numbers.
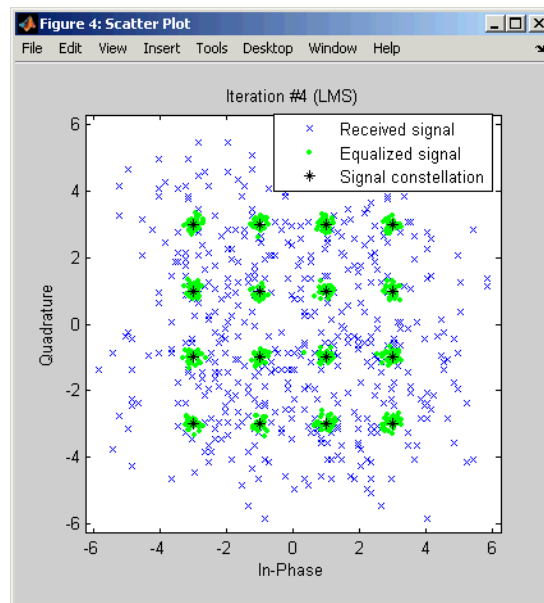
Procedures for Equalizing Within a Loop

This section describes two procedures for equalizing within a loop. The first procedure uses the same equalizer in each iteration, and 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 12-20. 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 12-23:

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 you want to use for the first iteration. Use = to establish the connection so that the two objects get updated together:

```
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

In this section...

Section Overview

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.

For background material about MLSE equalizers, see the works listed in "Selected Bibliography for Equalizers" on page 12-36.

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 12-30.

- 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 12-33.

If you are not processing a series of vectors and do not need to specify a preamble or postamble, 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 12-30.

Equalizing in Continuous Operation Mode

If your data is partitioned into a series of vectors (that you process within a loop, for example), 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 12-30. 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 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 12-31 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 12-31

- "Simulating the System Using a Loop" on page 12-32

- "Computing an Error Rate and Plotting Results" on page 12-32

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. 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 follow.

```
numsymerrs =

     0
```
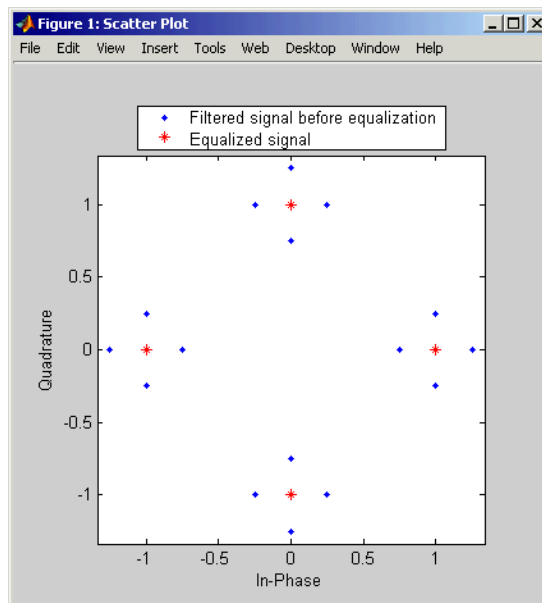

Using a Preamble or Postamble

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, use a postamble vector of [] when invoking mlseeq. Similarly, if your system uses a postamble without a preamble, use a preamble vector of [].

Example: Using a Preamble

The example below illustrates how to accommodate a preamble when using mlseeq. The same preamble symbols appear at the beginning of the message vector and in the syntax for mlseeq. If you want to use a postamble, you can 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, John Wiley & Sons, 1998.

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

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

[5] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, John Wiley & Sons, 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.

If you need to use Galois fields having an odd number of elements, see Galois Fields of Odd Characteristic in the Communications Toolbox online documentation.

For more details about specific functions that process arrays of Galois field elements, see the online reference pages in the documentation for MATLAB or for Communications Toolbox software. MATLAB functions whose generalization to Galois fields is straightforward to describe do not

have reference pages in this manual because the entries would be identical to those in the MATLAB documentation.

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). 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

Section Overview

This section describes how to create a *Galois array*, which is a MATLAB expression that represents the elements of a Galois field. This section also describes how MATLAB technical computing software interprets the numbers that you use in the representation, and includes several examples.

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 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, gf uses a default primitive polynomial for GF(2^m). For information about this argument, see "Specifying the Primitive Polynomial" on page 13-10.

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, 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, MATLAB interprets both as Galois arrays in the same field. 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 13-15.

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. 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, and 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). The output shows 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

Integer Representation	Binary Representation	Element of GF(8)
2	010	A
3	011	A + 1
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 among

- 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 13-4 by describing how to specify your own primitive polynomial when you create a Galois array. The topics are

If you perform many computations using a nondefault primitive polynomial, see "Speed and Nondefault Primitive Polynomials" on page 13-41.

Specifying the Primitive Polynomial

The discussion in "How Integers Correspond to Galois Field Elements" on page 13-8 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 13-8. 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

In this section...
"Section Overview" on page 13-14
"Example: Addition and Subtraction" on page 13-15
"Example: Multiplication" on page 13-16
"Example: Division" on page 13-17
"Example: Exponentiation" on page 13-18
"Example: Elementwise Logarithm" on page 13-19

Section Overview

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	^

For multiplication and division of polynomials over a Galois field, see "Addition and Subtraction of Polynomials" on page 13-33.

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 -, is 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. 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 13-7 for more details.

Logical Operations in Galois Fields

In this section...
"Section Overview" on page 13-20
"Testing for Equality" on page 13-20
"Testing for Nonzero Values" on page 13-21

Section Overview

You can apply logical tests to Galois arrays and obtain a logical array. Some important types of tests are testing for the 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, MATLAB technical computing software 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, use the built-in `isequal` function. However, `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

In this section...
"Basic Manipulations of Galois Arrays" on page 13-23
"Basic Information About Galois Arrays" on page 13-24

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 are 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

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
```

13-25

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, the output x is the unique solution to the
  linear equation.

- If A is square and singular, the syntax in the table produces an error.

- If A is not square, 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, 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

In this section...

Section Overview

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])
```
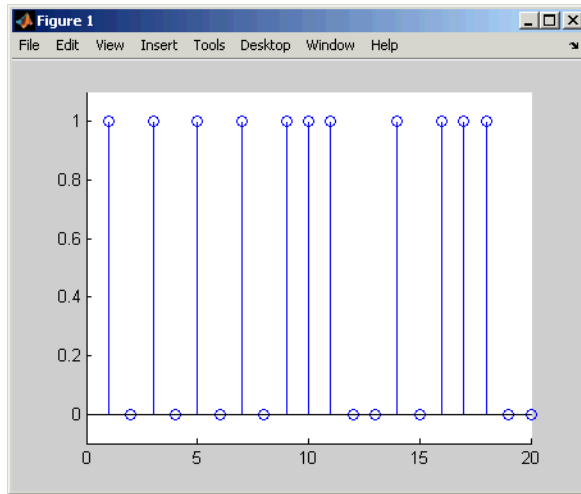
Convolution

Communications Toolbox software 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 13-34. 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, 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), it might be faster to use `dftmtx` once and matrix multiplication many times, instead of using `fft` many times.

Polynomials over Galois Fields

Section Overview

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, you must pad the shorter vector with zeros at the beginning so 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 13-14 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 $GF(2^m)$, 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 is 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 the 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

In this section...
"Section Overview" on page 13-38
"Determining Whether a Variable Is a Galois Array" on page 13-38
"Extracting Information from a Galois Array" on page 13-38

Section Overview

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 13-8.

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.
```

Converting Galois Array to Doubles

```
a = gf([1,0])
b = double(a.x) %a.x is in uint16
```

MATLAB returns the following:

```
a = GF(2) array.

Array elements =

          1          0


b =

    1    0
```

Speed and Nondefault Primitive Polynomials

The section "Specifying the Primitive Polynomial" on page 13-10 described how to 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 the MATLAB application to a folder to which you have write permission. You can use either the `cd` function or the Current Folder 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 folder 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`, 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, 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, MA, Addison-Wesley, 1983, p. 105.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, MA, Addison-Wesley, 1993.

[3] Lin, Shu, and Daniel J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ, 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, NJ, Prentice Hall, 1995.

14

EyeScope: An Eye Diagram Analysis Tool

This section describes the EyeScope Tool and provides a tutorial on how to perform essential end-user tasks.

Introduction

Use EyeScope to examine the data in the eye diagram object. EyeScope shows both the eye diagram plot and measurement results in a unified, graphical environment. You can import, and compare measurement results for, multiple eye diagram objects.

For a description of eye diagrams, refer to 'Eye Diagrams' in the *Communications Toolbox User's Guide*.

