## Communications Toolbox ${ }^{\text {TM }}$

User's Guide

## MATLAB $^{\circ} \&$ SIMULINK $^{\circ}$

## How to Contact MathWorks



Latest news:
Sales and services:
User community:
Technical support:
www. mathworks . com
www.mathworks.com/sales_and_services
www.mathworks.com/matlabcentral
www.mathworks.com/support/contact_us

Phone:
508-647-7000

The MathWorks, Inc. 1 Apple Hill Drive Natick, MA 01760-2098

Communications Toolbox ${ }^{\mathrm{TM}}$ User's Guide
© COPYRIGHT 2011-2020 by The MathWorks, Inc.
The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.
FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014 Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

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

## Patents

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

April 2011
September 2011
March 2012
September 2012
March 2013
September 2013
March 2014
October 2014
March 2015
September 2015
March 2016
September 2016
March 2017
September 2017
March 2018
September 2018
March 2019
September 2019
March 2020

First printing
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only
Online only

New for Version 5.0 (Release 2011a)
Revised for Version 5.1 (Release 2011b)
Revised for Version 5.2 (Release 2012a)
Revised for Version 5.3 (Release 2012b)
Revised for Version 5.4 (Release 2013a)
Revised for Version 5.5 (Release 2013b)
Revised for Version 5.6 (Release 2014a)
Revised for Version 5.7 (Release 2014b)
Revised for Version 6.0 (Release 2015a)
Revised for Version 6.1 (Release 2015b)
Revised for Version 6.2 (Release 2016a)
Revised for Version 6.3 (Release 2016b)
Revised for Version 6.4 (Release 2017a)
Revised for Version 6.5 (Release 2017b)
Revised for Version 6.6 (Release 2018a)
Revised for Version 7.0 (Release 2018b)
Revised for Version 7.1 (Release 2019a)
Revised for Version 7.2 (Release 2019b)
Revised for Version 7.3 (Release 2020a)

Input, Output, and Display
1
Signal Terminology ..... 1-2
Matrices, Vectors, and Scalars ..... 1-2
Export Data to MATLAB ..... 1-3
Use a To Workspace Block ..... 1-3
Configure the To Workspace Block ..... 1-3
View Error Rate Data in Workspace ..... 1-4
Send Signal and Error Data to Workspace ..... 1-4
View Signal and Error Data in Workspace ..... 1-5
Analyze Signal and Error Data ..... 1-6
Sources and Sinks ..... 1-7
Data Sources ..... 1-7
Noise Sources ..... 1-9
Sequence Generators ..... 1-10
Scopes ..... 1-13
View a Sinusoid ..... 1-14
View a Modulated Signal ..... 1-16
Spreading Sequences ..... 1-21
Orthogonal Spreading for Multiuser System in Single-Path Channel ..... 1-21
Orthogonal Spreading for Single-User System in Multipath Channel ..... 1-22
PN Spreading for Single-User System in Multipath Channel ..... 1-23
PN Spreading for Multiuser System in Multipath Channel ..... 1-24
Benefits of Diversity Combining for Nonorthogonal Sequence Spreading ..... 1-25
Kasami Spreading for Multiuser System in Multipath Channel ..... 1-25
Read Signals From Hardware Devices ..... 1-27
Data and Signal Management
2
Matrices, Vectors, and Scalars ..... 2-2
Processing Rules ..... 2-2
Sample-Based and Frame-Based Processing ..... 2-4
Floating-Point and Fixed-Point Data Types ..... 2-5
Access the Block Support Table ..... 2-5
Delays ..... 2-6
Section Overview ..... 2-6
Sources of Delays ..... 2-6
ADSL Example Model ..... 2-7
Punctured Coding Model ..... 2-8
Use the Find Delay Block ..... 2-10
Digital Modulation
3
Phase Modulation ..... 3-2
Baseband and Passband Simulation ..... 3-3
BPSK ..... 3-3
QPSK ..... 3-5
Higher-Order PSK ..... 3-7
DPSK ..... 3-8
OQPSK ..... 3-8
Soft-Decision Demodulation ..... 3-10
Various User Guide Topic Examples
4
Create a Standalone GSM Waveform Explorer Application with MATLAB Compiler ..... 4-2
GSM TDMA Frame Parameterization for Waveform Generation ..... 4-5
Compensate for Frequency Offset Using Coarse and Fine Compensation ..... 4-21
Correct Symbol Timing and Doppler Offsets ..... 4-24
Estimate Turbo Code BER Performance in AWGN ..... 4-29
Random Noise Generators ..... 4-32
Visualize Effects of Frequency-Selective Fading ..... 4-36
FSK Modulation in Fading Channel ..... 4-36
QPSK Modulation in Fading Channel ..... 4-44
Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization ..... 4-51
Adjust Carrier Synchronizer Damping Factor to Correct Frequency Offset4-55
Modulate and Demodulate 8-PSK Signal ..... 4-59
Binary to Gray Conversion in Simulink ..... 4-61
What Is Bluetooth? ..... 5-2
Network Architecture ..... 5-2
Bluetooth Products ..... 5-4
Bluetooth Low Energy Core Configuration ..... 5-5
Bluetooth Protocol Stack ..... 5-7
BLE Protocol Stack ..... 5-7
Bluetooth BR/EDR Protocol Stack ..... 5-12
Bluetooth Parameterization ..... 5-16
Configuration Objects in Communications Toolbox Library for the Bluetooth Protocol ..... 5-16
Create Configuration Objects ..... 5-17
Create Link Layer Data Channel PDU Configuration Object ..... 5-17
Create Link Layer Advertising Channel PDU Configuration Object ..... 5-18
Create L2CAP Frame Configuration Object ..... 5-19
Create GAP Data Block Configuration Object ..... 5-20
Create Attribute PDU Configuration Object ..... 5-21
Bluetooth Packet Structure ..... 5-23
BLE Packet Structure ..... 5-23
Bluetooth BR/EDR Packet Structure ..... 5-30
Equalization
6
Equalization ..... 6-2
Equalizer Structure Options ..... 6-2
Selected References for Equalizers ..... 6-3
Adaptive Equalizers ..... 6-5
Symbol Tap Spacing ..... 6-5
Linear Equalizers ..... 6-6
Decision-Feedback Equalizers ..... 6-7
Reference Signal and Operating Modes ..... 6-8
Error Calculation ..... 6-8
Updating Tap Weights ..... 6-8
Configuring Adaptive Equalizers ..... 6-9
Using Adaptive Equalizers in Simulink ..... 6-29
Adaptive Equalization with Filtering and Fading Channel ..... 6-29
MLSE Equalizers ..... 6-35
Equalize a Vector Signal in MATLAB ..... 6-35
Equalizing Signals in Continuous Operation Mode ..... 6-36
Use a Preamble or a Postamble ..... 6-39
Using MLSE Equalizers in Simulink ..... 6-40
MLSE Equalization with Dynamically Changing Channel ..... 6-40
DF Equalize QPSK-Modulated Signal in Simulink ..... 7-2
Linearly Equalize QPSK-Modulated Signal in Simulink ..... 7-6
Adaptive Equalization with Filtering and Fading Channel ..... 7-10
MLSE Equalization with Dynamically Changing Channel ..... 7-16
Adaptive Equalization ..... 7-19
Equalize BSPK Signal ..... 7-27
Compare RLS and LMS Algorithms ..... 7-31
System Design
Source Coding ..... 8-2
Represent Partitions ..... 8-2
Represent Codebooks ..... 8-2
Determine Which Interval Each Input Is In ..... 8-3
Optimize Quantization Parameters ..... 8-3
Differential Pulse Code Modulation ..... 8-4
Optimize DPCM Parameters ..... 8-6
Compand a Signal ..... 8-7
Huffman Coding ..... 8-8
Arithmetic Coding ..... 8-9
Quantize a Signal ..... 8-10
Error Detection and Correction ..... 8-13
Cyclic Redundancy Check Codes ..... 8-13
Block Codes ..... 8-16
Convolutional Codes ..... 8-29
Linear Block Codes ..... 8-53
Hamming Codes ..... 8-61
BCH Codes ..... 8-67
Reed-Solomon Codes ..... 8-71
LDPC Codes ..... 8-80
Galois Field Computations ..... 8-80
Galois Fields of Odd Characteristic ..... 8-103
Interleaving ..... 8-115
Block Interleaving ..... 8-115
Convolutional Interleaving ..... 8-119
Selected Bibliography for Interleaving ..... 8-127
Digital Modulation ..... 8-128
Digital Modulation Features ..... 8-128
Signals and Delays ..... 8-131
PM Modulation ..... 8-137
AM Modulation ..... 8-138
CPM Modulation ..... 8-142
Exact LLR Algorithm ..... 8-144
Approximate LLR Algorithm ..... 8-145
Delays in Digital Modulation ..... 8-145
Selected Bibliography for Digital Modulation ..... 8-147
Analog Passband Modulation ..... 8-148
Analog Modulation Features ..... 8-148
Represent Signals for Analog Modulation ..... 8-148
Sampling Issues in Analog Modulation ..... 8-150
Filter Design Issues ..... 8-151
Phase-Locked Loops ..... 8-153
Phase-Locked Loop Features ..... 8-153
Selected Bibliography for Synchronization ..... 8-154
Multiple-Input Multiple-Output (MIMO) ..... 8-156
Orthogonal Space-Time Block Codes (OSTBC) ..... 8-156
MIMO Fading Channel ..... 8-157
Spherical Decoding ..... 8-157
Selected Bibliography for MIMO systems ..... 8-157
Huffman Coding ..... 8-158
Create a Huffman Code Dictionary ..... 8-158
Create and Decode a Huffman Code ..... 8-159
Differential Pulse Code Modulation ..... 8-160
Section Overview ..... 8-160
DPCM Terminology ..... 8-160
Represent Predictors ..... 8-160
Example: DPCM Encoding and Decoding ..... 8-161
Optimize DPCM Parameters ..... 8-161
Compand a Signal ..... 8-163
Quantize and Compand an Exponential Signal ..... 8-163
Arithmetic Coding ..... 8-165
Represent Arithmetic Coding Parameters ..... 8-165
Create and Decode an Arithmetic Code ..... 8-165
Quantization ..... 8-167
Represent Partitions ..... 8-167
Represent Codebooks ..... 8-167
Determine Which Interval Each Input Is In ..... 8-167
Optimize Quantization Parameters ..... 8-168
Quantize a Signal ..... 8-169
OFDM with User-Specified Pilot Indices ..... 9-2
SER Simulation for OFDM Link ..... 9-6
OFDM with MIMO Simulation ..... $9-8$
Gray Coded 8-PSK ..... 9-11
Introduction ..... 9-11
Initialization ..... 9-12
Stream Processing Loop ..... 9-13
Cleanup ..... 9-14
Conclusions ..... 9-14
CPM Phase Tree ..... 9-17
Structure of the Example ..... 9-17
Results and Displays ..... 9-18
Exploring the Example ..... 9-20
Filtered QPSK vs. MSK ..... 9-21
Structure of the Example ..... 9-21
Results and Displays ..... 9-22
GMSK vs. MSK ..... 9-25
Structure of the Example ..... 9-25
Results and Displays ..... 9-26
GMSK vs. MSK ..... 9-29
Gray Coded 8-PSK ..... 9-33
Structure of the Example ..... 9-33
Gray-Coded M-PSK Modulation ..... 9-33
Exploring the Example ..... 9-35
Simulation Results ..... 9-36
Comparison with Pure Binary Coding and Theory ..... 9-37
Soft Decision GMSK Demodulator ..... 9-38
Structure of the Example ..... 9-38
The Serial GMSK Receiver ..... 9-39
Results and Displays ..... 9-39
16-PSK with Custom Symbol Mapping ..... 9-45
General QAM Modulation in AWGN Channel ..... 9-48
FM Modulate and Demodulate a Sinusoidal Signal ..... 9-50
MSK Signal Recovery ..... 10-2
MSK Signal Recovery ..... 10-8
Exploring the Model ..... 10-8
Results and Displays ..... 10-9
Experimenting with the Example ..... 10-12
Reed-Solomon Coding
11
Reed-Solomon Coding Part I - Erasures ..... 11-2
Reed-Solomon Coding Part II - Punctures ..... 11-7
Reed-Solomon Coding Part III - Shortening ..... 11-12
Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink ..... 11-17
Decoding with Receiver Generated Erasures ..... 11-17
Simulation and Visualization with Erasures Only ..... 11-18
BER Performance with Erasures Only ..... 11-20
Simulation with Erasures and Punctures ..... 11-20
BER Performance with Erasures and Punctures ..... 11-21
Specifying a Shortened Code ..... 11-22
Simulation with Erasures, Punctures, and Shortening ..... 11-22
BER Performance with Erasures, Punctures, and Shortening ..... 11-23
Further Exploration ..... 11-23
Estimate LDPC Performance in AWGN ..... 11-24
Character Representation of Polynomials ..... 11-26
Estimate BER of 8-PSK in AWGN with Reed-Solomon Coding ..... 11-27
Transmit and Receive Shortened Reed-Solomon Codes ..... 11-29
Galois Fields
12
Working with Galois Fields ..... 12-2
Creating Galois Field Arrays ..... 12-2
Using Galois Field Arrays ..... 12-2
Arithmetic in Galois Fields ..... 12-2
Using MATLAB® Functions with Galois Arrays ..... 12-3
Hamming Code Example ..... 12-4ElGamal Public Key Cryptosystem