For an explanation about constructing an eye diagram object, running a simulation, and analyzing the simulated data, refer to the 'Eye Diagram Measurements' demo.

For a complete list of EyeScope measurements definitions, refer to 'Measurements' in the *Communications Toolbox User's Guide*.

For instructions on how to perform basic EyeScope tasks, see the `EyeScope` reference page.

EyeScope Tutorial

This section provides a step-by-step introduction for using EyeScope to import eye diagram objects, select and change which eye diagram measurements EyeScope displays, compare measurement results, and print a plot object.

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

1 Type `load commeye_EyeMeasureDemoData` at the MATLAB command line to load nine eye diagram objects into the MATLAB workspace.

2 Type `eyescope` at the MATLAB command line to start the EyeScope tool.

3 In the EyeScope window, select **File > Import Eye Diagram Object**.

The **Import eye diagram object** dialog box opens.
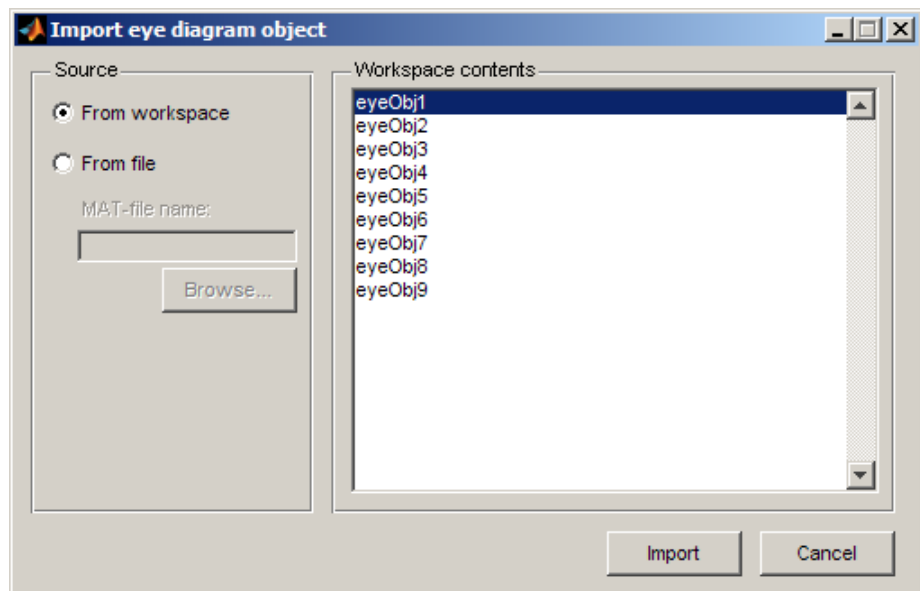
In this window, the **Workspace contents** panel displays all eye diagram objects available in the source location.

4 Select **eyeObj1** and click **Import**. EyeScope imports the object, displaying an image in the object plot and listing the file name in the **Eye diagram objects** list.

Note Object names associated with eye diagram objects that you import from the work space begin with the prefix *ws*.
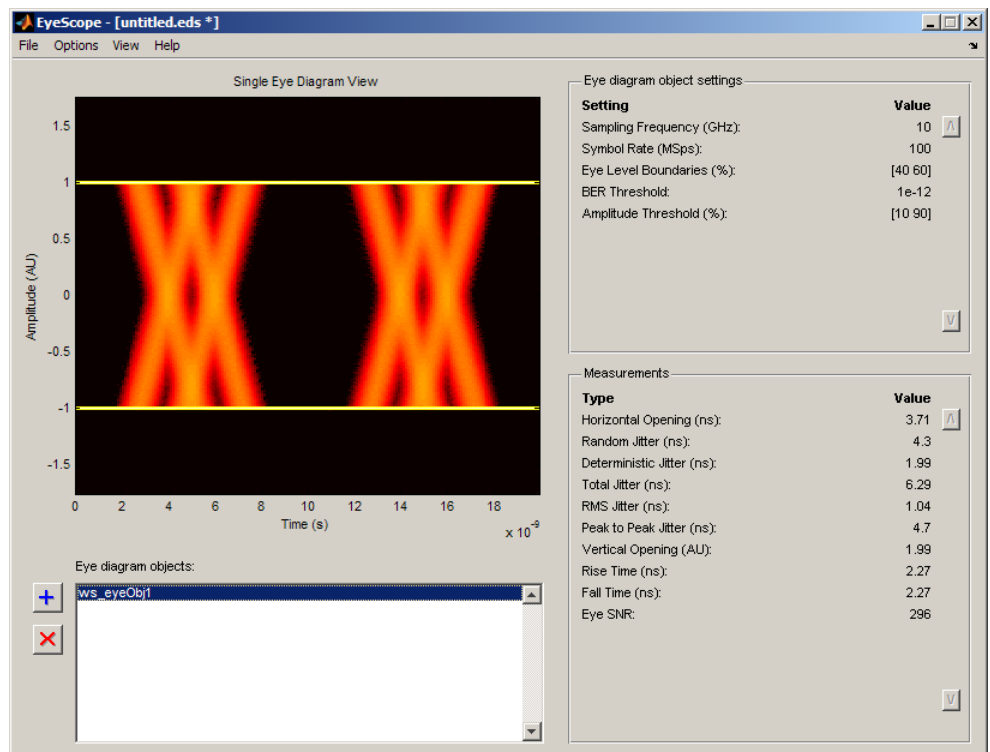
Review the image and note the default **Eye diagram object settings** and **Measurements** selections. For more information, refer to the EyeScope reference page.

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

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

- The EyeScope window changes, displaying a new plot and adding ws_eyeObj5 to the **Eye diagram objects** list. EyeScope displays the same settings and measurements for both eye diagram objects.

- You can switch between the eyediagram plots EyeScope displays by clicking on an object name in the Eye diagram object list.

- Next, click **ws_eyeObj1** and note the EyeScope plot and measurement values changes.

7 To change or remove measurements from the EyeScope display:

- Select **Options > Measurements View**. The **Configure measurement view** shuttle control opens.
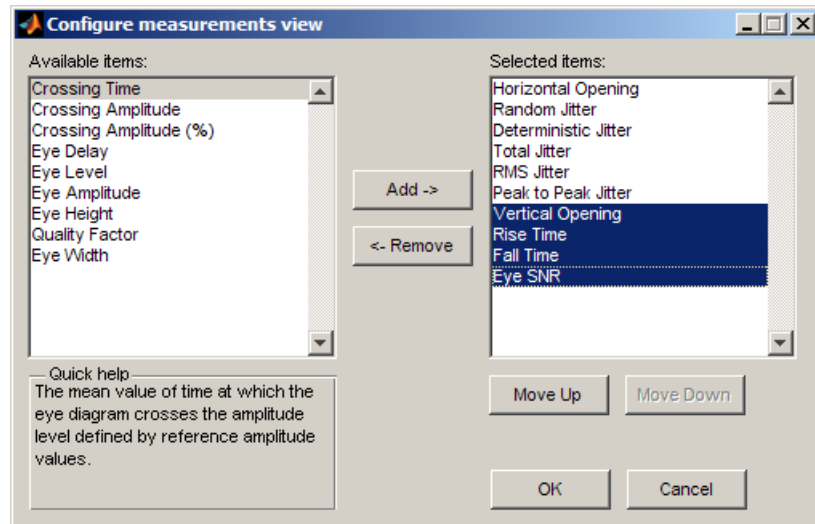
- Hold down the <Ctrl> key and click to select Vertical Opening, Rise Time, Fall Time, Eye SNR. Then click **Remove**.

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

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

- Select **Options > Measurements View**.

- When the **Configure measurement view** shuttle control opens, hold down the <Ctrl> key and click to select **Crossing Time** and **Crossing Amplitude**.

- Click the **Move Up** button until these selections appear at the top of the list. Then, click **OK**

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

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

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

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

14 To compare measurement results for multiple eye diagram objects, click **View > Compare Measurement Results View**.
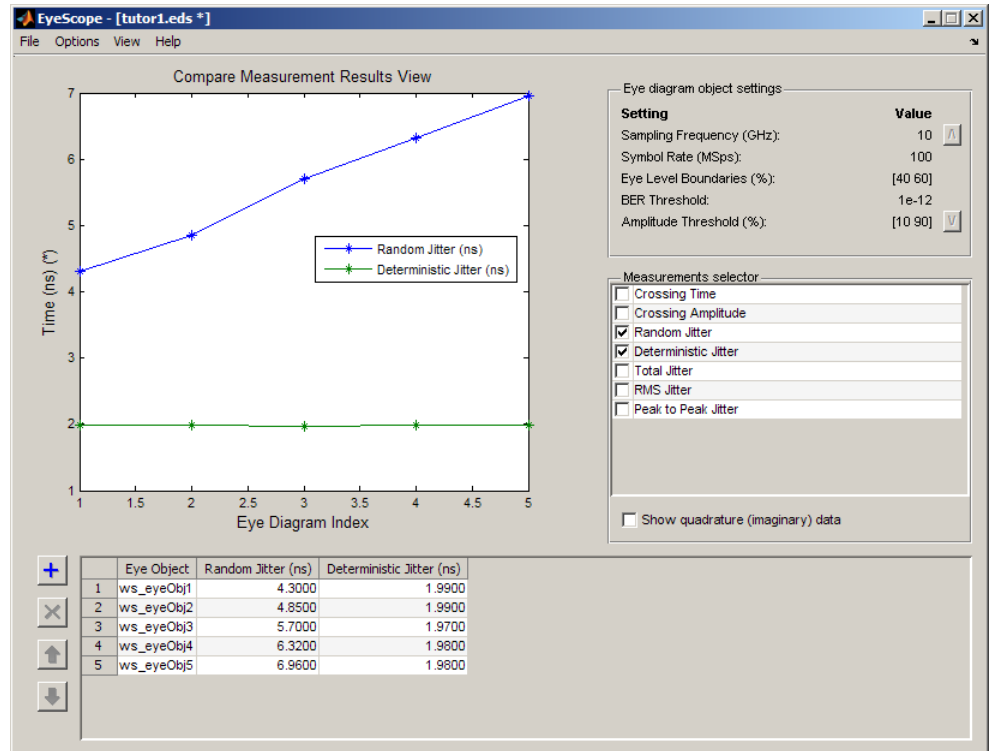
In the data set, random jitter increases from experiment 1 to experiment 5, as you can see in both the table and plot figure.

15 To include any data from the Measurements selection you chose earlier in this procedure, use the **Measurement selector**. Go to the **Measurement selector** and select, the **Total Jitter** check box. The object plot updates to display the additional measurements.
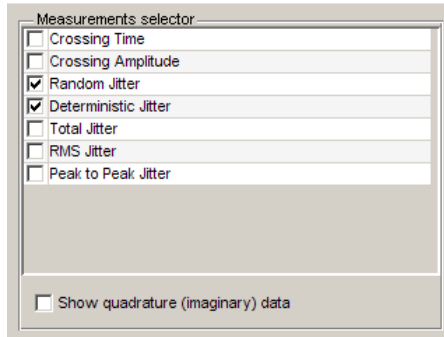
You can also remove measurements from the plot display. In the **Measurements selector**, select the check boxes for **Random Jitter** and **Deterministic Jitter**. The object plot updates, removing these two measurements.

16 To print the plot display, select **File > Print to Figure**. From **Figure** window, click the print button.
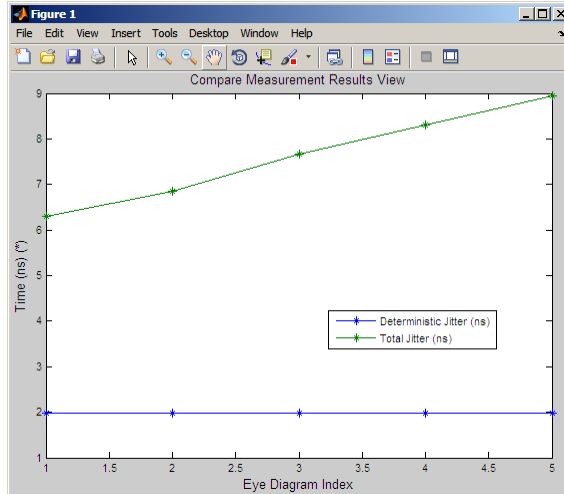
Galois Fields of Odd Characteristic

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 [5].

- 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

In this section...
"Section Overview" on page A-3
"Exponential Format" on page A-3
"Polynomial Format" on page A-4
"List of All Elements of a Galois Field" on page A-5
"Nonuniqueness of Representations" on page A-6

Section Overview

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 technical computing software 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 A-7.

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

$A(1) + A(2) A + A(3) A^2 + ... + A(m) A^{m-1}$

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 A-4 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.

The `gftuple` function is discussed in more detail in "Converting and Simplifying Element Formats" on page A-8.

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 Elements of GF(9) on page A-5 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 A-5.

```
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

In this section...

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 technical computing software 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 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 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.

The output is below.

A-9

```
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.

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 Behavior of gftuple Depending on Format of First Two Inputs on page A-8. 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 =

     2     1

expformat =

     6
```

This output appears at first to contradict the information in the table Elements of GF(9) on page A-5, 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.

The output below reflects the information in the bottom line of the table.

```
polyformat2 =

     2     1


expformat2 =

     7
```

Arithmetic in Galois Fields

In this section...
"Section Overview" on page A-12
"Arithmetic in Prime Fields" on page A-12
"Arithmetic in Extension Fields" on page A-13

Section Overview

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.

The output for this example follows.

```
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 A-21 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 A-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

$$A^2 + A^4 = (1 + 2A) + (2 + 0A) = 3 + 2A = 0 + 2A = A^5 \text{ modulo } 3.$$

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

In this section...
"Section Overview" on page A-15
"Cosmetic Changes of Polynomials" on page A-15
"Polynomial Arithmetic" on page A-16
"Characterization of Polynomials" on page A-17
"Roots of Polynomials" on page A-17

Section Overview

A polynomial over GF(p) is a polynomial whose coefficients are elements of GF(p). Communications Toolbox software 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,

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 $2 + 2x^2 + x^3$.

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.

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 following table.

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.

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 Communications Toolbox software:

- `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.

Analytical Expressions Used in berawgn, bercoding, berfading, and BERTool

This appendix summarizes the main theoretical expressions used in the functions `berawgn`, `berfading`, `bercoding`, and in the Theoretical pane of BERTool. For each modulation scheme, expressions are given for the bit error rate (BER) for both the coded and uncoded cases, and for the symbol error rate (SER) in the uncoded case. Gray coding is assumed in all cases. The BER and SER are the same for binary modulation schemes. Simplifying expressions are also given for certain special cases.

Common Notation

The following notation is used throughout this Appendix:

Quantity or Operation	Notation
Size of modulation constellation	M
Number of bits per symbol	$k = \log_2 M$
Energy per bit-to-noise power-spectral-density ratio	$\dfrac{E_b}{N_0}$
Energy per symbol-to-noise power-spectral-density ratio	$\dfrac{E_s}{N_0} = k\dfrac{E_b}{N_0}$
Bit error rate (BER)	P_b
Symbol error rate (SER)	P_s
Real part	$\mathrm{Re}[\cdot]$
Largest integer smaller than	$\lfloor \cdot \rfloor$

The following mathematical functions are used:

Function	Mathematical Expression
Q function	$$Q(x) = \frac{1}{\sqrt{2\pi}} \int\limits_{x}^{\infty} \exp(-t^2/2)dt$$
Marcum Q function	$$Q(a,b) = \int\limits_{b}^{\infty} t \exp\left(-\frac{t^2 + a^2}{2}\right) I_0(at)dt$$
Modified Bessel function of the first kind of order ν	$$I_\nu(z) = \sum_{k=0}^{\infty} \frac{(z/2)^{\nu+2k}}{k!\Gamma(\nu+k+1)}$$ where $$\Gamma(x) = \int\limits_{0}^{\infty} e^{-t} t^{x-1} dt$$ is the gamma function.
Confluent hypergeometric function	$$_1F_1(a,c;x) = \sum_{k=0}^{\infty} \frac{(a)_k}{(c)_k} \frac{x^k}{k!}$$ where the Pochhammer symbol, $(\lambda)_k$, is defined as $(\lambda)_0 = 1$, $(\lambda)_k = \lambda(\lambda+1)(\lambda+2)\cdots(\lambda+k-1)$.