## Convolutional Coding

## 13

Punctured Convolutional Coding ..... 13-2
Iterative Decoding of a Serially Concatenated Convolutional Code ..... 13-6
Exploring the Example ..... 13-6
Variables in the Example ..... 13-7
Creating a Serially Concatenated Code ..... 13-8
Convolutional Encoding Details ..... 13-8
Decoding Using an Iterative Process ..... 13-8
Computations in Each Iteration ..... 13-8
Results of the Iterative Loop ..... 13-9
Results and Displays ..... 13-9
Punctured Convolutional Encoding ..... 13-11
Structure of the Example ..... 13-11
Generating Random Data ..... 13-12
Convolutional Encoding with Puncturing ..... 13-12
Transmitting Data ..... 13-13
Demodulating ..... 13-13
Viterbi Decoding of Punctured Codes ..... 13-13
Calculating the Error Rate ..... 13-13
Evaluating Results ..... 13-14
Rate 2/3 Convolutional Code in AWGN ..... 13-16
Estimate BER for Hard and Soft Decision Viterbi Decoding ..... 13-18
Creation, Validation, and Testing of User Defined Trellis Structure ..... 13-21
Create User Defined Trellis Structure ..... 13-21
Convolutional Encoder with Uncoded Bits and Feedback ..... 13-25
Channel Modeling and RF Impairments
14
AWGN Channel ..... 14-2
Section Overview ..... 14-2
AWGN Channel Noise Level ..... 14-2
Configure Eb/No for AWGN Channels with Coding ..... 14-5
Using AWGN Channel Block for Coded Signals ..... 14-7
Fading Channels ..... 14-8
Overview of Fading Channels ..... $14-8$
Methodology for Simulating Multipath Fading Channels ..... 14-10
Specify Fading Channels ..... 14-13
Specify Doppler Spectrum of Fading Channel ..... 14-16
Configure Channel Objects ..... 14-22
Use Fading Channels ..... 14-24
Rayleigh Fading Channel ..... 14-25
Rician Fading Channel ..... 14-30
Using Channel Visualization ..... 14-34
MIMO Channel ..... 14-35
WINNER II Channel ..... 14-36
Mapping of WINNER II Open Source Download to WINNER II Channel Model for Communications Toolbox ..... 14-37
RF Impairments ..... 14-40
Illustrate RF Impairments That Distort a Signal ..... 14-40
Phase/Frequency Offsets and Phase Noise ..... 14-43
Receiver Thermal Noise and Free Space Path Loss ..... 14-43
Nonlinearity and I/Q Imbalance ..... 14-43
Apply Nonlinear Distortion to Input Signal ..... 14-44
View Phase Noise Effects on Signal Spectrum ..... 14-44
Add RF Impairments to DQPSK Signal ..... 14-48
Selected Bibliography for Channel Modeling ..... 14-51
Measurements
15
Bit Error Rate (BER) ..... 15-2
Theoretical Results ..... 15-2
Performance Results via Simulation ..... 15-16
Performance Results via the Semianalytic Technique ..... 15-18
Theoretical Performance Results ..... 15-21
Error Rate Plots ..... 15-25
BERTool ..... 15-28
Error Vector Magnitude (EVM) ..... 15-64
Measuring Modulator Accuracy ..... 15-64
Modulation Error Ratio (MER) ..... 15-68
Adjacent Channel Power Ratio (ACPR) ..... 15-69
Obtain ACPR Measurements ..... 15-69
Complementary Cumulative Distribution Function CCDF ..... 15-75
Selected Bibliography for Measurements ..... 15-76
Filtering ..... 16-2
Filter Features ..... 16-2
Selected Bibliography Filtering ..... 16-3
Group Delay ..... 16-4
Implications of Delay for Simulations ..... 16-4
Pulse Shaping Using a Raised Cosine Filter ..... 16-6
Design Raised Cosine Filters Using MATLAB Functions ..... 16-10
Section Overview ..... 16-10
Example Designing a Square-Root Raised Cosine Filter ..... 16-10
Filter Using Simulink Raised Cosine Filter Blocks ..... 16-12
Combining Two Square-Root Raised Cosine Filters ..... 16-12
Design Raised Cosine Filters in Simulink ..... 16-16
Reduce ISI Using Raised Cosine Filtering ..... 16-21
Find Delay for Encoded and Filtered Signal ..... 16-25
Visual Analysis
17
Constellation Visualization ..... 17-2
Observe Modulator Design Affect Signal Constellation ..... 17-2
Plot Signal Constellations ..... 17-8
Create 16-PSK Constellation Diagram ..... 17-8
Create 32-QAM Constellation Diagram ..... 17-9
Create 8-QAM Gray Coded Constellation Diagram ..... 17-9
Plot a Triangular Constellation for QAM ..... 17-10
Eye Diagram Analysis ..... 17-12
Amplitude Measurements ..... 17-13
Time Measurements ..... 17-17
Scatter Plots and Constellation Diagrams ..... 17-23
View Signals Using Constellation Diagrams ..... 17-23
Channel Visualization ..... 17-30
Impulse Response ..... 17-30
Frequency Response ..... 17-32
Doppler Spectrum ..... 17-33
Visualize RF Impairments ..... 17-35
Generate C Code from MATLAB Code ..... 18-2
Set Up the Compiler ..... 18-2
Break Out the Computational Part of the Algorithm into a MATLAB Function ..... 18-2
Make Code Suitable for Code Generation ..... 18-3
Compare the MEX Function with the Simulation ..... 18-5
Generate a Standalone Executable ..... 18-5
Read and Verify the Binary File Data ..... 18-7
Relocate Code to Another Development Environment ..... 18-8
Generate C Code from Simulink Model ..... 18-9
Open the Model ..... 18-9
Configure Model for Code Generation ..... 18-10
Simulate the Model ..... 18-10
Generate Code from the Model ..... 18-11
Build and Run the Generated Code ..... 18-12
HDL Code Generation
19
HDL Code Generation Support for Communications Toolbox ..... 19-2
Blocks ..... 19-2
System Objects ..... 19-3
Find Blocks and System Objects Supporting HDL Code Generation ..... 19-4
Blocks ..... 19-4
System Objects ..... 19-5
Wireless Communications Design for FPGAs and ASICs ..... 19-6
From Mathematical Algorithm to Hardware Implementation ..... 19-6
HDL-Optimized Blocks ..... 19-8
Reference Applications ..... 19-8
Generate HDL Code and Prototype on FPGA ..... 19-9
Simulation Acceleration
20
Simulation Acceleration Using GPUs ..... 20-2
GPU-Based System objects ..... 20-2
General Guidelines for Using GPUs ..... 20-2
Transmit and decode using BPSK modulation and turbo coding ..... 20-3
Process Multiple Data Frames Using a GPU ..... 20-4
Process Multiple Data Frames Using NumFrames Property ..... 20-4
gpuArray and Regular MATLAB Numerical Arrays ..... 20-5
Pass gpuArray as an Input ..... 20-5

## Wireless Waveform Generator App

## RF Propagation

## 22

Site Viewer ..... 22-2
Introduction ..... 22-2
Dimension Picker ..... 22-2
Basemaps ..... 22-3
View Latitude and Longitude Coordinates of Location ..... 22-6
Limitations for Terrain and Buildings ..... 22-7
Troubleshooting Site Viewer ..... 22-8
Internet Connection Failure ..... 22-8
Graphics Environment ..... 22-8
Access Basemaps and Terrain in Site Viewer ..... 22-9
Access and Download Basemaps ..... 22-9
Access Terrain ..... 22-9
Access TIREM Software ..... 22-11

## Input, Output, and Display

Learn how to input, output and display data and signals with Communications Toolbox.

- "Signal Terminology" on page 1-2
- "Export Data to MATLAB" on page 1-3
- "Sources and Sinks" on page 1-7
- "Spreading Sequences" on page 1-21
- "Read Signals From Hardware Devices" on page 1-27


## Signal Terminology

This section defines important Communications Toolbox terms related to matrices, vectors, and scalars, as well as frame-based and sample-based processing.

## Matrices, Vectors, and Scalars

This document uses the unqualified words scalar and vector in ways that emphasize a signal's number of elements, not its strict dimension properties:

- A scalar signal contains a single element. The signal could be a one-dimensional array with one element, or a matrix of size 1-by-1.
- A vector signal contains one or more elements, arranged in a series. The signal could be a onedimensional array, a matrix that has exactly one column, or a matrix that has exactly one row. The number of elements in a vector is called its length or, sometimes, its width.

In cases when it is important for a description or schematic to distinguish among different types of scalar signals or different types of vector signals, this document mentions the distinctions explicitly. For example, the terms one-dimensional array, column vector, and row vector distinguish among three types of vector signals.

The size of a matrix is the pair of numbers that indicate how many rows and columns the matrix has. The orientation of a two-dimensional vector is its status as either a row vector or column vector. A one-dimensional array has no orientation - this is sometimes called an unoriented vector.

A matrix signal that has more than one row and more than one column is called a full matrix signal.

## Export Data to MATLAB

```
In this section...
"Use a To Workspace Block" on page 1-3
"Configure the To Workspace Block" on page 1-3
"View Error Rate Data in Workspace" on page 1-4
"Send Signal and Error Data to Workspace" on page 1-4
"View Signal and Error Data in Workspace" on page 1-5
"Analyze Signal and Error Data" on page 1-6
```


## Use a To Workspace Block

This section explains how to send data from a Simulink ${ }^{\circledR}$ model to the MATLAB ${ }^{\circledR}$ workspace so you can analyze the results of simulations in greater detail.

You can use a To Workspace block, from the DSP System Toolbox ${ }^{\text {TM }} /$ Sinks library to send data to the MATLAB workspace as a vector. For example, you can send the error rate data from the Hamming code model, described in the section Reducing the Error Rate Using a Hamming Code on page 8-62. To insert a To Workspace block into the model, follow these steps:

1 To open the model, at the MATLAB prompt, enter doc_hamming.
2 To add a To Workspace block, begin typing the name 'to workspace' in the model window and select the To Workspace block from the DSP System Toolbox/Sinks library. Connect it as shown.


Tip More than one To Workspace block is available. Be sure to select To Workspace from the DSP System Toolbox / Sinks sublibrary.

## Configure the To Workspace Block

To configure the To Workspace block, follow these steps:
1 Double-click the block to display its dialog box.
2 Type hammcode_BER in the Variable name field.

3 Type 1 in the Limit data points to last field. This limits the output vector to the values at the final time step of the simulation.
4 Click OK.
When you run a simulation, the model sends the output of the Error Rate Calculation block to the workspace as a vector of size 3, called hamming_BER. The entries of this vector are the same as those shown by the Error Rate Display block.

## View Error Rate Data in Workspace

After running a simulation, you can view the output of the To Workspace block by typing the following commands at the MATLAB prompt:

```
format short e
hammcode_BER
```

The vector output is the following:
hammcode_BER =
$5.4066 \mathrm{e}-\overline{0} 03 \quad 1.0000 \mathrm{e}+002 \quad 1.8496 \mathrm{e}+004$
The command format short e displays the entries of the vector in exponential form. The entries are as follows:

- The first entry is the error rate.
- The second entry is the total number of errors.
- The third entry is the total number of comparisons made.


## Send Signal and Error Data to Workspace

To analyze the error-correction performance of the Hamming code, send the transmitted signal, the received signal, and the error vectors, created by the Binary Symmetric Channel block, to the workspace. An example of this is shown in the following figure.


1 Type doc_channel at the MATLAB command line to open the starter model,
2 Double-click the Binary Symmetric Channel block to open its dialog box, and select Output error vector. This creates an output port for the error data.

3 Move blocks to make room so that you can insert Hamming Encoder and Hamming Decoder blocks. To find them, start typing Hamming in the model window. Select them from the options presented. These Hamming Encoder and Hamming Decoder blocks are in the Communications Toolbox/Error Detection and Correction /Block sublibrary.
4 Add three To Workspace blocks into the model window and connect them as shown in the preceding figure.

Tip More than one To Workspace block is available. Be sure to select To Workspace from the DSP System Toolbox / Sinks sublibrary.
5 Double-click the left To Workspace block.

- Type Tx in the Variable name field in the block's dialog box. The block sends the transmitted signal to the workspace as an array called Tx.
- In the Save 2-D signals as field, select 3-D array (concatenate along third dimension). This preserves each frame as a separate column of the array Tx.
- Click OK.

6 Double-click the middle To Workspace block:

- Type errors in the Variable name field.
- In the Save 2-D signals as field, select 3-D array (concatenate along third dimension). This preserves each frame as a separate column of the array Tx.
- Click OK.

7 Double-click the right To Workspace block:

- Type Rx in the Variable name field.
- In the Save 2-D signals as field, select 3-D array (concatenate along third dimension). This preserves each frame as a separate column of the array Tx.
- Click OK.


## View Signal and Error Data in Workspace

After running a simulation, you can display individual frames of data. For example, to display the tenth frame of Tx, at the MATLAB prompt type

Tx(: : : ,10)
This returns a column vector of length 4, corresponding to the length of a message word. Usually, you should not type Tx by itself, because this displays the entire transmitted signal, which is very large.

To display the corresponding frame of errors, type
errors(:,:,10)
This returns a column vector of length 7, corresponding to the length of a codeword.
To display frames 1 through 5 of the transmitted signal, type
$\operatorname{Tx}(:,:, 1: 5)$

## Analyze Signal and Error Data

You can use MATLAB to analyze the data from a simulation. For example, to identify the differences between the transmitted and received signals, type
diffs $=$ Tx~=Rx;
The vector diffs is the XOR of the vectors Tx and Rx. A 1 in diffs indicates that Tx and Rx differ at that position.

You can determine the indices of frames corresponding to message words that are incorrectly decoded with the following MATLAB command:
error_indices = find(diffs);
A 1 in the vector not_equal indicates that there is at least one difference between the corresponding frame of Tx and Rx. The vector error_indices records the indices where Tx and Rx differ. To view the first incorrectly decoded word, type

Tx(:,: ,error_indices(1))
To view the corresponding frame of errors, type
errors(:,:,error_indices(1))
Analyze this data to determine the error patterns that lead to incorrect decoding.

## Sources and Sinks

```
In this section...
"Data Sources" on page 1-7
"Noise Sources" on page 1-9
"Sequence Generators" on page 1-10
"Scopes" on page 1-13
"View a Sinusoid" on page 1-14
"View a Modulated Signal" on page 1-16
```

Communications Toolbox provides sources, sinks, and display devices that facilitate analysis of communication system performance.

## Data Sources

Use the functions and blocks listed in "Sources and Sinks" to generate random data to simulate a signal source.

## 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 random, independently chosen, and uniformly distributed in the set $\{1,3,5\}$.

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

| 5 | 1 | 1 | 1 |
| :--- | :--- | :--- | :--- |
| 5 | 1 | 5 | 3 |
| 1 | 3 | 5 | 5 |
| 5 | 5 | 3 | 5 |
| 3 | 5 | 5 | 5 |

To skew the distribution so that 1 is twice as likely to occur as either 3 or 5 , use the command below. 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 = 5x4
```

| 3 | 5 | 3 | 5 |
| :--- | :--- | :--- | :--- |
| 1 | 3 | 1 | 3 |
| 5 | 1 | 1 | 1 |
| 5 | 3 | 1 | 5 |
| 3 | 1 | 1 | 1 |

## Random Integers

In MATLAB®, the randi 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 .
$c=\operatorname{randi}([2,10], 5,4)$
c $=5 \times 4$

| 5 | 6 | 4 | 6 |
| :--- | :--- | :--- | ---: |
| 5 | 6 | 8 | 10 |
| 8 | 7 | 7 | 5 |
| 9 | 8 | 3 | 7 |
| 3 | 8 | 3 | 4 |

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.
$d=\operatorname{randi}([0,10], 5,4) ;$
$\mathrm{e}=\operatorname{randi}([010], 5,4)$;
In Simulink ${ }^{\circledR}$, the Random Integer Generator and Poisson Integer Generator blocks both generate vectors containing random nonnegative integers. The Random Integer Generator block uses a uniform distribution on a bounded range that you specify in the block mask. The Poisson Integer Generator block uses a Poisson distribution to determine its output. In particular, the output can include any nonnegative integer.

## Random Bit Error Patterns

In MATLAB, the randerr function generates matrices whose entries are either 0 or 1 . However, its options are different from those of randi, 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 = 5×4
```

| 0 | 0 | 0 | 1 |
| :--- | :--- | :--- | :--- |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

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])
```

$g=5 \times 4$

| 1 | 0 | 1 | 0 |
| :--- | :--- | :--- | :--- |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |

Adding the random matrix $g$ to your code matrix (modulo 2) introduces one or two errors into each codeword with the specified probability of occurrence for each. The probability matrix that is the third argument of randerr affects only the number of 1 s 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 = randi([0 1],100,1); % Two possible values
binarymatrix3 = randerr(100,1,[0 1; 0.5 0.5]); % No 1s, or one 1
```

In Simulink, the Bernoulli Binary Generator block generates random bits and is suitable for representing sources. The block considers each element of the signal to be an independent Bernoulli random variable. Also, different elements need not be identically distributed.

## Noise Sources

Construct noise generator blocks in Simulink to simulate communication links.

## Random Noise Generators

You can construct random noise generators to simulate channel noise by using the MATLAB Function block with random number generating functions. Construct different types of channel noise by using the following combinations.

| Distribution | Block | Function |
| :--- | :--- | :--- |
| Gaussian | MATLAB Function | wgn |
| Rayleigh | MATLAB Function | randn |
| Rician | MATLAB Function | randn |
| Uniform on a bounded interval | MATLAB Function | rand |

See "Random Noise Generators" on page 4-32 for an example of how Rayleigh and Rician distributed noise is created.

## Gaussian Noise Generator

In MATLAB®, 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 . By default, the power type in dBW and 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.

```
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 14-2 for more information.

## Sequence Generators

Use the functions, System objects, and blocks listed in "Sources and Sinks" to generate sequences for spreading or synchronization in a communication system. You can generate pseudorandom sequences, synchronization codes, and orthogonal codes. For examples comparing correlation properties of these sequence generators, see "Spreading Sequences" on page 1-21.

- "Pseudorandom Sequences" on page 1-10
- "Model PN Sequence Generation With Linear Feedback Shift Register" on page 1-11
- "Synchronization Codes" on page 1-13
- "Orthogonal Codes" on page 1-13


## Pseudorandom Sequences

You can generate pseudorandom or pseudonoise (PN) sequences using these System objects in MATLAB and these blocks in Simulink. The applications of these sequences range from multipleaccess spread spectrum communication systems to ranging, synchronization, and data scrambling.

| Sequence | System object ${ }^{\mathrm{mm}}$ | Block |
| :--- | :--- | :--- |
| Gold sequences | comm. GoldSequence | Gold Sequence Generator |
| Kasami sequences | comm. KasamiSequence | Kasami Sequence Generator |
| PN sequences | comm. PNSequence | PN Sequence Generator |

To generate pseudorandom sequences, the underlying code implements shift registers, as illustrated in this diagram.


All $r$ registers in the generator update their values at each time step according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2 . The shift register can be described by a binary polynomial in $z, g_{\mathrm{r}} \mathrm{r}^{\mathrm{r}}+\mathrm{g}_{\mathrm{r}-1} z^{\mathrm{r}-1}+\ldots+g_{0}$. The coefficient $g_{\mathrm{i}}$ is 1 if there is a connection, or 0 if there is no connection, from the ith shift register to the adder.

The coefficient $m_{\mathrm{i}}$ is 1 if there is a delay, or a 0 if there is no delay, from the ith shift register to the adder preceding the output. If the shift is zero, the $m_{0}$ switch is closed while all other $m_{k}$ switches are open.

The Kasami and PN sequence generators use this polynomial description for their generator polynomial. The Gold sequence generator uses this polynomial description for the preferred first and second generator polynomial PN sequences.

## Model PN Sequence Generation With Linear Feedback Shift Register

This example shows that sequences output from the PN Sequence Generator can be modeled using a linear feedback shift register (LFSR) built with primitive Simulink ${ }^{\circledR}$ blocks.

## PN Sequence Generation



For the chosen generator polynomial, $p(z)=z^{\wedge} 6+z+1$, the model generates a PN sequence of period 63, by using the PN Sequence Generator block and by modeling a LFSR using primitive Simulink blocks. The two parameters, Initial states and Output mask vector (or scalar shift value), are interpreted in the LFSR model schematic. The PreLoadFcn callback function is used to initialize runtime parameters. To view the callback functions, go to MODELING> SETUP> Model Settings> Model Properties, and select the Callbacks tab.

The scope output shows the two implementations produce matching PN sequences.


Using the PN Sequence Generator block allows you to easily generate PN sequences of large periods. To experiment further, open the model. Modify settings to see how the performance varies for different path delays or adjust the PN sequence generator parameters. You can experiment with different initial states, by changing the value of Initial states prior to running the simulation. For all values, the two generated sequences are the same.

## Synchronization Codes

Use the comm. BarkerCode System object and Barker Code Generator block to generate Barker codes to perform synchronization. Barker codes are subsets of PN sequences. They are short codes, with a length at most 13 , which are low-correlation sidelobes. A correlation sidelobe is the correlation of a codeword with a time-shifted version of itself.

## Orthogonal Codes

Orthogonal codes are used for spreading to benefit from their perfect correlation properties. When used in multi-user spread spectrum systems, where the receiver is perfectly synchronized with the transmitter, the despreading operation is ideal.

| Code | System object | Block |
| :--- | :--- | :--- |
| Hadamard codes | comm. HadamardCode | Hadamard Code Generator |
| OVSF codes | comm. OVSFCode | OVSF Code Generator |
| Walsh codes | comm. WalshCode | Walsh Code Generator |

## Scopes

The Comm Sinks block library contains scopes for viewing three types of signal plots:

- "Eye Diagrams" on page 1-14
- "Scatter Plots" on page 1-14
- "Signal Trajectories" on page 1-14

The following table lists the blocks and the plots they generate.

| Block Name | Plots |
| :--- | :--- |
| Eye Diagram | Eye diagram of a signal |
| Constellation Diagram | Constellation diagram and signal trajectory of a <br> signal |

## Eye Diagrams

An eye diagram is a simple and convenient tool for studying the effects of intersymbol interference and other channel impairments in digital transmission. When this software product constructs an eye diagram, it plots the received signal against time on a fixed-interval axis. At the end of the fixed interval, it wraps around to the beginning of the time axis. As a result, the diagram consists of many overlapping curves. One way to use an eye diagram is to look for the place where the eye is most widely opened, and use that point as the decision point when demapping a demodulated signal to recover a digital message.

The Eye Diagram block produces eye diagrams. This block processes discrete-time signals and periodically draws a line to indicate a decision, according to a mask parameter.

Examples appear in "View a Sinusoid" on page 1-14 and "View a Modulated Signal" on page 1-16.

## Scatter Plots

A constellation diagram of a signal plots the signal's value at its decision points. In the best case, the decision points should be at times when the eye of the signal's eye diagram is the most widely open.

The Constellation Diagram block produces a constellation diagram from discrete-time signals. An example appears in "View a Sinusoid" on page 1-14.

## Signal Trajectories

A signal trajectory is a continuous plot of a signal over time. A signal trajectory differs from a scatter plot in that the latter displays points on the signal trajectory at discrete intervals of time.

The Constellation Diagram block produces signal trajectories. The Constellation Diagram block produces signal trajectories when the ShowTrajectory property is set to true. A signal trajectory connects all points of the input signal, irrespective of the specified decimation factor (Samples per symbol)

## View a Sinusoid

The following model produces a constellation diagram and an eye diagram from a complex sinusoidal signal. Because the decision time interval is almost, but not exactly, an integer multiple of the period of the sinusoid, the eye diagram exhibits drift over time. More specifically, successive traces in the eye diagram and successive points in the scatter diagram are near each other but do not overlap.


To open the model, enter doc_eyediagram at the MATLAB command line. To build the model, gather and configure these blocks:

- Sine Wave, in the Sources library of the DSP System Toolbox (not the Sine Wave block in the Simulink Sources library)
- Set Frequency to . 502.
- Set Output complexity to Complex.
- Set Sample time to $1 / 16$.
- Constellation Diagram, in the Comm Sinks library
- On the Constellation Properties panel, set Samples per symbol to 16.
- Eye Diagram, in the Comm Sinks library
- On the Plotting Properties panel, set Samples per symbol to 16.
- On the Figure Properties panel, set Scope position to figposition([42.5 5535 35]);

Connect the blocks as shown in the preceding figure. In the Simulate section, set Stop time to 250. The Simulate section appears on multiple tabs. Running the model produces the following scatter diagram plot.


The points of the scatter plot lie on a circle of radius 1 . Note that the points fade as time passes. This is because the box next to Color fading is checked under Rendering Properties, which causes the
scope to render points more dimly the more time that passes after they are plotted. If you clear this box, you see a full circle of points.

The Constellation Diagram block displays a circular trajectory.
In the eye diagram, the upper set of traces represents the real part of the signal and the lower set of traces represents the imaginary part of the signal.


## View a Modulated Signal

This multipart example creates an eye diagram, scatter plot, and signal trajectory plot for a modulated signal. It examines the plots one by one in these sections:

- "Eye Diagram of a Modulated Signal" on page 1-16
- "Constellation Diagram of a Modulated Signal" on page 1-18
- "Signal Trajectory of a Modulated Signal" on page 1-19


## Eye Diagram of a Modulated Signal

The following model modulates a random signal using QPSK, filters the signal with a raised cosine filter, and creates an eye diagram from the filtered signal.


To open the model, enter doc_signaldisplays at the MATLAB command line. To build the model, gather and configure the following blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
- Set M-ary number to 4.
- Set Sample time to 0.01.
- QPSK Modulator Baseband, in PM in the Digital Baseband sublibrary of the Modulation library of Communications Toolbox, with default parameters
- AWGN Channel, in the Channels library of Communications Toolbox, with the following changes to the default parameter settings:
- Set Mode to Signal-to-noise ratio (SNR).
- Set SNR (dB) to 15.
- Raised Cosine Transmit Filter, in the Comm Filters library
- Set Filter shape to Normal.
- Set Rolloff factor to 0.5.
- Set Filter span in symbols to 6.
- Set Output samples per symbol to 8.
- Set Input processing to Elements as channels (sample based).
- Eye Diagram, in the Comm Sinks library
- Set Samples per symbol to 8 .
- Set Symbols per trace to 3. This specifies the number of symbols that are displayed in each trace of the eye diagram. A trace is any one of the individual lines in the eye diagram.
- Set Traces displayed to 3.
- Set New traces per display to 1 . This specifies the number of new traces that appear each time the diagram is refreshed. The number of traces that remain in the diagram from one refresh to the next is Traces displayed minus New traces per display.
- On the Rendering Properties panel, set Markers to + to indicate the points plotted at each sample. The default value of Markers is empty, which indicates no marker.
- On the Figure Properties panel, set Eye diagram to display to In-phase only.

When you run the model, the Eye Diagram displays the following diagram. Your exact image varies depending on when you pause or stop the simulation.


Three traces are displayed. Traces 2 and 3 are faded because Color fading under Rendering Properties is selected. This causes traces to be displayed less brightly the older they are. In this
picture, Trace 1 is the most recent and Trace 3 is the oldest. Because New traces per display is set to 1 , only Trace 1 is appearing for the first time. Traces 2 and 3 also appear in the previous display.

Because Symbols per trace is set to 3 , each trace contains three symbols, and because Samples per trace is set to 8 , each symbol contains eight samples. Note that trace 1 contains 24 points, which is the product of Symbols per trace and Samples per symbol. However, traces 2 and 3 contain 25 points each. The last point in trace 2, at the right border of the scope, represents the same sample as the first point in trace 1, at the left border of the scope. Similarly, the last point in trace 3 represents the same sample as the first point in trace 2. These duplicate points indicate where the traces would meet if they were displayed side by side, as illustrated in the following picture.


You can view a more realistic eye diagram by changing the value of Traces displayed to 40 and clearing the Markers field.


When the Offset parameter is set to 0 , the plotting starts at the center of the first symbol, so that the open part of the eye diagram is in the middle of the plot for most points.

## Constellation Diagram of a Modulated Signal

The following model creates a scatter plot of the same signal considered in "Eye Diagram of a Modulated Signal" on page 1-16.


To build the model, follow the instructions in "Eye Diagram of a Modulated Signal" on page 1-16 but replace the Eye Diagram block with the following block:

- Constellation Diagram, in the Communications Toolbox/Comm Sinks library
- Set Samples per symbol to 2
- Set Offset to 0. This specifies the number of samples to skip before plotting the first point.
- Set Symbols to display to 40.

When you run the simulation, the Constellation Diagram block displays the following plot.


The plot displays 30 points. Because Color fading under Rendering Properties is selected, points are displayed less brightly the older they are.

## Signal Trajectory of a Modulated Signal

The following model creates a signal trajectory plot of the same signal considered in "Eye Diagram of a Modulated Signal" on page 1-16.


To build the model, follow the instructions in "Eye Diagram of a Modulated Signal" on page 1-16 but replace the Eye Diagram block with the following block:

- Constellation Diagram , in the Communications Toolbox/Comm Sinks library
- Set Samples per symbol to 8.
- Set Symbols to display to 40. This specifies the number of symbols displayed in the signal trajectory. The total number of points displayed is the product of Samples per symbol and Symbols to display.

When you run the model, the Constellation Diagram displays a trajectory like the one below.


The plot displays 40 symbols. Because Color fading under Rendering Properties is selected, symbols are displayed less brightly the older they are.

See "Constellation Diagram of a Modulated Signal" on page 1-18 to compare the preceding signal trajectory to the scatter plot of the same signal. The Constellation Diagram block connects the points displayed by the Constellation Diagram block to display the signal trajectory.

If you increase Symbols to display to 100, the model produces a signal trajectory like the one below. The total number of points displayed at any instant is 800 , which is the product of the parameters Samples per symbol and Symbols to display.


## See Also

"Spreading Sequences" on page 1-21

## Spreading Sequences

Spreading consists of multiplying input data bits by a pseudorandom or pseudonoise (PN) sequence. The ratio of the PN sequence bit rate to the data rate is called the spreading factor. When the PN sequence has a bit rate higher than the data bit rate, the spreading factor is greater than 1 . When the spreading factor is greater than 1 , spreading input data adds redundancy to the transmission signal.

Spreading input data by using spreading sequences with low cross-correlation properties enables the receiver to resolve individual user data after despreading the received signal. Using spreading sequences with low cross-correlation properties helps resolve individual user data in a multipath environment in the presence of interference signals.

After signal synchronization on the receiver side, the received signal is multiplied by the same PN that was used by the transmitter. This operation removes the spreading from the received signal. Ideally, after this despreading, the signal for the user of interest is recovered with no further contribution by the signals of interferers. In CDMA systems, each transmitter is assigned distinct spreading codes that have low cross-correlation properties, such as the ideal orthogonal codes or any one of the PN, Gold, or Kasami sequences.

Spread-spectrum communication systems spread the transmission signal over a wide frequency band, typically much wider than the minimum bandwidth required to transmit the data. The spreading uses a waveform that appears random to anyone except the intended receiver of the transmitted signal. The waveform is actually pseudorandom in the sense that it can be generated by precise rules, yet has the statistical properties of a truly random sequence.

The following sections highlight various spreading sequences, their properties, and characteristic performance in single-user or multiuser and single-path or multipath transmission environments.

## Orthogonal Spreading for Multiuser System in Single-Path Channel

This model compares data recovery for a single-user system versus a two-user system. Transmission data passes through a single-path AWGN channel in two data streams that are independently spread by different orthogonal codes.

The model uses random binary data, which is BPSK modulated (real), spread by orthogonal codes of length 64, and then transmitted over an AWGN channel. The receiver consists of a despreader followed by a BPSK demodulator.

## Two-user Orthogonal Real Spreading with BPSK modulation



Using the same transmission data, the model calculates the BER performance for recovery of the single-user and two-user transmissions through identically configured AWGN channels.

The bit error rate results are exactly the same for the individual users in both cases. The matching error rates result from perfect despreading due to the ideal cross-correlation properties of the orthogonal codes selected.

To experiment further, open the model. Modify the settings to see how the performance varies with different Hadamard codes for the individual users.

## Orthogonal Spreading for Single-User System in Multipath Channel

This model simulates orthogonal spreading for a single-user system in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently by using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by orthogonal codes of length 64, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.

## Single-user Orthogonal Real Spreading with BPSK modulation and multiple paths



The non-ideal, auto-correlation values of the chosen orthogonal spreading codes prevent perfect resolution of the individual paths. As a consequence, BER performance is not improved by using diversity combining in the receiver. For a mulitpath example that uses PN sequences when spreading user data and uses diversity combining in the receiver, see "PN Spreading for Single-User System in Multipath Channel" on page 1-23.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different Hadamard codes.

## PN Spreading for Single-User System in Multipath Channel

This model simulates pseudo-random spreading for a single-user system in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently by using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by PN sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator. The receiver achieves gains from diversity combining due to the ideal auto-correlation properties of the PN sequences used when spreading the data.

Single-user Random Real Spreading with BPSK modulation and multiple paths


To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or adjust the PN sequence generator parameters.

## PN Spreading for Multiuser System in Multipath Channel

This model simulates pseudo-random spreading for two users in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by PN sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.

## Two-user Random Real Spreading with BPSK modulation and multiple paths



Using the same transmission data, the model calculates the performance for two-user transmissions through identically configured, multipath AWGN channels.

Because the transmissions for the individual users were spread using different PN sequences, the error rate computed for the users are different. Due to the higher cross-correlation properties of the nonorthogonal PN sequences used to spread the data, BER performance is degraded in a multipath environment. Sequences with high orthogonality, such as Hadamard and Kasami, are a better choice for multipath environments. For a mulitpath example that uses Hadamard code sequences when spreading user data, see "Orthogonal Spreading for Multiuser System in Single-Path Channel" on page 1-21. For a multipath example that uses Kasami code sequences when spreading user data, see "Kasami Spreading for Multiuser System in Multipath Channel" on page 1-25.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different PN sequences for the individual users.

## Benefits of Diversity Combining for Nonorthogonal Sequence Spreading

For the "PN Spreading for Multiuser System in Multipath Channel" on page 1-24 example, the individual user performance is degraded for the same channel conditions that were used in the "PN Spreading for Single-User System in Multipath Channel" on page 1-23 example. This is primarily due to the higher cross-correlation values between the two sequences, which prevent ideal separation. However, there are still advantages to diversity combining when using nonorthogonal sequence spreading, because the error rate for a multipath AWGN channel received using RAKE with diversity combining is nearly as good as the AWGN-only case in the "Orthogonal Spreading for Multiuser System in Single-Path Channel" on page 1-21 example.

## Kasami Spreading for Multiuser System in Multipath Channel

This model simulates Kasami sequence spreading for two users in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by Kasami sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.

Two-user Random Real Spreading with BPSK modulation and multiple paths


Using the same transmission data, the model calculates the performance for two-user transmissions through identically configured multipath AWGN channels.

The computed BER indicates transmission data spread using Kasami sequences exhibit low crosscorrelation. The Kasami sequences provide a balance between the ideal cross-correlation properties of orthogonal codes and the ideal auto-correlation properties of PN sequences.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different Kasami sequence generator settings for the individual users.