The following acronyms are used:

Acronym	Definition
M-PSK	M-ary phase-shift keying
DE-M-PSK	Differentially encoded M-ary phase-shift keying
BPSK	Binary phase-shift keying
DE-BPSK	Differentially encoded binary phase-shift keying
QPSK	Quaternary phase-shift keying
DE-QPSK	Differentially encoded quaternary phase-shift keying
OQPSK	Offset quaternary phase-shift keying
DE-OQPSK	Differentially encoded offset quaternary phase-shift keying
M-DPSK	M-ary differential phase-shift keying
M-PAM	M-ary pulse amplitude modulation
M-QAM	M-ary quadrature amplitude modulation
M-FSK	M-ary frequency-shift keying
MSK	Minimum shift keying
M-CPFSK	M-ary continuous-phase frequency-shift keying

Analytical Expressions Used in berawgn

M-PSK

From equation 8.22 in [6]:

$$P_s = \frac{1}{\pi} \int\limits_{0}^{(M-1)\pi/M} \exp\left(-\frac{kE_b}{N_0} \frac{\sin^2\left[\pi/M\right]}{\sin^2\theta} \right) d\theta$$

The following expression is very close, but not strictly equal, to the exact BER (from [2] and equation 8.29 from[6]):

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2} (w_i^{'})P_i\right)$$

where $w_i^{'} = w_i + w_{M-i}$, $w_{M/2}^{'} = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$
\begin{aligned}
P_i = {} & \frac{1}{2\pi}\int_0^{\pi(1-(2i-1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2\left[(2i-1)\pi/M\right]}{\sin^2\theta}\right)d\theta \\
& -\frac{1}{2\pi}\int_0^{\pi(1-(2i+1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2\left[(2i+1)\pi/M\right]}{\sin^2\theta}\right)d\theta
\end{aligned}
$$

Special case of $M = 2$, e.g., BPSK (equation 5.2-57 from [4]):

$$P_s = P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

Special case of $M = 4$, e.g., QPSK (equations 5.2-59 and 5.2-62 from [4]):

$$P_s = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1 - \frac{1}{2}Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

DE-M-PSK

$M = 2$, e.g., DE-BPSK (equation 8.36 from [6]):

$$P_s = P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right) - 2Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

$M = 4$, e.g., DE-QPSK (equation 8.38 from [6]):

$$P_s = 4Q\left(\sqrt{\frac{2E_b}{N_0}}\right) - 8Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right) + 8Q^3\left(\sqrt{\frac{2E_b}{N_0}}\right) - 4Q^4\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

From equation 5 in [7]:

$$P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$

OQPSK

Same BER/SER as QPSK [6].

DE-OQPSK

Same BER/SER as DE-QPSK [7].

M-DPSK

From equation 8.84 in [6]:

$$P_s = \frac{\sin(\pi/M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{\exp\left(-(kE_b/N_0)(1-\cos(\pi/M)\cos\theta)\right)}{1-\cos(\pi/M)\cos\theta}d\theta$$

The following expression is very close, but not strictly equal, to the exact BER [2]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i^{'})A_i\right)$$

where $w_i^{'} = w_i + w_{M-i}$, $w_{M/2}^{'} = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$A_i = F\left((2i+1)\frac{\pi}{M}\right) - F\left((2i-1)\frac{\pi}{M}\right)$$

$$F(\psi) = -\frac{\sin\psi}{4\pi}\int_{-\pi/2}^{\pi/2}\frac{\exp\left(-kE_b/N_0(1-\cos\psi\cos t)\right)}{1-\cos\psi\cos t}\,dt$$

Special case of $M = 2$ (equation 8.85 from [6]):

$$P_b = \frac{1}{2}\exp\left(-\frac{E_b}{N_0}\right)$$

M-PAM

From equations 8.3 and 8.7 in [6], and equation 5.2-46 in [4]:

$$P_s = 2\left(\frac{M-1}{M}\right)Q\left(\sqrt{\frac{6}{M^2-1}\frac{kE_b}{N_0}}\right)$$

From [1]:

$$P_b = \frac{2}{M\log_2 M}\times$$

$$\sum_{k=1}^{\log_2 M}\sum_{i=0}^{(1-2^{-k})M-1}\left\{(-1)^{\left\lfloor\frac{i2^{k-1}}{M}\right\rfloor}\left(2^{k-1}-\left\lfloor\frac{i2^{k-1}}{M}+\frac{1}{2}\right\rfloor\right)Q\left((2i+1)\sqrt{\frac{6\log_2 M}{M^2-1}\frac{E_b}{N_0}}\right)\right\}$$

M-QAM

For square M-QAM, $k = \log_2 M$ is even (equation 8.10 from [6], and equations 5.2-78 and 5.2-79 from [4]):

$$P_s = 4\frac{\sqrt{M}-1}{\sqrt{M}}Q\left(\sqrt{\frac{3}{M-1}\frac{kE_b}{N_0}}\right) - 4\left(\frac{\sqrt{M}-1}{\sqrt{M}}\right)^2 Q^2\left(\sqrt{\frac{3}{M-1}\frac{kE_b}{N_0}}\right)$$

From [1]:

$$P_b = \frac{2}{\sqrt{M} \log_2 \sqrt{M}}$$

$$\times \sum_{k=1}^{\log_2 \sqrt{M}} \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{\sqrt{M}} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1)\sqrt{\frac{6 \log_2 M}{2(M-1)} \frac{E_b}{N_0}} \right) \right\}$$

For rectangular (non-square) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$,

$I = 2^{\frac{k-1}{2}}$, and $J = 2^{\frac{k+1}{2}}$:

$$P_s = \frac{4IJ - 2I - 2J}{M}$$

$$\times Q\left(\sqrt{\frac{6 \log_2(IJ)}{(I^2 + J^2 - 2)} \frac{E_b}{N_0}} \right) - \frac{4}{M}(1 + IJ - I - J)Q^2\left(\sqrt{\frac{6 \log_2(IJ)}{(I^2 + J^2 - 2)} \frac{E_b}{N_0}} \right)$$

From [1]:

$$P_b = \frac{1}{\log_2(IJ)} \left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l) \right)$$

where

$$P_I(k) = \frac{2}{I} \sum_{i=0}^{(1-2^{-k})I-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{I} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{I} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1)\sqrt{\frac{6 \log_2(IJ)}{I^2 + J^2 - 2} \frac{E_b}{N_0}} \right) \right\}$$

and

B-9

$$P_J(k) = \frac{2}{J} \sum_{j=0}^{(1-2^{-l})J-1} \left\{ (-1)^{\left\lfloor \frac{j2^{l-1}}{J} \right\rfloor} \left(2^{l-1} - \left\lfloor \frac{j2^{l-1}}{J} + \frac{1}{2} \right\rfloor \right) Q\left((2j+1) \sqrt{\frac{6\log_2(IJ)}{I^2 + J^2 - 2} \frac{E_b}{N_0}} \right) \right\}$$

Orthogonal M-FSK with Coherent Detection

From equation 8.40 in [6] and equation 5.2-21 in [4]:

$$P_s = 1 - \int_{-\infty}^{\infty} \left[Q\left(-q - \sqrt{\frac{2kE_b}{N_0}} \right) \right]^{M-1} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{q^2}{2} \right) dq$$

$$P_b = \frac{2^{k-1}}{2^k - 1} P_s$$

Nonorthogonal 2-FSK with Coherent Detection

For $M = 2$ (from equation 5.2-21 in [4] and equation 8.44 in [6]):

$$P_s = P_b = Q\left(\sqrt{\frac{E_b(1 - \mathrm{Re}[\rho])}{N_0}} \right)$$

ρ is the complex correlation coefficient:

$$\rho = \frac{1}{2E_b} \int_0^{T_b} \tilde{s}_1(t)\tilde{s}_2^*(t)dt$$

where $\tilde{s}_1(t)$ and $\tilde{s}_2(t)$ are complex lowpass signals, and

$$E_b = \frac{1}{2} \int_0^{T_b} |\tilde{s}_1(t)|^2 \, dt = \frac{1}{2} \int_0^{T_b} |\tilde{s}_2(t)|^2 \, dt$$

For example:

$$\tilde{s}_1(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_1 t}, \ \tilde{s}_2(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_2 t}$$

$$\rho = \frac{1}{2E_b} \int_0^{T_b} \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_1 t} \sqrt{\frac{2E_b}{T_b}} e^{-j2\pi f_2 t} dt = \frac{1}{T_b} \int_0^{T_b} e^{j2\pi(f_1 - f_2)t} dt$$

$$= \frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} e^{j\pi \Delta f t}$$

where $\Delta f = f_1 - f_2$.

$$\mathrm{Re}[\rho] = \mathrm{Re}\left[\frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} e^{j\pi \Delta f t} \right] = \frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} \cos(\pi \Delta f T_b) = \frac{\sin(2\pi \Delta f T_b)}{2\pi \Delta f T_b}$$

$$\Rightarrow P_b = Q\left(\sqrt{\frac{E_b(1 - \sin(2\pi \Delta f T_b)/(2\pi \Delta f T_b))}{N_0}} \right)$$

(from equation 8.44 in [6], where $h = \Delta f T_b$)

Orthogonal M-FSK with Noncoherent Detection

From equation 5.4-46 in [4] and equation 8.66 in [6]:

$$P_s = \sum_{m=1}^{M-1} (-1)^{m+1} \binom{M-1}{m} \frac{1}{m+1} \exp\left[-\frac{m}{m+1} \frac{kE_b}{N_0} \right]$$

$$P_b = \frac{1}{2} \frac{M}{M-1} P_s$$

Nonorthogonal 2-FSK with Noncoherent Detection

For $M = 2$ (from equation 5.4-53 in [4] and equation 8.69 in [6]):

$$P_s = P_b = Q(\sqrt{a}, \sqrt{b}) - \frac{1}{2} \exp\left(-\frac{a+b}{2} \right) I_0(\sqrt{ab})$$

where

$$a = \frac{E_b}{2N_0}(1 - \sqrt{1 - |\rho|^2}), \; b = \frac{E_b}{2N_0}(1 + \sqrt{1 - |\rho|^2})$$

Precoded MSK with Coherent Detection

Same BER/SER as BPSK.