## See Also

## Read Signals From Hardware Devices

Communications Toolbox software can read a signal from external hardware devices using the Communications Toolbox support packages for software-defined radio (SDR). You can design, prototype and test SDR applications in MATLAB and Simulink with live radio signals. Use the supported hardware as a radio peripheral with the supplied bit stream and also run your own design in the FPGA with the automated targeting workflow using HDL Coder ${ }^{\mathrm{TM}}$.

Generate both fixed and user-provided bit streams with hardware support packages that support HDL targeting. For example, the Communications Toolbox Support Package for Xilinx ${ }^{\circledR}$ Zynq ${ }^{\circledR}$-Based Radio and Communications Toolbox Support Package for USRP ${ }^{\circledR}$ Embedded Series Radio can be used for this purpose.

See "Supported Hardware - Software-Defined Radio" for more information on the SDR hardware support packages.

For more information about SDRs used with MATLAB and Simulink, visit Software-Defined Radio (SDR) on the MathWorks ${ }^{\circledR}$ web site.

## Data and Signal Management

- "Matrices, Vectors, and Scalars" on page 2-2
- "Sample-Based and Frame-Based Processing" on page 2-4
- "Floating-Point and Fixed-Point Data Types" on page 2-5
- "Delays" on page 2-6


## Matrices, Vectors, and Scalars

Simulink supports matrix signals, one-dimensional arrays, sample-based processing, and frame-based processing. This section describes how Communications Toolbox processes certain kinds of matrices and signals.

This documentation uses the unqualified words scalar and vector in ways that emphasize a signal's number of elements, not its strict dimension properties:

- A scalar signal contains a single element. The signal could be a one-dimensional array with one element, or a matrix of size 1-by-1.
- A vector signal contains one or more elements, arranged in a series. The signal could be a onedimensional array, a matrix that has exactly one column, or a matrix that has exactly one row. The number of elements in a vector is called its length or, sometimes, its width.

In cases when it is important for a description or schematic to distinguish among different types of scalar signals or different types of vector signals, this document mentions the distinctions explicitly. For example, the terms one-dimensional array, column vector, and row vector distinguish among three types of vector signals.

The size of a matrix is the pair of numbers that indicate how many rows and columns the matrix has. The orientation of a two-dimensional vector is its status as either a row vector or column vector. A one-dimensional array has no orientation - this is sometimes called an unoriented vector.

A matrix signal that has more than one row and more than one column is called a full matrix signal.

## Processing Rules

The following rules indicate how the blocks in the Communications Toolbox process scalar, vector, and matrix signals.

- In their numerical computations, blocks that process scalars do not distinguish between onedimensional scalars and one-by-one matrices. If the block produces a scalar output from a scalar input, the block preserves dimension.
- For vector input signals:
- The numerical computations do not distinguish between one-dimensional arrays and M-by-1 matrices.
- Most blocks do not process row vectors and do not support multichannel functionality.
- The block output preserves dimension and orientation.
- The block treats elements of the input vector as a collection that arises naturally from the block's operation (for example, a collection of symbols that jointly represent a codeword) or as successive samples from a single time series.
- Most blocks do not process matrix signals that have more than one row and more than one column. For blocks that do, a signal in the shape of an $N$-by- $M$ matrix represents a series of $N$ successive samples from $M$ channels. An Input processing parameter on the block determines whether each element or column of the input signal is a channel.
- Some blocks, such as the digital baseband modulation blocks, can produce multiple output values for each value of a scalar input signal. A Rate options parameter on the block determines if the additional samples are output by increasing the rate of the output signal or by increasing the size of the output signal.
- Blocks that process continuous-time signals do not process frame-based inputs. Such blocks include the analog phase-locked loop blocks.

To learn which blocks processes scalar signals, vector signals, or matrices, refer to each block's individual Help page.

## Sample-Based and Frame-Based Processing

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. For more information, see "Sample- and FrameBased Concepts" (DSP System Toolbox).

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample of a distinct channel. For more information, see "What Is Sample-Based Processing?" (DSP System Toolbox).

## Floating-Point and Fixed-Point Data Types

The inputs and outputs of the communications blocks support various data types. For some blocks, changing to single outputs can lead to different results when compared with double outputs for the same set of parameters. Some blocks may naturally output different data types than what they receive (e.g. digital modulators) a signal. Refer to the individual block reference pages for details.

For more information, see "Floating-Point Numbers" (Fixed-Point Designer) and "Fixed-Point Signal Processing" (DSP System Toolbox).

## Access the Block Support Table

The Simulink Block Data Type Support for Communications Toolbox table provides details regarding capabilities and limitations pertaining to code generation, variable-sizing, and supported data types for all Communications Toolbox blocks. To access the table, type showcommblockdatatypetable at the MATLAB command line.

## Delays

## In this section...

> "Section Overview" on page 2-6
"Sources of Delays" on page 2-6
"ADSL Example Model" on page 2-7
"Punctured Coding Model" on page 2-8
"Use the Find Delay Block" on page 2-10

## Section Overview

Some models require you to know how long it takes for data in one portion of a model to influence a signal in another portion of a model. For example, when configuring an error rate calculator, you must indicate the delay between the transmitter and the receiver. If you miscalculate the delay, the error rate calculator processes mismatched pairs of data and consequently returns a meaningless result.

This section illustrates the computation of delays in multirate models and in models where the total delay in a sequence of blocks comprises multiple delays from individual blocks. This section also indicates how to use the Find Delay and Delay blocks to help deal with delays in a model.

## Other References for Delays

Other parts of this documentation set also discuss delays. For information about dealing with delays or about delays in specific types of blocks, see

- "Group Delay" on page 16-4
- Find Delay block reference page
- Delay block reference page
- Viterbi Decoder block reference page
- Derepeat block reference page

For discussions of delays in simpler examples than the ones in this section, see

- Example: A Rate 2/3 Feedforward Encoder. on page 8-43.
- Example: Soft-Decision Decoding on page 8-47. (See Delay in Received Data on page 8-50.)
- Example: Delays from Demodulation on page 8-146.


## Sources of Delays

While some blocks can determine their current output value using only the current input value, other blocks need input values from multiple time steps to compute the current output value. In the latter situation, the block incurs a delay. An example of this case is when the Derepeat block must average five samples from a scalar signal. The block must delay computing the average until it has received all five samples.

In general, delays in your model might come from various sources:

- Digital demodulators
- Convolutional interleavers or deinterleavers
- Equalizers
- Viterbi Decoder block
- Buffering, downsampling, derepeating, and similar signal operations
- Explicit delay blocks, such as Delay and Variable Integer Delay
- Filters

The following discussions include some of these sources of delay.

## ADSL Example Model

This section examines the "256-Channel ADSL" example and shows how to compute the correct Receive delay parameter value in one of the Error Rate Calculation blocks in the model. The model includes delays from convolutional interleaving and an explicit delay block. To open the ADSL example model, enter commadsl in the MATLAB Command Window.

In the ADSL example, data follows one of two parallel paths, one with a nonzero delay and the other with a delay of zero. One path includes a convolutional interleaver and deinterleaver, while the other does not. Near the end of each path is an Error Rate Calculation block, whose Receive delay parameter must reflect the delay of the given path. The rest of the discussion makes an observation about frame periods in the model on page 2-7 and then considers the path for interleaved data on page 2-7.

## Frame Periods in the Model

Before searching for individual delays, first observe that most signal lines throughout the model share the same frame period. On the Debug tab, expand Information Overlays. In the Sample Time section, select Colors. This option colors blocks and signals according to their frame periods (or sample periods, in the case of sample-based signals). All signal lines at the top level of the model are the same color, which means that they share the same frame period. As a consequence, frames are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by each frame's period) reduces to a sum.

## Path for Interleaved Data

In the transmitter portion of the model, the interleaved path is the lower branch, shown in yellow below. Similarly, the interleaved path in the receiver portion of the model is the lower branch. Near the end of the interleaved path is an Error Rate Calculation block that computes the value labeled Interleaved BER.


The following table summarizes the delays in the path for noninterleaved data. Subsequent paragraphs explain the delays in more detail and explain why the total delay on page $2-8$ relative to the Error Rate Calculation block is one frame, or 776 samples.

| Block | Delay, in Output <br> Samples from <br> Individual Block | Delay, in Frames | Delay, in Input <br> Samples to Error Rate <br> Calculation Block |
| :--- | :--- | :--- | :--- |
| Convolutional <br> Interleaver and <br> Convolutional <br> Deinterleaver pair | 40 | 1 (combined) | 776 (combined) |
| Delay | 800 |  |  |
| Total | N/A | 1 | 776 |

## Interleaving

Unlike the noninterleaved path, the interleaved path contains a Convolutional Interleaver block in the transmitter and a Convolutional Deinterleaver block in the receiver. The delay of the interleaver/ deinterleaver pair is the product of the Rows of shift registers parameter, the Register length step parameter, and one less than the Rows of shift registers parameter. In this case, the delay of the interleaver/deinterleaver pair turns out to be $5 * 2 * 4=40$ samples.

## Delay Block

The receiver portion of the interleaved path also contains a Delay block. This block explicitly causes a delay of 800 samples having the same sample time as the 40 samples of delay from the interleaver/ deinterleaver pair. Therefore, the total delay from interleaving, deinterleaving, and the explicit delay is 840 samples. These 840 samples make up one frame of data leaving the Delay block.

## Summing the Delays

No other blocks in the interleaved path of the example cause any delays. Adding the delays from the interleaver/deinterleaver pair and the Delay block indicates that the total delay in the interleaved path is one frame.

## Total Delay Relative to Error Rate Calculation Block

The Error Rate Calculation block that computes the value labeled Interleaved BER requires a Receive delay parameter value that is equivalent to one frame. The Receive delay parameter is measured in samples and each input frame to the Error Rate Calculation block contains 776 samples. Also, the frame rate at the output ports of all delay-causing blocks in the interleaved path equals the frame rate at the input of the Error Rate Calculation block. Therefore, the correct value for the Receive delay parameter is 776 samples.

## Punctured Coding Model

This section discusses a punctured coding model that includes delays from decoding, downsampling, and filtering. Two Error Rate Calculation blocks in the model work correctly if and only if their Receive delay parameters accurately reflect the delays in the model. To open the model, enter doc_punct in the MATLAB Command Window.

## Punctured Coding Model



## Frame Periods in the Model

Before searching for individual delays, if the Timing Legend pane is not already open, on the Debug tab, expand Information Overlays. In the Sample Time section, select Legend. In the Timing Legend pane >Highlight>All. Only the rightmost portion of the model differs in color from the rest of the model. This means that all signals and blocks in the model except those in the rightmost edge share the same frame period. Consequently, frames at this predominant frame rate are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by each frame's period) reduces to a sum.

The yellow blocks represent multirate systems, while the AWGN Channel block runs at a higher frame rate than all the other blocks in the model.

## Inner Error Rate Block

The block labeled Inner Error Rate, located near the center of the model, is a copy of the Error Rate Calculation block from the Comm Sinks library. It computes the bit error rate for the portion of the model that excludes the punctured convolutional code. In the portion of the model between this block's two input signals, delays come from the Tx Filter and the Rx Filter. This section explains why the Inner Error Rate block's Receive delay parameter is the total delay on page 2-10 value of 16 .

## Tx Filter Block

The block labeled Tx Filter is a copy of the Raised Cosine Transmit Filter block. It interpolates the input signal by a factor of 8 and applies a square-root raised cosine filter. The value of the block's Filter span in symbols parameter is 6, which means its group delay is 3 symbols. Since this block's sample rate increases from input port to output port, it must output an initial frame of zeros at the beginning of the simulation. Since its input frame size is 2 , the block's total delay is $2+3=5$ symbols. This corresponds to 5 samples at the block's input port.

## Rx Filter Block

The block labeled Rx Filter is a copy of the Raised Cosine Receive Filter block. It decimates its input signal by a factor of 8 and applies another square-root raised cosine filter. The value of this block's

Filter span in symbols parameter is 6 , which means its group delay is 3 symbols. At the block's output, the 3 symbols correspond to 3 samples.

## QPSK Demodulator Block

The block labeled QPSK Demodulator Baseband receives complex QPSK signals and outputs 2 bits for each complex input. This conversion to output bits doubles the cumulative delay at the input of the block.

## Summing the Delays

No other blocks in the portion of the model between the Inner Error Rate block's two input signals cause any delays. The total delay is then $(2+3+3) * 2=16$ samples. This value can be used as the Receive Delay parameter in the Inner Error Rate block.

## Outer Error Rate Block

The block labeled Outer Error Rate, located at the left of the model, is a copy of the Error Rate Calculation block from the Comm Sinks library. It computes the bit error rate for the entire model, including the punctured convolutional code. Delays come from the Tx Filter, Rx Filter, and Viterbi Decoder blocks. This section explains why the Outer Error Rate block's Receive delay parameter is the total delay on page 2-10 value of 108 .

## Filter and Downsample Blocks

The Tx Filter, Rx Filter, and Downsample blocks have a combined delay of 16 samples. For details, see "Inner Error Rate Block" on page 2-9.

## Viterbi Decoder Block

Because the Viterbi Decoder block decodes a rate $3 / 4$ punctured code, it actually reduces the delay seen at its input. This reduction is given as $16 * 3 / 4=12$ samples.

The Viterbi Decoder block decodes the convolutional code, and the algorithm's use of a traceback path causes a delay. The block processes a frame-based signal and has Operation mode set to Continuous. Therefore, the delay, measured in output samples, is equal to the Traceback depth parameter value of 96 . (The delay amount is stated on the reference page for the Viterbi Decoder block.) Because the output of the Viterbi Decoder block is precisely one of the inputs to the Outer Error Rate block, it is easier to consider the delay to be 96 samples rather than to convert it to an equivalent number of frames.

## Total Delay Relative to Outer Error Rate Block

The Outer Error Rate block requires a Receive delay parameter value that is the sum of all delays in the system. This total delay is $12+96=108$ samples.

## Use the Find Delay Block

The preceding discussions explained why certain Error Rate Calculation blocks in the models had specific Receive delay parameter values. You could have arrived at those numbers independently by using the Find Delay block. This section explains how to find the signal delay using the ADSL example model, commadsl, as an example. Applying the technique to the punctured convolutional coding example, discussed in "Punctured Coding Model" on page 2-8, would be similar.

## Using the Find Delay Block to Determine the Correct Receive Delay

Recall from "Path for Interleaved Data" on page 2-7 that the delay in the path for interleaved data is 776 samples. To have the Find Delay block compute that value for you, use this procedure:

1 Insert a Find Delay block and a Display block in the model near the Error Rate Calculation block that computes the value labeled Interleaved BER.
2 Connect the blocks as shown below.


3 Set the Find Delay block's Correlation window length parameter to a value substantially larger than 776 , such as 2000.

Note You must use a sufficiently large correlation window length or else the values produced by the Find Delay block do not stabilize at a correct value.
4 Run the simulation.
The new Display block now shows the value 776, as expected.

## Manipulate Delays

- "Delays and Alignment Problems" on page 2-11
- "Observing the Problem" on page 2-12
- "Aligning Words of a Block Code" on page 2-14
- "Aligning Words for Interleaving" on page 2-15
- "Aligning Words of a Concatenated Code" on page 2-17
- "Aligning Words for Nonlinear Digital Demodulation" on page 2-19


## Delays and Alignment Problems

Some models require you not only to compute delays but to manipulate them. For example, if a model incurs a delay between a block encoder and its corresponding decoder, the decoder might misinterpret the boundaries between the codewords that it receives and, consequently, return meaningless results. More generally, such a situation can arise when the path between paired components of a block-oriented operation (such as interleaving, block coding, or bit-to-integer conversions) includes a delay-causing operation (such as those listed in "Sources of Delays" on page 2-6).

To avoid this problem, you can insert an additional delay of an appropriate amount between the encoder and decoder. If the model also computes an error rate, then the additional delay affects that process, as described in "Delays" on page 2-6. This section uses examples to illustrate the purpose, methods, and implications of manipulating delays in a variety of circumstances.

This section illustrates the sensitivity of block-oriented operations to delays, using a small model that aims to capture the essence of the problem in a simple form. Open the model by entering doc_alignment in the MATLAB Command Window. Then run the simulation so that the Display blocks show relevant values.


In this model, two coding blocks create and decode a block code. Two copies of the Delay block create a delay between the encoder and decoder. The two Delay blocks have different purposes in this illustrative model:

- The Inherent Delay block represents any delay-causing blocks that might occur in a model between the encoder and decoder. See "Sources of Delays" on page 2-6 for a list of possibilities that might occur in a more realistic model.
- The Added Delay block is an explicit delay that you insert to produce an appropriate amount of total delay between the encoder and decoder. For example, the commadsl model contains a Delay block that serves this purpose.


## Observing the Problem

By default, the Delay parameters in the Inherent Delay and Added Delay blocks are set to 1 and 0 , respectively. This represents the situation in which some operation causes a one-bit delay between the encoder and decoder, but you have not yet tried to compensate for it. The total delay between the encoder and decoder is one bit. You can see from the blocks labeled Word and Delayed Word that the codeword that leaves the encoder is shifted downward by one bit by the time it enters the decoder. The decoder receives a signal in which the boundary of the codeword is at the second bit in the frame, instead of coinciding with the beginning of the frame. That is, the codewords and the frames that hold them are not aligned with each other.

This nonalignment is problematic because the Hamming Decoder block assumes that each frame begins a new codeword. As a result, it tries to decode a word that consists of the last bit of one output frame from the encoder followed by the first six bits of the next output frame from the encoder. You can see from the Error Rate Display block that the error rate from this decoding operation is close to $1 / 2$. That is, the decoder rarely recovers the original message correctly.

To use an analogy, suppose someone corrupts a paragraph of prose by moving each period symbol from the end of the sentence to the end of the first word of the next sentence. If you try to read such
a paragraph while assuming that a new sentence begins after a period, you misunderstand the start and end of each sentence. As a result, you might fail to understand the meaning of the paragraph.

To see how delays of different amounts affect the decoder's performance, vary the values of the Delay parameter in the Added Delay block and the Receive delay parameter in the Error Rate Calculation block and then run the simulation again. Many combinations of parameter values produce error rates that are close to $1 / 2$. Furthermore, if you examine the transmitted and received data by entering