Differentially Encoded MSK with Coherent Detection

Same BER/SER as DE-BPSK.

MSK with Noncoherent Detection (Optimum Block-by-Block)

Upper bound (from equations 10.166 and 10.164 in [5]):

$$P_s = P_b$$

$$\leq \frac{1}{2}\left[1 - Q\left(\sqrt{b_1}, \sqrt{a_1}\right) + Q\left(\sqrt{a_1}, \sqrt{b_1}\right)\right] + \frac{1}{4}\left[1 - Q\left(\sqrt{b_4}, \sqrt{a_4}\right) + Q\left(\sqrt{a_4}, \sqrt{b_4}\right)\right] + \frac{1}{2}e^{-\frac{E_b}{N_0}}$$

where

$$a_1 = \frac{E_b}{N_0}\left(1 - \sqrt{\frac{3 - 4/\pi^2}{4}}\right), \quad b_1 = \frac{E_b}{N_0}\left(1 + \sqrt{\frac{3 - 4/\pi^2}{4}}\right)$$

$$a_4 = \frac{E_b}{N_0}\left(1 - \sqrt{1 - 4/\pi^2}\right), \quad b_4 = \frac{E_b}{N_0}\left(1 + \sqrt{1 - 4/\pi^2}\right)$$

CPFSK Coherent Detection (Optimum Block-by-Block)

Lower bound (from equation 5.3-17 in [4]):

$$P_s > K_{\delta_{\min}} Q\left(\sqrt{\frac{E_b}{N_0}\delta_{\min}^2}\right)$$

Upper bound:

$$\delta_{\min}^2 > \min_{1 \le i \le M-1} \left\{ 2i\left(1 - \mathrm{sinc}(2ih)\right) \right\}$$

where h is the modulation index, and $K_{\delta_{\min}}$ is the number of paths having the minimum distance.

$$P_b \cong \frac{P_s}{k}$$

Analytical Expressions Used in berfading

Notation

The following notation is used for the expressions found in `berfading`.

Value	Notation
Power of the fading amplitude r	$\Omega = E\left[r^2\right]$, where $E[\cdot]$ denotes statistical expectation
Number of diversity branches	L

Value	Notation
SNR per symbol per branch	$\overline{\gamma}_l = \left(\Omega_l \dfrac{E_s}{N_0}\right)/L = \left(\Omega_l \dfrac{kE_b}{N_0}\right)/L$ For identically-distributed diversity branches: $\overline{\gamma} = \left(\Omega \dfrac{kE_b}{N_0}\right)/L$
Moment generating functions for each diversity branch	Rayleigh fading: $M_{\gamma_l}(s) = \dfrac{1}{1 - s\overline{\gamma}_l}$ Rician fading: $M_{\gamma_l}(s) = \dfrac{1+K}{1+K-s\overline{\gamma}_l} e^{\left[\frac{Ks\overline{\gamma}_l}{(1+K)-s\overline{\gamma}_l}\right]}$ where K is the ratio of energy in the specular component to the energy in the diffuse component (linear scale).For identically-distributed diversity branches: $M_{\gamma_l}(s) = M_\gamma(s)$ for all l.

The following acronyms are used:

Acronym	Definition
MRC	maximal-ratio combining
EGC	equal-gain combining

M-PSK with MRC

From equation 9.15 in [6]:

$$P_s = \frac{1}{\pi} \int\limits_0^{(M-1)\pi/M} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{\sin^2(\pi/M)}{\sin^2\theta}\right)d\theta$$

From [2] and [6]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2} (w_i')\bar{P}_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$\bar{P}_i = \frac{1}{2\pi} \int\limits_0^{\pi(1-(2i-1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i-1)\pi}{M}\right)d\theta$$
$$-\frac{1}{2\pi} \int\limits_0^{\pi(1-(2i+1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i+1)\pi}{M}\right)d\theta$$

For the special case of Rayleigh fading with $M = 2$ (from equations C-18, C-21, and Table C-1 in [5]):

$$P_b = \frac{1}{2}\left[1 - \mu\sum_{i=0}^{L-1}\binom{2i}{i}\left(\frac{1-\mu^2}{4}\right)^i\right]$$

where

$$\mu = \sqrt{\frac{\bar{\gamma}}{\bar{\gamma}+1}}$$

If $L = 1$:

$$P_b = \frac{1}{2}\left[1 - \sqrt{\frac{\overline{\gamma}}{\overline{\gamma}+1}}\right]$$

DE-M-PSK with MRC

For $M = 2$ (from equations 8.37 and 9.8-9.11 in [6]):

$$P_s = P_b = \frac{2}{\pi}\int\limits_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta - \frac{2}{\pi}\int\limits_0^{\pi/4}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta$$

M-PAM with MRC

From equation 9.19 in [6]:

$$P_s = \frac{2(M-1)}{M\pi}\int\limits_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{3/(M^2-1)}{\sin^2\theta}\right)d\theta$$

From [1] and [6]:

$$P_b = \frac{2}{\pi M\log_2 M}$$

$$\times \sum_{k=1}^{\log_2 M}\sum_{i=0}^{(1-2^{-k})M-1}\left\{(-1)^{\left\lfloor\frac{i2^{k-1}}{M}\right\rfloor}\left(2^{k-1}-\left\lfloor\frac{i2^{k-1}}{M}+\frac{1}{2}\right\rfloor\right)\int\limits_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(M^2-1)}{\sin^2\theta}\right)d\theta\right\}$$

M-QAM with MRC

For square M-QAM, $k = \log_2 M$ is even (equation 9.21 in [6]):

$$P_s = \frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right)\int_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{3/(2(M-1))}{\sin^2\theta}\right)d\theta$$

$$-\frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right)^2\int_0^{\pi/4}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{3/(2(M-1))}{\sin^2\theta}\right)d\theta$$

From [1] and [6]:

$$P_b = \frac{2}{\pi\sqrt{M}\log_2\sqrt{M}}$$

$$\times \sum_{k=1}^{\log_2\sqrt{M}}\sum_{i=0}^{(1-2^{-k})\sqrt{M}-1}\left\{(-1)^{\left\lfloor\frac{i2^{k-1}}{\sqrt{M}}\right\rfloor}\left(2^{k-1} - \left\lfloor\frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(2(M-1))}{\sin^2\theta}\right)d\theta\right\}$$

For rectangular (nonsquare) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$, $I = 2^{\frac{k-1}{2}}$,

$J = 2^{\frac{k+1}{2}}$, $\overline{\gamma}_l = \Omega_l \log_2(IJ)\frac{E_b}{N_0}$, and

$$P_s = \frac{4IJ - 2I - 2J}{M\pi}\int_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{3/(I^2 + J^2 - 2)}{\sin^2\theta}\right)d\theta$$

$$-\frac{4}{M\pi}(1 + IJ - I - J)\int_0^{\pi/4}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{3/(I^2 + J^2 - 2)}{\sin^2\theta}\right)d\theta$$

From [1] and [6]:

$$P_b = \frac{1}{\log_2(IJ)}\left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l)\right)$$

$$P_I(k) = \frac{2}{I\pi}\sum_{i=0}^{(1-2^{-k})I-1}\left\{(-1)^{\left\lfloor\frac{i2^{k-1}}{I}\right\rfloor}\left(2^{k-1} - \left\lfloor\frac{i2^{k-1}}{I}+\frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$

$$P_J(k) = \frac{2}{J\pi}\sum_{j=0}^{(1-2^{-l})J-1}\left\{(-1)^{\left\lfloor\frac{j2^{l-1}}{J}\right\rfloor}\left(2^{l-1} - \left\lfloor\frac{j2^{l-1}}{J}+\frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L}M_{\gamma_l}\left(-\frac{(2j+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$

M-DPSK with Postdetection EGC

From equation 8.165 in [6]:

$$P_s = \frac{\sin(\pi/M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{[1-\cos(\pi/M)\cos\theta]}\prod_{l=1}^{L}M_{\gamma_l}\left(-[1-\cos(\pi/M)\cos\theta]\right)d\theta$$

From [2] and [6]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')\bar{A}_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$\bar{A}_i = \bar{F}\left((2i+1)\frac{\pi}{M}\right) - \bar{F}\left((2i-1)\frac{\pi}{M}\right)$$

$$\bar{F}(\psi) = -\frac{\sin\psi}{4\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{(1-\cos\psi\cos t)}\prod_{l=1}^{L}M_{\gamma_l}\left(-(1-\cos\psi\cos t)\right)dt$$

For the special case of Rayleigh fading with $M=2$, and $L=1$ (equation 8.173 from [6]):

$$P_b = \frac{1}{2(1 + \overline{\gamma})}$$

Orthogonal 2-FSK, Coherent Detection with MRC

From equation 9.11 in [6]:

$$P_s = P_b = \frac{1}{\pi} \int\limits_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1/2}{\sin^2\theta}\right) d\theta$$

For the special case of Rayleigh fading (equations 14.4-15 and 14.4-21 in [4]):

$$P_s = P_b = \frac{1}{2^L}\left(1 - \sqrt{\frac{\overline{\gamma}}{2 + \overline{\gamma}}}\right)^L \sum_{k=0}^{L-1}\binom{L-1+k}{k}\frac{1}{2^k}\left(1 + \sqrt{\frac{\overline{\gamma}}{2 + \overline{\gamma}}}\right)^k$$

Nonorthogonal 2-FSK, Coherent Detection with MRC

Equations 9.11 and 8.44 in [6]:

$$P_s = P_b = \frac{1}{\pi} \int\limits_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(1 - \mathrm{Re}[\rho])/2}{\sin^2\theta}\right) d\theta$$

For the special case of Rayleigh fading with $L = 1$ (equation 20 in [3] and equation 8.130 in [6]):

$$P_s = P_b = \frac{1}{2}\left[1 - \sqrt{\frac{\overline{\gamma}(1 - \mathrm{Re}[\rho])}{2 + \overline{\gamma}(1 - \mathrm{Re}[\rho])}}\right]$$

Orthogonal M-FSK, Noncoherent Detection with EGC

Rayleigh fading (equation 14.4-47 in [4]):

$$P_s = 1 - \int_0^\infty \frac{1}{(1+\overline{\gamma})^L (L-1)!} U^{L-1} e^{-\frac{U}{1+\overline{\gamma}}} \left(1 - e^{-U} \sum_{k=0}^{L-1} \frac{U^k}{k!} \right)^{M-1} dU$$

$$P_b = \frac{1}{2} \frac{M}{M-1} P_s$$

Rician fading (equation 41 in [3]):

$$P_s = \sum_{r=1}^{M-1} \frac{(-1)^{r+1} e^{-LK\overline{\gamma}_r /(1+\overline{\gamma}_r)}}{(r(1+\overline{\gamma}_r)+1)^L} \binom{M-1}{r} \sum_{n=0}^{r(L-1)} \beta_{nr} \frac{\Gamma(L+n)}{\Gamma(L)} \left[\frac{1+\overline{\gamma}_r}{r+1+r\overline{\gamma}_r} \right]^n {}_1F_1 \left(L+n, L; \frac{LK\overline{\gamma}_r /(1+\overline{\gamma}_r)}{r(1+\overline{\gamma}_r)+1} \right)$$

$$P_b = \frac{1}{2} \frac{M}{M-1} P_s$$

where

$$\overline{\gamma}_r = \frac{1}{1+K} \overline{\gamma}$$

$$\beta_{nr} = \sum_{i=n-(L-1)}^{n} \frac{\beta_{i(r-1)}}{(n-i)!} I_{[0,(r-1)(L-1)]}(i)$$

$$\beta_{00} = \beta_{0r} = 1$$

$$\beta_{n1} = 1/n!$$

$$\beta_{1r} = r$$

and $I_{[a,b]}(i) = 1$ if $a \le i \le b$ and 0 otherwise.

Nonorthogonal 2-FSK, Noncoherent Detection with No Diversity

From equation 8.163 in [6]:

$$P_s = P_b = \frac{1}{4\pi} \int_{-\pi}^{\pi} \frac{1-\varsigma^2}{1+2\varsigma\sin\theta+\varsigma^2} M_\gamma \left(-\frac{1}{4} (1+\sqrt{1-\rho^2})(1+2\varsigma\sin\theta+\varsigma^2) \right) d\theta$$

where

$$\varsigma = \sqrt{\frac{1 - \sqrt{1 - \rho^2}}{1 + \sqrt{1 - \rho^2}}}$$

Analytical Expressions Used in bercoding and BERTool

In this section...
"Common Notation for This Section" on page B-23
"Block Coding" on page B-23
"Convolutional Coding" on page B-26

Common Notation for This Section

Description	Notation
Energy-per-information bit-to-noise power-spectral-density ratio	$\gamma_b = \dfrac{E_b}{N_0}$
Message length	K
Code length	N
Code rate	$R_c = \dfrac{K}{N}$

Block Coding

Specific notation for block coding expressions: $d_{\min}$ is the minimum distance of the code.

Soft Decision

BPSK, QPSK, OQPSK, PAM-2, QAM-4, and precoded MSK (equation 8.1-52 in [4]):

$$P_b \le \frac{1}{2}(2^K - 1)Q\left(\sqrt{2\gamma_b R_c d_{\min}}\right)$$

DE-BPSK, DE-QPSK, DE-OQPSK, and DE-MSK:

$$P_b \le \frac{1}{2}(2^K - 1)\Big[2Q\big(\sqrt{2\gamma_b R_c d_{\min}}\big)\Big[1 - Q\big(\sqrt{2\gamma_b R_c d_{\min}}\big)\Big]\Big]$$

BFSK, coherent detection (equations 8.1-50 and 8.1-58 in [4]):

$$P_b \le \frac{1}{2}(2^K - 1)Q\big(\sqrt{\gamma_b R_c d_{\min}}\big)$$

BFSK, noncoherent square-law detection (equations 8.1-65 and 8.1-64 in [4]):

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp\left(-\frac{1}{2}\gamma_b R_c d_{\min}\right)\sum_{i=0}^{d_{\min}-1}\left(\frac{1}{2}\gamma_b R_c d_{\min}\right)^i\frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

DPSK:

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp\left(-\gamma_b R_c d_{\min}\right)\sum_{i=0}^{d_{\min}-1}\left(\gamma_b R_c d_{\min}\right)^i\frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

Hard Decision

General linear block code (equations 4.3, 4.4 in [9], and 12.136 in [5]):

$$P_b \le \frac{1}{N}\sum_{m=t+1}^{N}(m+t)\binom{N}{m}p^m(1-p)^{N-m}$$

$$t = \left\lfloor \frac{1}{2}(d_{\min} - 1)\right\rfloor$$

Hamming code (equations 4.11, 4.12 in [9], and 6.72, 6.73 in [10]):

$$P_b \approx \frac{1}{N}\sum_{m=2}^{N}m\binom{N}{m}p^m(1-p)^{N-m} = p - p(1-p)^{N-1}$$

(24, 12) extended Golay code (equation 4.17 in [9], and 12.139 in [5]):

$$P_b \leq \frac{1}{24} \sum_{m=4}^{24} \beta_m \binom{24}{m} p^m (1-p)^{24-m}$$

where β_m is the average number of channel symbol errors that remain in corrected N-tuple when the channel caused m symbol errors (table 4.2 in [9]).

Reed-Solomon code with $N = Q - 1 = 2^q - 1$:

$$P_b \approx \frac{2^{q-1}}{2^q - 1} \frac{1}{N} \sum_{m=t+1}^{N} m \binom{N}{m} (P_s)^m (1 - P_s)^{N-m}$$

for FSK (equations 4.25, 4.27 in [9], 8.1-115, 8.1-116 in [4], 8.7, 8.8 in [10], and 12.142, 12.143 in [5]), and

$$P_b \approx \frac{1}{q} \frac{1}{N} \sum_{m=t+1}^{N} m \binom{N}{m} (P_s)^m (1 - P_s)^{N-m}$$

otherwise.

If $\log_2 Q / \log_2 M = q / k = h$ where h is an integer (equation 1 in [8]):

$$P_s = 1 - (1-s)^h$$

where s is the symbol error rate (SER) in an uncoded AWGN channel.

For example, for BPSK, $M = 2$ and $P_s = 1 - (1-s)^q$

Otherwise, P_s is given by table 1 and equation 2 in [8].