```
[tx rx]
```

in the MATLAB Command Window, you might not detect any correlation between the transmitted and received data.

## Correcting the Delays

Some combinations of parameter values produce error rates of zero because the delays are appropriate for the system. For example:

- In the Added Delay block, set Delay to 6.
- In the Error Rate Calculation block, set Receive delay to 4.
- Run the simulation.
- Enter [tx rx] in the MATLAB Command Window.

The top number in the Error Rate Display block shows that the error rate is zero. The decoder recovered each transmitted message correctly. However, the Word and Displayed Word blocks do not show matching values. It is not immediately clear how the encoder's output and the decoder's input are related to each other. To clarify the matter, examine the output in the MATLAB Command Window. The sequence along the first column ( tx ) appears in the second column ( $r x$ ) four rows later. To confirm this, enter

```
isequal(tx(1:end-4),rx(5:end))
```

in the MATLAB Command Window and observe that the result is 1 (true). This last command tests whether the first column matches a shifted version of the second column. Shifting the MATLAB vector rx by four rows corresponds to the Error Rate Calculation block's behavior when its Receive delay parameter is set to 4 .

To summarize, these special values of the Delay and Receive delay parameters work for these reasons:

- Combined, the Inherent Delay and Added Delay blocks delay the encoded signal by a full codeword rather than by a partial codeword. Thus the decoder is correct in its assumption that a codeword boundary falls at the beginning of an input frame and decodes the words correctly. However, the delay in the encoded signal causes each recovered message to appear one word later, that is, four bits later.
- The Error Rate Calculation block compensates for the one-word delay in the system by comparing each word of the transmitted signal with the data four bits later in the received signal. In this way, it correctly concludes that the decoder's error rate is zero.
$\overline{\text { Note These are not the only parameter values that produce error rates of zero. Because the code }}$ in this model is a $(7,4)$ block code and the inherent delay value is 1 , you can set the Delay and Receive delay parameters to $7 \mathrm{k}-1$ and 4 k , respectively, for any positive integer k . It is important
that the sum of the inherent delay (1) and the added delay ( $7 \mathrm{k}-1$ ) is a multiple of the codeword length (7).


## Aligning Words of a Block Code

The ADSL example, discussed in "ADSL Example Model" on page 2-7, illustrates the need to manipulate the delay in a model so that each frame of data that enters a block decoder has a codeword boundary at the beginning of the frame. The need arises because the path between a block encoder and block decoder includes a delay-causing convolutional interleaving operation. This section explains why the model uses a Delay block to manipulate the delay between the convolutional deinterleaver and the block decoder, and why the Delay block is configured as it is. To open the ADSL example model, enter commadsl in the MATLAB Command Window.

## Misalignment of Codewords

In the ADSL example, the Convolutional Interleaver and Convolutional Deinterleaver blocks appear after the Scrambler \& FEC subsystems but before the Descrambler \& FEC subsystems. These two subsystems contain blocks that perform Reed-Solomon coding, and the coding blocks expect each frame of input data to start on a new word rather than in the middle of a word.

As discussed in "Path for Interleaved Data" on page 2-7, the delay of the interleaver/deinterleaver pair is 40 samples. However, the input to the Descrambler \& FEC subsystem is a frame of size 840 , and 40 is not a multiple of 840. Consequently, the signal that exits the Convolutional Deinterleaver block is a frame whose first entry does not represent the beginning of a new codeword. As described in "Observing the Problem" on page 2-12, this misalignment, between codewords and the frames that contain them, prevents the decoder from decoding correctly.

## Inserting a Delay to Correct the Alignment

The ADSL example solves the problem by moving the word boundary from the 41st sample of the 840sample frame to the first sample of a successive frame. Moving the word boundary is equivalent to delaying the signal. To this end, the example contains a Delay block between the Convolutional Deinterleaver block and the Descrambler \& FEC subsystem.


The Delay parameter in the Delay block is 800 because that is the minimum number of samples required to shift the 41st sample of one 840 -sample frame to the first sample of the next 840 -sample frame. In other words, the sum of the inherent 40 -sample delay (from the interleaving/deinterleaving process) and the artificial 800 -sample delay is a full frame of data, not a partial frame.

This 800 -sample delay has implications for other parts of the model, specifically, the Receive delay parameter in one of the Error Rate Calculation blocks. For details about how the delay influences the value of that parameter, see "Path for Interleaved Data" on page 2-7.

## Using the Find Delay Block

The preceding discussion explained why an 800-sample delay is necessary to correct the misalignment between codewords and the frames that contain them. Knowing that the Descrambler \& FEC subsystem requires frame boundaries to occur on word boundaries, you could have arrived at the number 800 independently by using the Find Delay block. Use this procedure:

1 Insert a Find Delay block and a Display block in the model.
2 Create a branch line that connects the input of the Convolutional Interleaver block to the sRef input of the Find Delay block.
3 Create another branch line that connects the output of the Convolutional Deinterleaver block to the sDel input of the Find Delay block.
4 Connect the delay output of the Find Delay block to the new Display block. The modified part of the model now looks like the following image (which also shows drop shadows on key blocks to emphasize the modifications).


Find Delay
5 Show the dimensions of each signal in the model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions.
6 Run the simulation.
The new Display block now shows the value 40. Also, the display of signal dimensions shows that the output from the Convolutional Deinterleaver block is a frame of length 840. These results indicate that the sequence of blocks between the Convolutional Interleaver and Convolutional Deinterleaver, inclusive, delays an 840-sample frame by 40 samples. An additional delay of 800 samples brings the total delay to 840 . Because the total delay is now a multiple of the frame length, the delayed deinterleaved data can be decoded.

## Aligning Words for Interleaving

This section describes an example that manipulates the delay before a deinterleaver, because the path between the interleaver and deinterleaver includes a delay from demodulation. To open the model, enter doc_gmskint in the MATLAB Command Window.

## Aligning Words for Interleaving



The model includes block coding, helical interleaving, and GMSK modulation. The table below summarizes the individual block delays in the model.

| Block | Delay, in Output Samples <br> from Individual Block | Reference |
| :--- | :--- | :--- |
| GMSK Demodulator Baseband | 16 | "Delays in Digital Modulation" <br> on page 8-145 |
| Helical Deinterleaver | 42 | "Delays of Convolutional <br> Interleavers" on page 8-120 |
| Delay | 5 | Delay reference page |

## Misalignment of Interleaved Words

The demodulation process in this model causes a delay between the interleaver and deinterleaver. Because the deinterleaver expects each frame of input data to start on a new word, it is important to ensure that the total delay between the interleaver and deinterleaver includes one or more full frames but no partial frames.

The delay of the demodulator is 16 output samples. However, the input to the Helical Deinterleaver block is a frame of size 21, and 16 is not a multiple of 21 . Consequently, the signal that exits the GMSK Demodulator Baseband block is a frame whose first entry does not represent the beginning of a new word. As described in "Observing the Problem" on page 2-12, this misalignment between words and the frames that contain them hinders the deinterleaver.

## Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 17 th sample of the 21 -sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by five samples. The Delay block between the GMSK Demodulator Baseband block and the Helical Deinterleaver block accomplishes such a delay. The Delay block has its Delay parameter set to 5 .

Combining the effects of the demodulator and the Delay block, the total delay between the interleaver and deinterleaver is a full 21 -sample frame of data, not a partial frame.

Checking Alignment of Block Codewords

The interleaver and deinterleaver cause a combined delay of 42 samples measured at the output from the Helical Deinterleaver block. Because the delayed output from the deinterleaver goes next to a Reed-Solomon decoder, and because the decoder expects each frame of input data to start on a new word, it is important to ensure that the total delay between the encoder and decoder includes one or more full frames but no partial frames.

In this case, the 42 -sample delay is exactly two frames. Therefore, it is not necessary to insert a Delay block between the Helical Deinterleaver block and the Binary-Output RS Decoder block.

## Computing Delays to Configure the Error Rate Calculation Blocks

The model contains two Error Rate Calculation blocks, labeled Channel Error Rate and System Error Rate. Each of these blocks has a Receive delay parameter that must reflect the delay of the path between the block's Tx and Rx signals. The following table explains the Receive delay values in the two blocks.

| Block | Receive Delay Value | Reason |
| :--- | :--- | :--- |
| Channel Error Rate | 16 | Delay of GMSK Demodulator Baseband block, in <br> samples |
| System Error Rate | $15 * 3$ | Three fifteen-sample frames: one frame from the <br> GMSK Demodulator Baseband and Delay blocks, <br> and two frames from the interleaver/deinterleaver <br> pair |

## Aligning Words of a Concatenated Code

This section describes an example that manipulates the delay between the two portions of a concatenated code decoder, because the first portion includes a delay from Viterbi decoding while the second portion expects frame boundaries to coincide with word boundaries. To open the model, enter doc_concat in the MATLAB Command Window. It uses the block and convolutional codes from the commdvbt example, but simplifies the overall design a great deal.

## Aligning Words of a Concatenated Code



The model includes a shortened block code and a punctured convolutional code. All signals and blocks in the model share the same frame period. The following table summarizes the individual block delays in the model.

| Block | Delay, in Output Samples from Individual Block |
| :--- | :--- |
| Viterbi Decoder | 136 |
| Delay | 1496 (that is, $1632-136$ ) |

## Misalignment of Block Codewords

The Viterbi decoding process in this model causes a delay between the Integer to Bit Converter block and the Bit to Integer Converter block. Because the latter block expects each frame of input data to start on a new 8 -bit word, it is important to ensure that the total delay between the two converter blocks includes one or more full frames but no partial frames.

The delay of the Viterbi Decoder block is 136 output samples. However, the input to the Bit to Integer Converter block is a frame of size 1632. Consequently, the signal that exits the Viterbi Decoder block is a frame whose first entry does not represent the beginning of a new word. As described in "Observing the Problem" on page 2-12, this misalignment between words and the frames that contain them hinders the converter block.

Note The outer decoder in this model (Integer-Output RS Decoder) also expects each frame of input data to start on a new codeword. Therefore, the misalignment issue in this model affects many concatenated code designs, not just those that convert between binary-valued and integer-valued signals.

## Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 137th sample of the 1632 -sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by 1632-136 samples. The Delay block between the Viterbi Decoder block and the Bit to Integer Converter block accomplishes such a delay. The Delay block has its Delay parameter set to 1496.

Combining the effects of the Viterbi Decoder block and the Delay block, the total delay between the interleaver and deinterleaver is a full 1632 -sample frame of data, not a partial frame.

## Computing Delays to Configure the Error Rate Calculation Blocks

The model contains two Error Rate Calculation blocks, labeled Inner Error Rate and Outer Error Rate. Each of these blocks has a Receive delay parameter that must reflect the delay of the path between the block's Tx and Rx signals. The table below explains the Receive delay values in the two blocks.

| Block | Receive Delay Value | Reason |
| :--- | :--- | :--- |
| Inner Error Rate | 136 | Delay of Viterbi Decoder block, in samples |
| Outer Error Rate | $1504(188 * 8$ bits $)$ | One 188-sample frame, from the combination of <br> the inherent delay of the Viterbi Decoder block <br> and the added delay of the Delay block |

## Aligning Words for Nonlinear Digital Demodulation

This example manipulates delay in order obtain the correct symbol synchronization of a signal so that symbol boundaries correctly align before demodulation occurs.