Convolutional Coding

Specific notation for convolutional coding expressions: d_{free} is the free distance of the code, and a_d is the number of paths of distance d from the all-zero path that merge with the all-zero path for the first time.

Soft Decision

From equations 8.2-26, 8.2-24, and 8.2-25 in [4], and equations 13.28 and 13.27 in [5]:

$$P_b < \sum_{d=d_{free}}^{\infty} a_d f(d) P_2(d)$$

with transfer function

$$T(D,N) = \sum_{d=d_{free}}^{\infty} a_d D^d N^{f(d)}$$

$$\left. \frac{dT(D,N)}{dN} \right|_{N=1} = \sum_{d=d_{free}}^{\infty} a_d f(d) D^d$$

where $f(d)$ is the exponent of N as a function of d.

Results for BPSK, QPSK, OQPSK, PAM-2, QAM-4, precoded MSK, DE-BPSK, DE-QPSK, DE-OQPSK, DE-MSK, DPSK, and BFSK are obtained as:

$$P_2(d) = P_b \big|_{\frac{E_b}{N_0} = \gamma_b R_c d}$$

where P_b is the BER in the corresponding uncoded AWGN channel. For example, for BPSK (equation 8.2-20 in [4]):

$$P_2(d) = Q\left(\sqrt{2\gamma_b R_c d}\right)$$

Hard Decision

From equations 8.2-33, 8.2-28, and 8.2-29 in [4], and equations 13.28, 13.24, and 13.25 in [5]:

$$P_b < \sum_{d=d_{free}}^{\infty} a_d f(d) P_2(d)$$

where

$$P_2(d) = \sum_{k=(d+1)/2}^{d} \binom{d}{k} p^k (1-p)^{d-k}$$

when d is odd, and

$$P_2(d) = \sum_{k=d/2+1}^{d} \binom{d}{k} p^k (1-p)^{d-k} + \frac{1}{2}\binom{d}{d/2} p^{d/2}(1-p)^{d/2}$$

when d is even (p is the bit error rate (BER) in an uncoded AWGN channel).

Selected Bibliography

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

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

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

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

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

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

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

[8] Gulliver, T. A., "Matching Q-ary Reed-Solomon codes with M-ary modulation," *IEEE Trans. Commun.*, vol. 45, no. 11, Nov. 1997, pp. 1349-1353.

[9] Odenwalder, J. P., *Error Control Coding Handbook* (Final report), Linkabit Corp., 15 July 1976.

[10] Sklar, B., *Digital Communications*, 2nd Ed., Prentice-Hall, 2001.

C

Algorithms

- "Algorithms Used to Decode BCH and Reed-Solomon Codes" on page C-2
- "Compute Optimum Quantizer Boundaries for use with Soft-Decision Type of Viterbi Decoder" on page C-6
- "References" on page C-11

Algorithms Used to Decode BCH and Reed-Solomon Codes

Errors-only Decoding

Overview

The errors-only decoding algorithm used for BCH and RS codes can be described by the following steps (sections 5.3.2, 5.4, and 5.6 in [1]).

1 Calculate the first $2t$ terms of the infinite degree syndrome polynomial, $S(z)$.

2 If those $2t$ terms of $S(z)$ are all equal to 0, then the code has no errors , no correction needs to be performed, and the decoding algorithm ends.

3 If one or more terms of $S(z)$ are nonzero, calculate the error locator polynomial, $\Lambda(z)$, via the Berlekamp algorithm.

4 Calculate the error evaluator polynomial, $\Omega(z)$, via

$$\Lambda(z)S(z) = \Omega(z) \bmod z^{2t}$$

5 Correct an error in the codeword according to

$$e_{i_m} = \frac{\Omega(\alpha^{-i_m})}{\Lambda'(\alpha^{-i_m})}$$

where e_{i_m} is the error magnitude in the i_m th position in the codeword, m is a value less than the error-correcting capability of the code, $\Omega(z)$ is the error magnitude polynomial, $\Lambda'(z)$ is the formal derivative [2] of the error locator polynomial, $\Lambda(z)$, and α is the primitive element of the Galois field of the code.

Further description of several of the steps is given in the following sections.

Syndrome Calculation

For narrow-sense codes, the $2t$ terms of $S(z)$ are calculated by evaluating the received codeword at successive powers of α (the field's primitive element) from 0 to $2t$-1. In other words, if we assume one-based indexing of codewords $C(z)$ and the syndrome polynomial $S(z)$, and that codewords are of the form $[c_1 \ c_1 \ ... \ c_N]$, then each term S_i of $S(z)$ is given as

$$S_i = \sum_{i=1}^{N} c_i \alpha^{N-1-i}$$

Error Locator Polynomial Calculation

The error locator polynomial, $\Lambda(z)$, is found using the Berlekamp algorithm. A complete description of this algorithm is found in [1], but we summarize the algorithm as follows.

We define the following variables.

Variable	Description
n	Iterator variable
k	Iterator variable
L	Length of the feedback register used to generate the first $2t$ terms of $S(z)$
$D(z)$	Correction polynomial
d	Discrepancy

The following diagram shows the iterative procedure (i.e., the Berlekamp algorithm) used to find $\Lambda(z)$.

$n = 0,\ L = 0,\ k = -1,$
$\Lambda(z) = 1,\ D(z) = z$

$d = \sum_{i=0}^{L} \Lambda_i S_{n-i}$

$d : 0$

$=$

$\neq$

$\Lambda^*(z) = \Lambda(z) - d D(z)$

$L : n - k$

$<$

$L^* = n - k$

$k = n - L$

$D(z) = \dfrac{\Lambda(z)}{d}$

$L = L^*$

$\geq$

$\Lambda(z) = \Lambda^*(z)$

$D(z) = z D(z)$

$n = n + 1$

$n : 2t$

$<$

$=$

STOP
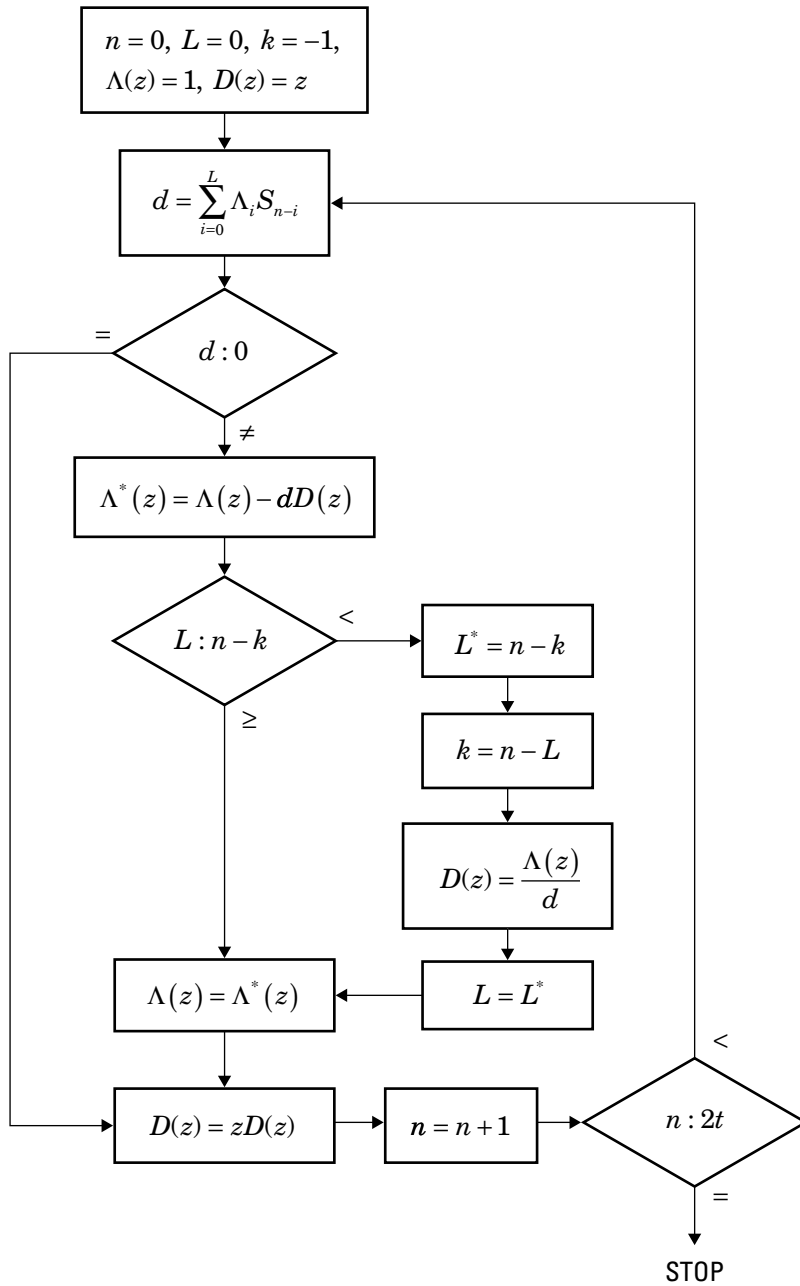
Error Evaluator Polynomial Calculation

The error evaluator polynomial, $\Omega(z)$, is simply the convolution of $\Lambda(z)$ and $S(z)$.