To open this model, type doc_nonlinear_digital_demod at the MATLAB command line.

Aligning Words for Nonlinear Digital Demodulation


This model includes a CPFSK modulation scheme and pulse shaping filter. For the demodulation to work properly, the input signal to the CPFSK demodulator block must have the correct alignment. Various blocks in this model introduce processing delays. Because of these delays, the input signal to the CPFSK demodulator block is not in the correct alignment.

Both the Raised Cosine Transmit and Receive Filter blocks introduce a delay. The delay is defined as: GroupDelay•Ts
where Ts represents the input sample time of the Raised Cosine Transmit Filter block.
The input sample time of the Raised Cosine Transmit Filter block equals the output sample time of the Raised Cosine Receive Filter block. Therefore, the total delay at the output of the Raised Cosine Receive Filter is:

## $2 \cdot G r o u p D e l a y \cdot T s$

or

as GroupDelay $=4$
The CPFSK demodulator block receives this delayed signal, and then it processes each collection of 8 samples per symbol to compute 1 output symbol. You must ensure that the CPFSK demodulator receives input samples in the correct collection of samples. For binary CPFSK with a Modulation index of $1 / 2$, the demodulator input must align along even numbers of symbols. Note that this requirement applies only to binary CPFSK with a modulation index of $1 / 2$. Other CPM schemes with different M-ary values and modulation indexes have different requirements.

To ensure that the CPFSK demodulator in this model receives the correct collection of input samples with the correct alignment, introduce a delay of 8 samples (in this example,

). The total delay at the input of the CPFSK demodulator is

, which equates to two symbol delays (2.T, where $T$ is the symbol period).
In sample-based mode, the CPFSK demodulator introduces a delay of Traceback length +1 samples at its output. In this example, Traceback length equals 16. Therefore, the total Receiver delay in the Error rate calculation block equals $17+2$ or 19. For more information, see "Delays in Digital Modulation" on page 8-145 in the Communications Toolbox User's Guide.

## Digital Modulation

## Phase Modulation

Phase modulation is a linear baseband modulation technique in which the message modulates the phase of a constant amplitude signal. Communications Toolbox provides modulators and demodulators for these phase modulation techniques:

- Phase shift keying (PSK) - Binary, quadrature, and general PSK
- Differential phase shift keying (DPSK) - Binary, quadrature, and general DPSK
- Offset QPSK (OQPSK)


To modulate input data with these techniques, you can use MATLAB functions, System objects, or Simulink blocks.

| Modulat ion Scheme | MATLAB functions | System objects | Simulink blocks |
| :---: | :---: | :---: | :---: |
| $\begin{aligned} & \text { Binary } \\ & \text { PSK } \\ & \text { (BPSK) } \end{aligned}$ |  | - comm.BPSKModulator <br> - comm.BPSKDemodulator | - BPSK Modulator Baseband <br> - BPSK Demodulator Baseband |
| Quadrat ure PSK (QPSK) |  | - comm.QPSKModulator <br> - comm.QPSKDemodulator | - QPSK Modulator Baseband <br> - QPSK Demodulator Baseband |
| General PSK | - pskmod <br> - pskdemod | - comm.PSKModulator <br> - comm.PSKDemodulator | - M-PSK Modulator Baseband <br> - M-PSK Demodulator Baseband |


| Modulat ion <br> Scheme | MATLAB functions | System objects | Simulink blocks |
| :---: | :---: | :---: | :---: |
| Different ial BPSK (DBPSK) |  | - comm.DBPSKModulator <br> - comm.DBPSKDemodulato r | - DBPSK Modulator Baseband <br> - DBPSK Demodulator Baseband |
| Different ial QPSK (DQPSK) |  | - comm.DQPSKModulator <br> - comm.DQPSKDemodulato r | - DQPSK Modulator Baseband <br> - DQPSK Demodulator Baseband |
| General DPSK | - dpskmod <br> - dpskdemod | - comm.DPSKModulator <br> - comm.DPSKDemodulator | - M-DPSK Modulator Baseband <br> - M-DPSK Demodulator Baseband |
| OQPSK |  | - comm.0QPSKModulator <br> - comm.0QPSKDemodulato r | - OQPSK Modulator Baseband <br> - OQPSK Demodulator Baseband |

## Baseband and Passband Simulation

Communications Toolbox supports baseband and passband simulation methods; however, the phase shift keying techniques support baseband simulation only.

A general passband waveform can be represented as

$$
Y_{1}(t) \cos \left(2 \pi f_{c} t+\theta\right)-Y_{2}(t) \sin \left(2 \pi f_{c} t+\theta\right),
$$

where $f_{c}$ is the carrier frequency and $\theta$ is the initial phase of the carrier signal. This equation is equal to the real part of

$$
\left[\left(Y_{1}(t)+j Y_{2}(t)\right) e^{j \theta}\right] \exp \left(j 2 \pi f_{c} t\right)
$$

In a baseband simulation, only the expression within the square brackets is modeled. The vector $y$ is a sampling of the complex signal

$$
\left(Y_{1}(t)+j Y_{2}(t)\right) e^{j \theta}
$$

## BPSK

In binary phase shift keying (BPSK), the phase of a constant amplitude signal switches between two values corresponding to binary 1 and binary 0 . The passband waveform of a BPSK signal is

$$
s_{n}(t)=\sqrt{\frac{2 E_{b}}{T_{b}}} \cos \left(2 \Pi f_{c} t+\Pi(1-n)\right) ; n \in\{0,1\},
$$

where:

- $E_{b}$ is the energy per bit.
- $T_{b}$ is the bit duration.
- $f_{c}$ is the carrier frequency.

In MATLAB, the baseband representation of a BPSK signal is

$$
s_{n}(t)=\cos (\pi n) ; n \in\{0,1\} .
$$

The BPSK signal has two phases: 0 and $п$.


The probability of a bit error in an AWGN channel is

$$
P_{b}=Q\left(\sqrt{\frac{2 E_{b}}{N_{0}}}\right),
$$

where $N_{0}$ is the noise power spectral density.

## QPSK

In quadrature phase shift keying, the message bits are grouped into 2-bit symbols, which are transmitted as one of four phases of a constant amplitude baseband signal. This grouping provides a bandwidth efficiency that is twice as great as the efficiency of BPSK. The general QPSK signal is expressed as

$$
s_{n}(t)=\sqrt{\frac{2 E_{s}}{T_{S}}} \cos \left(2 \pi f_{c} t+(2 n+1) \frac{\Pi}{4}\right) ; \quad n \in\{0,1,2,3\},
$$

where $E_{s}$ is the energy per symbol and $T_{s}$ is the symbol duration. The complex baseband representation of a QPSK signal is

$$
s_{n}(t)=\exp \left(j \Pi\left(\frac{2 n+1}{4}\right)\right) ; \quad n \in\{0,1,2,3\} .
$$

In this QPSK constellation diagram, each 2-bit sequence is mapped to one of four possible states. The states correspond to phases of $\Pi / 4,3 \pi / 4,5 \pi / 4$, and $7 \Pi / 4$.


To improve bit error rate performance, the incoming bits can be mapped to a Gray-coded ordering.

## Binary-to-Gray Mapping

| Binary Sequence | Gray-coded Sequence |
| :--- | :--- |
| 00 | 00 |
| 01 | 01 |
| 10 | 11 |
| 11 | 10 |

The primary advantage of the Gray code is that only one of the two bits changes when moving between adjacent constellation points. Gray codes can be applied to higher-order modulations, as shown in this Gray-coded QPSK constellation.


The bit error probability for QPSK in AWGN with Gray coding is

$$
P_{b}=Q\left(\sqrt{\frac{2 E_{b}}{N_{0}}}\right)
$$

which is the same as the expression for BPSK. As a result, QPSK provides the same performance with twice the bandwidth efficiency.

## Higher-Order PSK

In MATLAB, you can modulate and demodulate higher-order PSK constellations. The complex baseband form for an M-ary PSK signal using natural binary-ordered symbol mapping is

$$
s_{n}(t)=\exp \left(j \Pi\left(\frac{2 n+1}{M}\right)\right) ; \quad n \in\{0,1, \ldots, M-1\} .
$$

This 8-PSK constellation uses Gray-coded symbol mapping.


For modulation orders beyond 4, the bit error rate performance of PSK in AWGN worsens. In the following figure, the QPSK and BPSK curves overlap one another.


## DPSK

DPSK is a noncoherent form of phase shift keying that does not require a coherent reference signal at the receiver. With DPSK, the difference between successive input symbols is mapped to a specific phase. As an example, for binary DPSK (DBPSK), the modulation scheme operates such that the difference between successive bits is mapped to a binary 0 or 1 . When the input bit is 1 , the differentially encoded symbol remains the same as the previous symbol, while an incoming 0 toggles the output symbol.

The disadvantage of DPSK is that it is approximately 3 dB less energy efficient than coherent PSK. The bit error probability for DBPSK in AWGN is $P_{b}=1 / 2 \exp \left(E_{b} / N_{0}\right)$.

## OQPSK

Offset QPSK is similar to QPSK except that the time alignment of the in-phase and quadrature bit streams differs. In QPSK, the in-phase and quadrature bit streams transition at the same time. In OQPSK, the transitions have an offset of a half-symbol period as shown.


The in-phase and quadrature signals transition only on boundaries between symbols. These transitions occur at 1 -second intervals because the sample rate is 1 Hz . The following figure shows the in-phase and quadrature signals for an OQPSK signal.


For OQPSK, the quadrature signal has a $1 / 2$ symbol period offset ( 0.5 s ).
The BER for an OQPSK signal in AWGN is identical to that of a QPSK signal. The BER is

$$
P_{b}=Q\left(\sqrt{\frac{2 E_{b}}{N_{0}}}\right),
$$

where $E_{b}$ is the energy per bit and $N_{0}$ is the noise power spectral density.

## Soft-Decision Demodulation

All Communications Toolbox demodulator functions, System objects and blocks can demodulate binary data using either hard decisions or soft decisions. Two soft-decision algorithms are available: exact log-likelihood ratio (LLR) and approximate LLR. Exact LLR provides the greatest accuracy but is slower, while approximate LLR is less accurate but more efficient.

## 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:

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

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

$$
L(b)=\log \left(\frac{\sum_{s \in S_{0}} e^{-\frac{1}{\sigma^{2}}\left(\left(x-s_{x}\right)^{2}+\left(y-s_{y}\right)^{2}\right)}}{\sum_{s \in S_{1}} e^{-\frac{1}{\sigma^{2}}\left(\left(x-s_{x}\right)^{2}+\left(y-s_{y}\right)^{2}\right)}}\right)
$$

| Variable | Description |
| :---: | :--- |
| $r$ | Received signal with coordinates $(x, y)$ |
| $b$ | Transmitted bit (one of the K bits in an M-ary symbol, assuming all M symbols <br> are equally probable) |
| $S_{0}$ | 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 |
| $\sigma^{2}$ | Noise variance of baseband signal |
| $\sigma_{x}^{2}$ | Noise variance along in-phase axis |
| $\sigma_{y}^{2}$ | Noise variance along quadrature axis |

Note Noise components along the in-phase and quadrature axes are assumed to be independent and of equal power, that is, $\sigma_{x}^{2}=\sigma_{y}^{2}=\sigma^{2} / 2$.

## Approximate LLR Algorithm

Approximate LLR is computed by using 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 in [2] as:

$$
L(b)=-\frac{1}{\sigma^{2}}\left(\min _{s \in S_{0}}\left(\left(x-s_{x}\right)^{2}+\left(y-s_{y}\right)^{2}\right)-\min _{s \in S_{1}}\left(\left(x-s_{\chi}\right)^{2}+\left(y-s_{y}\right)^{2}\right)\right)
$$

## References

[1] Rappaport, Theodore S. Wireless Communications: Principles and Practice. Upper Saddle River, NJ: Prentice Hall, 1996, pp. 238-248.
[2] 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, Feb. 1998, pp. 260-264

## See Also

## Related Examples

- "Estimate BER of 8-PSK in AWGN with Reed-Solomon Coding" on page 11-27
- "Gray Coded 8-PSK" on page 9-11
- "16-PSK with Custom Symbol Mapping" on page 9-45
- "LLR vs. Hard Decision Demodulation"


## Various User Guide Topic Examples

- "Create a Standalone GSM Waveform Explorer Application with MATLAB Compiler" on page 4-2
- "GSM TDMA Frame Parameterization for Waveform Generation" on page 4-5
- "Compensate for Frequency Offset Using Coarse and Fine Compensation" on page 4-21
- "Correct Symbol Timing and Doppler Offsets" on page 4-24
- "Estimate Turbo Code BER Performance in AWGN" on page 4-29
- "Random Noise Generators" on page 4-32
- "Visualize Effects of Frequency-Selective Fading" on page 4-36
- "Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization" on page 4-51
- "Adjust Carrier Synchronizer Damping Factor to Correct Frequency Offset" on page 4-55
- "Modulate and Demodulate 8-PSK Signal" on page 4-59
- "Binary to Gray Conversion in Simulink" on page 4-61


## Create a Standalone GSM Waveform Explorer Application with MATLAB Compiler

This example shows how to use MATLAB Compiler ${ }^{\mathrm{TM}}$ to create a standalone application from the custom MATLAB ${ }^{\mathrm{TM}}$ app, GSMWaveformExplorer, which was created by using App Designer. By installing "MATLAB Runtime" (MATLAB Compiler) you can run standalone applications on systems that do not have MATLAB installed. For more information, see "Create and Run a Simple App Using App Designer" (MATLAB).

## MATLAB Simulation

Open the GSMWaveformExplorer app in MATLAB by entering:
GSMWaveformExplorer


The GSMWaveformExplorer app allows you to explore GSM TDMA frame configurations by using the gsmUplinkConfig and gsmDownlinkConfig objects and the gsmFrame function. You can select the Link direction as Uplink or Downlink. In the Timeslot tab, you can adjust the burst type and attenuation of individual timeslots. In the Waveform tab, you can adjust the samples per symbol and burst shape. Select View to visualize the time domain and spectrogram plots of the waveform.


## Compile the MATLAB Function into a Standalone Application

Compile GSMWaveformExplorer into a standalone application by using the mcc function and specifying the ' -m ' option. This step takes a few minutes to complete. The first message shown below appears only if your have a network installation and the second message appears only if you are running MATLAB Compiler with a demo license.

```
mcc('-m','GSMWaveformExplorer');
Disregard cmd.exe warnings about UNC directory pathnames.
DEMO Compiler license.
    The generated application will expire 30 days from today,
    on Sun Jul 21 12:05:58 2019.
```

You can also use the interactive applicationCompiler app to generate the standalone application.

## Run the Standalone Application

Before deploying the standalone app, you can test it on a machine that has MATLAB installed by running commands in the MATLAB Command Window. You can run the standalone GSMWaveformExplorer app on a machine that has MATLAB installed by using the system command.

- If you are running in the Windows or Linux operating system, type:
status = system(fullfile(pwd,'GSMWaveformExplorer'));
- If you are running in the Mac operating system, type:

```
status =
system(fullfile('GSMWaveformExplorer.app','Contents','MacOS','GSMWaveformExpl
orer'));
```

Running the standalone application with the system command uses the MATLAB environment and any library files needed from this installation of MATLAB. As with running the app in MATLAB, running the standalone version of the GSMWaveformExplorer app opens an GSM Waveform Explorer window that allows you to adjust the GSM TDMA frame configurations and view the waveform.

To deploy this application on a machine that does not have MATLAB installed, see "MATLAB Runtime" (MATLAB Compiler).

## GSM TDMA Frame Parameterization for Waveform Generation

This example shows how to parameterize and generate different GSM TDMA frames and multiframe structures.

## Introduction

The GSM standard [1 on page 4-0 ] specifies a TDMA frame as a combination of 8 time slots. Each time slot has a duration of $3 / 5200$ seconds (about 0.577 ms ) and a time slot number (TN) from 0 to 7 . GSM frames use GMSK modulation, where one symbol is equivalent to one bit. Each time slot is 156.25 bits long. The content of a time slot is called a burst. The transmission timing of a burst within a time slot is defined in terms of the bit number. The bit number (BN) refers to a particular bit period within a time slot. The bit with the lowest bit number is transmitted first. BN0 is the first bit period and BN156 is the last quarter-bit period. This figure shows time frames, time slots, and bursts for a GSM system [1 on page 4-0 ].


A TDMA contains eight time slots with each timeslot separated by a guard period. Each time slot can carry only one type of burst. Available burst type are: normal burst (NB), frequency correction burst (FB), synchronization burst (SB), access burst (AB), or dummy burst [2 on page 4-0 ]. The different burst types and the guard period are described in these next sections.

## Normal Burst (NB)

The normal burst consists of these bit fields and can appear in uplink or downlink frames. All tail bits are zero. Based on the specified training sequence code (TSC), the training sequence field contains one of eight possible training sequences.

```
normalBurstDescription()
```

```
ans=6\times3 table
    BitNumber LengthOfField
    _-_
-
                                    ContentsOfField
```

```
"tail bits"
```

"tail bits"
"encrypted bits"
"encrypted bits"
"training sequence bits"
"training sequence bits"
"encrypted bits"
"encrypted bits"
"tail bits"
"tail bits"
"guard period (bits)"

```
"guard period (bits)"
```


## Access Burst (AB)

The access burst consists of these bit fields and can appear in uplink frames only. All tail bits are zero.

```
accessBurstDescription()
ans=5\times3 table
    BitNumber Length0fField Contents0fField
    "0 - 7" {[ 8]} "extended tail bits"
    "8 - 48" {[ 41]} "synch. sequence bits"
    "49 - 84" {[ 36]} "encrypted bits"
    "85- 87" {[ 3]} "tail bits"
    "88 - 156" {[68.2500]} "guard period (bits)"
```


## Frequency Correction Burst (FB)

The frequency correction burst consists of these bit fields and can appear in downlink frames only. All tail bits and fixed bits are zero. Modulating all zeros with the GMSK modulator results in a constant phase rotation of -90 degrees for each symbol duration. Therefore, this burst generates an unmodulated carrier with a positive frequency offset of $1625 / 24 \mathrm{kHz}$.
frequencyCorrectionBurstDescription()

```
ans=4\times3 table
    "0 - 2"
    "3 - 144"
    "145 - 147"
    "148 - 156"
```

    BitNumber Length0fField Contents0fField
        \(\begin{array}{lr}\{[ & 3]\} \\ \{[ & 142]\} \\ \{[ & 3]\} \\ \{[8.2500]\}\end{array}\)
    "tail bits"
"fixed bits"
"tail bits"
"guard period (bits)"

## Synchronization Burst (SB)

The synchronization burst consists of these bit fields and can appear in downlink frames only. All tail bits are zero.
synchronizationBurstDescription()

```
ans=6\times3 table
    BitNumber LengthOfField
    "0 - 2"
    "3-41" {[ 39]} "encrypted bits"
    "42 - 105" {[ 64]} "extended training sequence bits"
    "106 - 144" {[ 39]} "encrypted bits"
    "145 - 147" {[ 3]} "tail bits"
    "148 - 156" {[8.2500]} "guard period (bits)"
        LengthOfField
        ContentsOfField
"tail bits"
    "3-41" {[ 39]} "encrypted bits"
```


## Dummy Burst

The dummy burst burst consists of these bit fields and can appear in downlink frames only. All tail bits are zero. Mixed bits contain a predetermined sequence of ones and zeros.

```
dummyBurstDescription()
ans=4\times3 table
    BitNumber LengthOfField Contents0fField
    "0 - 2"
    "3 - 144"
    "145 - 147"
    "148 - 156"
```



ContentsOfField

```
```

"tail bits"

```
```

"tail bits"
"mixed bits"
"mixed bits"
"tail bits"
"tail bits"
"guard period (bits)"

```
```

"guard period (bits)"

```
```


## Guard Period

The GSM standard, [3 on page 4-0 ], requires mobile stations to attenuate their transmission during the period between bursts. The ramp-up and ramp-down of the signal power level occurs during the guard periods. The useful part of a burst starts half way through the bit number 0 . The useful part ends halfway through BN87 for ABs and BN147 for NBs, FBs, SBs,and dummy bursts. This figure shows the useful and active parts of a burst.


## Generate Single Uplink Frame

Configure an uplink GSM TDMA frame using the gsmUplinkConfig object.

```
cfg = gsmUplinkConfig()
cfg =
    gsmUplinkConfig with properties:
        BurstType: [NB NB NB NB NB NB NB NB]
        SamplesPerSymbol: 16
            TSC: [0 1 2 3 4 5 6 7]
            Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
            RiseDelay: 0
                        FallTime: 2
            FallDelay: 0
```

Set time slots 2 and 5 to carry access bursts. Since MATLAB array indices start from 1, but time slots start from 0 , set the third and sixth elements of the BurstType to "AB".

```
cfg.BurstType([2 5] +1) = "AB"
```


## cfg =

gsmUplinkConfig with properties:

```
            BurstType: [NB NB AB NB NB AB NB NB]
        SamplesPerSymbol: 16
            TSC: [0 1 2 3 4 5 6 7]
            Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
            RiseDelay: 0
            FallTime: 2
            FallDelay: 0
```

Assign training sequence codes $3,5,1,7,0$, and 2 to time slots $0,1,3,4,6$, and 7 , respectively.

```
cfg.TSC([0 1 3 4 6 7] +1) = [3 5 1 7 0 2]
cfg =
    gsmUplinkConfig with properties:
```



Generate the baseband samples of the frame using the gsmFrame function.

```
x = gsmFrame(cfg);
```

Plot the frame. Get the sample rate of the generated waveform by using the gsmInfo function, and then calculate the time axis values in ms. The plot shows 8 bursts in the frame, with guard periods between each burst. As described in the Access Burst (AB) on page 4-0 section, ABs are short burst and have a wider guard period than other bursts.

```
wfInfo = gsmInfo(cfg);
Rs = wfInfo.SampleRate;
t = (0:length(x) - 1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(x))
grid on
axis([0 5 0 1.2])
title('GSM Uplink TDMA Frame - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(x)))
grid on
title('GSM Uplink TDMA Frame - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Plot the spectrogram of the frame.
figure
spectrogram(x,500,[],[],Rs,' centered')
title('GSM Uplink TDMA Frame - Spectrogram')


## Generate Single Downlink Frame

Configure a downlink GSM TDMA frame using the gsmDownlinkConfig object.

```
cfg = gsmDownlinkConfig
cfg =
    gsmDownlinkConfig with properties:
```

```
    BurstType: [NB NB NB NB NB NB NB NB]
```

    BurstType: [NB NB NB NB NB NB NB NB]
        SamplesPerSymbol: 16
        SamplesPerSymbol: 16
            TSC: [0 1 1 2 3 4 5 6 7]
            TSC: [0 1 1 2 3 4 5 6 7]
            Attenuation: [0 0 0 0 0 0 0 0]
            Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
            RiseTime: 2
            RiseDelay: 0
            RiseDelay: 0
            FallTime: 2
            FallTime: 2
            FallDelay: 0
    ```
            FallDelay: 0
```

Set time slots 0 to carry a frequency correction burst, set time slots 4 and 6 to carry dummy bursts, and set time slot 2 to be empty.

```
cfg.BurstType(0 +1) = "FB";
cfg.BurstType([4 6] +1) = "Dummy";
cfg.BurstType(2 +1) = "Off"
cfg =
    gsmDownlinkConfig with properties:
```

```
        BurstType: [FB NB Off NB Dummy NB Dummy NB]
SamplesPerSymbol: 16
            TSC: [0 1 2 3 4 5 6 7]
        Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
        RiseDelay: 0
            FallTime: 2
        FallDelay: 0
```

Generate the baseband samples of the frame using the gsmFrame function. This function inserts random bits instead of encrypted bits.

```
x = gsmFrame(cfg);
```

Plot the frame.

```
wfInfo = gsmInfo(cfg);
```

Rs = wfInfo.SampleRate;
$\mathrm{t}=$ (0:length(x) - 1)/Rs*1e3;
subplot $(2,1,1)$
plot(t,abs(x))
grid on
axis([0 50 1.2])
title('GSM Uplink TDMA Frame - Amplitude')
xlabel('Time (ms)');ylabel('Amplitude')
subplot (2,1,2)
plot(t,unwrap(angle(x)))
grid on
title('GSM Uplink TDMA Frame - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')


Plot the spectrogram of the frame. This plot shows the single tone during time slot 0 due to the FB .
figure
spectrogram(x,500,[],[],Rs,' centered')
title('GSM Uplink TDMA Frame - Spectrogram')


## Generate Multiframe Structure

Create a 51 -frame multiframe structure, as shown in the figure in the Introduction on page 4-0 section. Create three gsmDownlinkConfig objects with specified burst configurations. To assembly the 51 -frame multiframe, use the first and second gsmDownlinkConfig objects once and repeat the third gsmDownlinkConfig objects for the next 49 frames. Repeat the multiframe structure 3 times.

```
cfg1 = gsmDownlinkConfig('BurstType',["FB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
cfg2 = gsmDownlinkConfig('BurstType',["SB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
cfg3 = gsmDownlinkConfig('BurstType',["NB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
wfInfo = gsmInfo(cfg);
frameLength = wfInfo.FrameLengthInSamples;
x = zeros(frameLength*51*3,1);
for p=1:3
    x1 = gsmFrame(cfg1);
    x2 = gsmFrame(cfg2);
    x3 = gsmFrame(cfg3,49);
    x((p-1)*frameLength*51+1:p*frameLength*51) = [x1;x2;x3];
end
```


## Simulate Power Control and Propagation Loss Effects

Due to power control and unique propagation loss for each user, the power of each time slot within a frame might be different. Set the power attenuation for time slots 0,3 , and 7 to 2,6 , and 10 dB , respectively.
cfg = gsmUplinkConfig;
cfg.Attenuation([0 3 7 $]$ +1) $=\left[\begin{array}{lll}2 & 6 & 10\end{array}\right]$

```
cfg =
    gsmUplinkConfig with properties:
        BurstType: [NB NB NB NB NB NB NB NB]
        SamplesPerSymbol: 16
            TSC: [0 1 2 3 4 5 6 7]
            Attenuation: [2 0 0 6 0 0 0 10]
                RiseTime: 2
                RiseDelay: 0
                    FallTime: 2
            FallDelay: 0
x = gsmFrame(cfg);
wfInfo = gsmInfo(cfg);
Rs = wfInfo.SampleRate;
t = (0:length(x) - 1)/Rs*1e3;
plot(t, 20*log10(abs(x)))
axis([0 5 -20 5])
grid on
title('GSM Uplink TDMA Frame Power')
xlabel('Time (ms)')
ylabel('Power (dB)')
```

GSM Uplink TDMA Frame Power


## Adjust Ramp-Up and Ramp-Down Behavior

GSM bursts must ramp up and ramp down during guard periods [2] on page 4-0. The gsmFrame function implements the rise and fall characteristics of the bursts as a sinusoid. The burst ramps up
from zero to full amplitude in the number of symbol durations specified by the RiseTime property value. The resolution of RiseTime is $1 / N_{\text {sps }}$, where $N_{\text {sps }}$ represents the SamplesPerSymbol property value of the gsmDownlinkConfig object.

Adjust the ramp-up characteristics of the bursts. Since SamplesPerFrame is 16, you can specify RiseTime with a symbol duration resolution of 0.0625 . Set RiseTime to a duration of 3.125 symbols.
cfg = gsmDownlinkConfig;
cfg.RiseTime = 3.125;
Visualize and check if the rise-time characteristics are within the GSM specifications by using the gsmCheckTimeMask function.
gsmCheckTimeMask(cfg)


Move the start of the rise time duration to the left by 1.5 symbols by setting the RiseDelay to -1.5. When RiseDelay is 0 , the burst reaches full amplitude at the start of the useful part of the burst.

```
cfg.RiseDelay = -1.5;
gsmCheckTimeMask(cfg)
```



The burst ramps down from full amplitude to zero in the number of symbol durations specified by the FallTime property. The resolution of FallTime is $1 / N_{\text {sps }}$, where $N_{\text {sps }}$ represents the SamplesPerSymbol property value of the gsmDownlinkConfig object. Set FallTime to a duration of 2.75 symbols.

Move the start of the fall time to the right by 0.25 symbols durations by setting the FallDelay to 0.25 . When FallDelay is 0 , the burst starts to ramp down from full amplitude at the end of the useful part of the burst.