Compute Optimum Quantizer Boundaries for use with Soft-Decision Type of Viterbi Decoder

This section describes how to compute the optimum quantizer boundaries to quantize log-likelihood ratios (LLRs) from a Rectangular QAM Demodulator Baseband block for use with the 'Soft decision' decision type of Viterbi Decoder. LLRs from a Rectangular QAM Demodulator Baseband block assume an AWGN Channel.

As an example, use a binary mapped (as opposed to Gray mapped) 16-QAM constellation with a binary mapping for each constellation point, as shown in the following figure. The minimum distance between symbols is two.
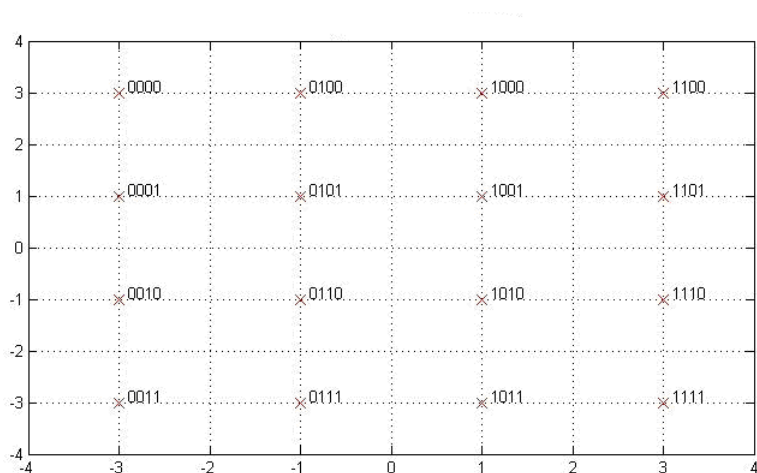
Binary mapped 16-QAM Constellation

The two most significant bits (MSB) in this constellation mapping remain constant along the quadrature axis. The analysis of these two bits can be simplified by looking along the in-phase axis only. The following figure shows the mapping of these two bits along the in-phase axis.
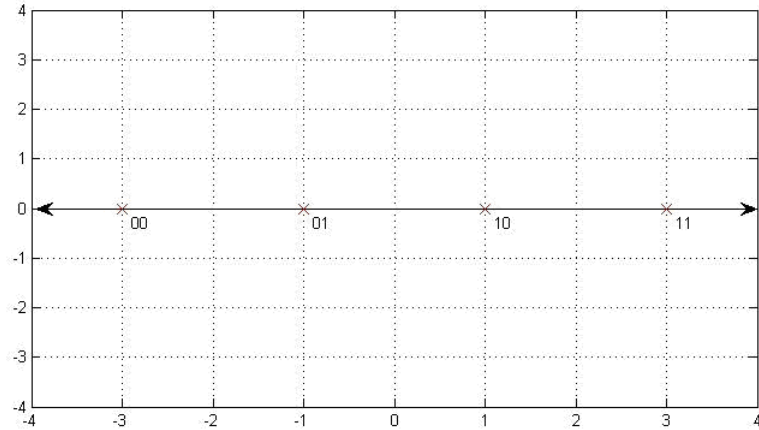
Mapping of the two most significant bits along in-phase axis

Similar simplification applies to the two least significant bits (LSB) that remain constant along the in-phase axis.

The LLR L(b) is given as:

$$L(b) = \log_e \left(\frac{\displaystyle\sum_{s \in S_0} e^{\frac{1}{\sigma^2}\left((x-s_x)^2+(y-s_y)^2\right)}}{\displaystyle\sum_{s \in S_1} e^{\frac{1}{\sigma^2}\left((x-s_x)^2+(y-s_y)^2\right)}} \right)$$

where r = received signal with coordinates x,y.

b = transmitted bit (one of the K bits in an M-ary constellation, assuming all M points are equally probable, where $K = \log_2(M)$)

S_0 = ideal symbols/constellation points with bit 0 (at the given bit position)

S_1 = ideal symbols/constellation points with bit 1 (at the given bit position)

sx = Inphase (or X) coordinate of ideal symbols/constellation points.

sy = Quadrature (or Y) coordinate of ideal symbols/constellation points.

σ^2 = Noise Variance. This cmoposite noise variance is the sum of noise components along theInphase axis and Quadrature axis which are assumed to be independent and of equal power.

For these two bits, as we are looking along the in-phase axis, it can be simplified as:

$$L(b) = \log e \left(\frac{\displaystyle\sum_{s \in S_o} e^{\frac{1}{\sigma^2}\left((x - s_x)^2\right)}}{\displaystyle\sum_{s \in S_1} e^{\frac{1}{\sigma^2}\left((x - s_x)^2\right)}} \right)$$

In the summation terms in the numerator and denominator, the effect of the nearest point would outweigh the rest of the points. Hence, this can be further simplified to:

$$L(b) \approx -\frac{1}{\sigma^2}\left((x - s_{x0})^2 - (x - s_{x1})^2\right)$$

where S_{x0} equals the nearest ideal constellation point with 0 mapping

where S_{x1} equals the nearest ideal constellation point with 1mapping

Consider the case when there is no noise in received signal and the constellation point with 00 mapping is received. From the previous figure, it is clear that the MSB has the best noise resistance against error. The LLR for the MSB can be approximated using above equation as:

$$L(b) \approx \frac{16}{\sigma^2}$$

as $(x\text{-}s_{x0}) = 0$ and $(x\text{-}s_{x1}) = 4$

Using the same equation, LLR for the second MSB can be approximated to:

$$L(b) \approx \frac{4}{\sigma^2}$$

as $(x-s_{x0}) = 0$ and $(x-s_{x1}) = 2$

Now consider the case when there is no noise in received signal and the constellation point with 11 mapping is received. Similar analysis for the MSB would yield:

$$L(b) \approx -\frac{16}{\sigma^2}$$

For the second MSB:

$$L(b) \approx -\frac{4}{\sigma^2}$$

Hence, under no noise condition, the range of LLR values for the MSB is:

$$\left(-\frac{16}{\sigma^2}, \frac{16}{\sigma^2} \right)$$

As noise increases, the distribution of LLR values will increase within this range indicating higher probability of error. Some values would fall outside this range as well and they inherently have more resistance to noise. Along the same lines, the range of LLR values for the second MSB is:

$$\left(-\frac{4}{\sigma^2}, \frac{4}{\sigma^2} \right)$$

When considering the design of quantization boundaries for L-level uniform quantizer, quantizing this range in L levels would provide optimum performance. Lets say, we have a 3-bit uniform quantizer ($L = 8$).

The quantization boundaries for the MSB would be:

$$B_0 = \frac{4}{\sigma^2}(-3:3)$$

For the second MSB:

$$B_1 = \frac{1}{\sigma^2}(-3:3)$$

Following similar analysis for second LSB and LSB along quadrature axis provides same results, respectively.

Now, B_0 optimizes the MSB while B_1 optimizes the second MSB. To find overall optimum quantizer boundaries, try out some values between these two sets of values. Our simulation results indicate that the optimum system BER performance is obtained with this set of boundary values:

$$B_{opt} = \frac{2}{\sigma^2}(-3:3)$$

The system used in this simulation is shown in the product demo titled "LLR vs. Hard Decision Demodulation" in Communications Blockset and Communications Toolbox. In the blockset, note that the LLR values computed by the demodulator need to be multiplied by -1 before feeding them to uniform quantizer. The sign of LLR values indicate which bit is received – positive for binary 0 and negative for binary 1. By multiplying these values with -1, we flip this notation – now, positive sign indicates binary 1 and negative sign indicates binary 0. The quantizer maps positive values to indices 4:7, 7 being the most confident 1 and negative values to 0:3, 0 being the most confident 0. This matches the Viterbi Decoder block's interpretation of its input, namely, 0 indicates most confident 0 while 7 indicates most confident 1. Thus, multiplying LLR values with -1 before feeding them to quantizer maps them to the right quantizer index for use with Viterbi Decoder.

References

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

[2] Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

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

Equalizers

Channels

Galois Field Computations

Index