```
cfg = gsmDownlinkConfig;
cfg.FallTime = 2.75;
cfg.FallDelay = 0.25;
gsmCheckTimeMask(cfg)
```



## References

[1] 3GPP TS 45.001. "GSM/EDGE Physical layer on the radio path. General description." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.
[2] 3GPP TS 45.002, "GSM/EDGE Multiplexing and multiple access on the radio path." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network
[3] 3GPP TS 45.004, "GSM/EDGE Modulation." General description." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network

## Helper Functions

## normalBurstDescription

This function formats a table to show information about normal burst fields.

```
function d = normalBurstDescription()
BitNumber = ["0 - 2";"3 - 60";"61 - 86";...
    "87 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;58;26;58;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "encrypted bits";...
    "training sequence bits";...
    "encrypted bits";...
    "tail bits";...
```

"guard period (bits)"...
];

```
d = table(BitNumber,LengthOfField,ContentsOfField);
```

end

## frequencyCorrectionBurstDescription

This function formats a table to show information about frequency correction burst fields.

```
function d = frequencyCorrectionBurstDescription()
BitNumber = ["0 - 2";"3 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;142;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "fixed bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```


## synchronizationBurstDescription

This function formats a table to show information about synchronization burst fields.

```
function d = synchronizationBurstDescription()
BitNumber = ["0 - 2";"3 - 41";"42 - 105";...
    "106 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;39;64;39;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "encrypted bits";...
    "extended training sequence bits";...
    "encrypted bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
dummyBurstDescription
```

This function formats a table to show information about dummy burst fields.

```
function d = dummyBurstDescription()
BitNumber = ["0 - 2";"3 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;142;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "mixed bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
accessBurstDescription
```

This function formats a table to show information about access burst fields.

```
function d = accessBurstDescription()
BitNumber = ["0 - 7";"8 - 48";"49 - 84";...
    "85 - 87";"88 - 156"];
LengthOfField = {8;41;36;3;68.25};
ContentsOfField = [...
    "extended tail bits";...
    "synch. sequence bits";...
    "encrypted bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```


## Compensate for Frequency Offset Using Coarse and Fine Compensation

Correct for a phase and frequency offset in a noisy QAM signal using a carrier synchronizer. Then correct for the offsets using both a carrier synchronizer and a coarse frequency compensator.

Set the example parameters.

```
fs = 10000; % Symbol rate (Hz)
sps = 4; % Samples per symbol
M = 16; % Modulation order
k = log2(M); % Bits per symbol
```

Create a QAM modulator and an AWGN channel.

```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',k,'SamplesPerSymbol',sps);
```

Create a constellation diagram object to visualize the effects of the offset compensation techniques. Specify the constellation diagram to display only the last 4000 samples.

```
constdiagram = comm.ConstellationDiagram(...
    'ReferenceConstellation',qammod(0:M-1,M), ...
    'SamplesPerSymbol',sps, ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',4000, ...
    'XLimits',[-5 5],'YLimits',[-5 5]);
```

Introduce a frequency offset of 400 Hz and a phase offset of 30 degrees.

```
phaseFreq0ffset = comm.PhaseFrequency0ffset(...
    'Frequency0ffset',400,...
    'Phase0ffset',30,...
    'SampleRate',fs);
```

Generate random data symbols and apply 16-QAM modulation.

```
data = randi([0 M-1],10000,1);
modSig = qammod(data,M);
```

Create a raised cosine filter object and filter the modulated signal.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps, ...
    'Gain',sqrt(sps));
txSig = txfilter(modSig);
```

Apply the phase and frequency offset, and then pass the signal through the AWGN channel.

```
freqOffsetSig = phaseFreqOffset(txSig);
rxSig = channel(freqOffsetSig);
```

Apply fine frequency correction to the signal by using the carrier synchronizer.

```
fineSync = comm.CarrierSynchronizer('DampingFactor',0.7, ...
    'NormalizedLoopBandwidth',0.005, ...
    'SamplesPerSymbol',sps, ...
    'Modulation','QAM');
rxData = fineSync(rxSig);
```

Display the constellation diagram of the last 4000 symbols.
constdiagram(rxData)


Even with time to converge, the spiral nature of the plot shows that the carrier synchronizer has not yet compensated for the large frequency offset. The 400 Hz offset is $1 \%$ of the sample rate.

Repeat the process with a coarse frequency compensator inserted before the carrier synchronizer.
Create a coarse frequency compensator to reduce the frequency offset to a manageable level.
coarseSync = comm.CoarseFrequencyCompensator('Modulation',' QAM', 'FrequencyResolution',1,'SampleR
Pass the received signal to the coarse frequency compensator and then to the carrier synchronizer.
syncCoarse = coarseSync(rxSig);
rxData = fineSync(syncCoarse);
Plot the constellation diagram of the signal after coarse and fine frequency compensation.
constdiagram(rxData)


The received data now aligns with the reference constellation.

## See Also

comm.CarrierSynchronizer |comm.CoarseFrequencyCompensator

## Correct Symbol Timing and Doppler Offsets

Correct symbol timing and frequency offset errors by using the comm. SymbolSynchronizer and comm. CarrierSynchronizer System objects.

## Configuration

Initialize simulation parameters.

```
M = 16; % Modulation order
nSym = 2000; % Number of symbols in a packet
sps = 2; % Samples per symbol
spsFilt = 8; % Samples per symbol for filters and channel
spsSync = 2; % Samples per symbol for synchronizers
lenFilt = 10; % RRC filter length
```

Create a matched pair of root raised cosine (RRC) filter System objects for transmitter and receiver.

```
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',lenFilt, ...
    'OutputSamplesPerSymbol',spsFilt,'Gain',sqrt(spsFilt));
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',lenFilt, ...
    'InputSamplesPerSymbol',spsFilt,'DecimationFactor',spsFilt/2,'Gain',sqrt(1/spsFilt));
```

Create a phase-frequency offset System object to introduce a 100 Hz Doppler shift.

```
doppler = comm.PhaseFrequency0ffset('Frequency0ffset',100, ...
    'PhaseOffset',45,'SampleRate',1e6);
```

Create a variable delay System object to introduce timing offsets.

```
varDelay = dsp.VariableFractionalDelay;
```

Create carrier and symbol synchronizer System objects to correct for Doppler shift and timing offset, respectively.

```
carrierSync = comm.CarrierSynchronizer('SamplesPerSymbol',spsSync);
symbolSync = comm.SymbolSynchronizer( ...
    'TimingErrorDetector','Early-Late (non-data-aided)', ...
    'SamplesPerSymbol',spsSync);
```

Create constellation diagram System objects to view the results.

```
refConst = qammod(0:M-1,M,'UnitAveragePower',true);
cdReceive = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsFilt,'Title','Received Signal');
cdDoppler = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsSync,'Title','Frequency Corrected Signal');
cdTiming = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsSync,'Title','Frequency and Timing Synchronized Signal');
```


## Main Processing Loop

The main processing loop:

- Generates random symbols and apply QAM modulation.
- Filters the modulated signal.
- Applies frequency and timing offsets.
- Passes the transmitted signal through an AWGN channel.
- Filters the received signal.
- Corrects the Doppler shift.
- Corrects the timing offset.

```
for k = 1:15
    data = randi([0 M-1],nSym,1);
    modSig = qammod(data,M,'UnitAveragePower',true);
    txSig = txfilter(modSig);
    txDoppler = doppler(txSig);
    txDelay = varDelay(txDoppler,k/15);
    rxSig = awgn(txDelay,25);
    rxFiltSig = rxfilter(rxSig);
    rxCorr = carrierSync(rxFiltSig);
    rxData = symbolSync(rxCorr);
end
```


## Visualization

Plot the constellation diagrams of the received signal, frequency corrected signal, and frequency and timing synchronized signal. Specific constellation points cannot be identified in the received signal and can be only partially identified in the frequency corrected signal. However, the timing and frequency synchronized signal aligns with the expected QAM constellation points.
cdReceive(rxSig)

cdDoppler(rxCorr)

cdTiming (rxData)


## See Also

comm. CarrierSynchronizer|comm.SymbolSynchronizer

## Estimate Turbo Code BER Performance in AWGN

Simulate an end-to-end communication link employing 16-QAM using turbo codes in an AWGN channel. Estimate the bit error rate.

## Initialize Simulation

Set the modulation order and the range of Eb/No values to evaluate. Set the packet length to 500 .

```
rng(10,'twister');
M = 16; % Modulation order
bps = log2(M); % Bits per symbol
EbNo = (2:0.5:4);
pktLen = 500;
Initialize the bit error rate vector.
```

```
ber = zeros(size(EbNo));
```

```
ber = zeros(size(EbNo));
```

Create System objects for a turbo encoder and decoder pair, where the interleaver indices are supplied as input arguments.

```
turboEnc = comm.TurboEncoder('InterleaverIndicesSource','Input port');
turboDec = comm.TurboDecoder('InterleaverIndicesSource','Input port', ...
    'NumIterations',4);
```

Create an AWGN channel System object and an error rate counter System object.

```
awgnChannel = comm.AWGNChannel('NoiseMethod','Variance','Variance',1);
errorRate = comm.ErrorRate;
```

Use the packet length and turbo encoder settings to determine actual transmitted bit rate. The turbocoding objects are initialized to use rate- $1 / 2$ trellis for their constituent convolutional codes, resulting in a turbo encoder output with $\mathbf{2}$ parity bit streams, (in addition to the systematic stream) and $\mathbf{1 2}$ tail bits for the input packet. The 12 tail bits are due to the specified constraint length of 4 per constituent encoder, which leads to 3-bit outputs per stream, for a total of 4 streams (S1 P1 S2 P2).

```
rate = pktLen/(3*pktLen+4*3);
```


## Main Processing Loop

The processing loop performs the following steps:

- Generate random binary data
- Generate random interleaver indices
- Turbo encode the data
- Apply 16-QAM modulation
- Pass the modulated signal through an AWGN channel
- Demodulate the noisy signal using an LLR algorithm
- Turbo decode the data
- Calculate the error statistics

```
for k = 1:length(EbNo)
```

Initialize the error statistics vector, signal-to-noise ratio, and noise variance. Update the AWGN channel System object noise variance value.

```
    errorStats = zeros(1,3);
    EsNo = EbNo(k) + 10*log10(bps);
    snrdB = EsNo + 10*log10(rate); % in dB
    noiseVar = 1./(10.^(snrdB/10));
    awgnChannel.Variance = noiseVar;
    while errorStats(2) < 100 && errorStats(3) < le7
    % Generate random binary data
    data = randi([0 1],pktLen,1);
    % Interleaver indices
    intrlvrInd = randperm(pktLen);
    % Turbo encode the data
    encodedData = turboEnc(data,intrlvrInd);
    % Modulate the encoded data
    modSignal = qammod(encodedData,M,'InputType','bit','UnitAveragePower',true);
    % Pass the signal through the AWGN channel
    rxSignal = awgnChannel(modSignal);
    % Demodulate the received signal
    demodSignal = qamdemod(rxSignal,M,'UnitAveragePower',true,'OutputType','llr','NoiseVariar
    % Turbo decode the demodulated signal. Because the bit mapping from the
    % demodulator is opposite that expected by the turbo decoder, the
    % decoder input must use the inverse of demodulated signal.
    rxBits = turboDec(-demodSignal,intrlvrInd);
    % Calculate the error statistics
    errorStats = errorRate(data,rxBits);
end
% Save the BER data and reset the bit error rate object
    ber(k) = errorStats(1);
    reset(errorRate)
end
```

Plot the bit error rate and compare it to the uncoded bit error rate.

```
semilogy(EbNo,ber,'-o')
grid
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
uncodedBER = berawgn(EbNo,'qam',M); % Estimate of uncoded BER
hold on
semilogy(EbNo,uncodedBER)
legend('Turbo','Uncoded','location','sw')
```



## See Also

comm.TurboDecoder | comm.TurboEncoder

## Random Noise Generators

You can generate noise for communication system modeling using the MATLAB Function block with a random number generator. This example generates and displays histogram plots of Gaussian, Rayleigh, Rician, and Uniform noise.


The noise generators output 1e5-by-1 vectors every second, which is equivalent to a 0.00001 second sample time. In this model, each MATLAB Function block defines a specific noise generator using its underlying function. To view the underlying code for a MATLAB Function block in the MATLAB Editor, open the model, select the desired MATLAB Function block, and then press Ctrl+u. Each MATLAB function block contains block mask parameters that map to the function arguments in the underlying code.

For each MATLAB Function block the Samples per frame parameter maps to its underlying function argument spf. Similarly, Seed maps to seed.

The Gaussian Noise MATLAB Function block maps the Power (dBW) parameter to p, and defines the function

$$
y=\operatorname{wgn}(s p f, 1, p)
$$

The Rayleigh Noise MATLAB Function block maps the Sigma parameter to alpha, and defines the function

$$
y=\operatorname{abs}(\operatorname{alpha} *(\operatorname{randn}(\operatorname{spf}, 1)+1 i * \operatorname{randn}(\operatorname{spf}, 1)))
$$

The Rician Noise MATLAB Function block maps the Rician K-factor parameter to $K$ and the Sigma parameter to s , and defines the function

$$
m 1=\operatorname{sqrt}(2 * K) \cdot * s
$$

$$
\begin{aligned}
& m 2=0 \\
& y=\operatorname{sqrt}\left(\left(s^{2} * \operatorname{randn}(\operatorname{spf}, 1)+m 1\right)^{2}+\left(s^{2} * \operatorname{randn}(\operatorname{spf}, 1)+m 2\right)^{2}\right)
\end{aligned}
$$

The Uniform Noise MATLAB Function block maps the Noise lower bound parameter to lb and the Noise upper bound parameter to ub, and defines the function

$$
y=l b+(u b-l b) \cdot * \operatorname{rand}(s p f, 1)
$$

The model generates these histogram plots to show the noise distribution across the spectrum for each noise generator.





For further exploration, open the model and adjust one of the noise generation settings. For example, the Rician noise generator has a K-factor of 10, which causes the mean value of the noise to be larger than that of the Rayleigh distributed noise. Double-click the Rician Noise MATLAB Function block to open the block mask and change the K-factor from 10 to 2 . Rerun the model to see the noise spectrum shift.

## Visualize Effects of Frequency-Selective Fading

## FSK Modulation in Fading Channel

Pass an FSK signal through a Rayleigh multipath fading channel. Change the signal bandwidth to observe the impact of the fading channel on the FSK spectrum.

## FSK Modulation in Flat Fading

Set modulation order to 4 , the modulated symbol rate to 45 bps , and the frequency separation to 200 Hz .

```
M = 4; % Modulation order
symbolRate = 45; % Symbol rate (bps)
freqSep = 200; % Frequency separation (Hz)
```

Calculate the samples per symbol parameter, sampPerSym, as a function of the modulation order, frequency separation, and symbol rate. To avoid output signal aliasing, the product of sampPerSym and symbolRate must be greater than the product of $M$ and $f$ reqSep. Calculate the sample rate of the FSK output signal.

```
sampPerSym = ceil(M*freqSep/symbolRate);
fsamp = sampPerSym*symbolRate;
```

Create an FSK modulator.

```
fskMod = comm.FSKModulator(M, ...
    'FrequencySeparation',freqSep, ...
    'SamplesPerSymbol',sampPerSym, ...
    'SymbolRate',symbolRate);
```

Set the path delays and average path gains for the fading channel.

```
pathDelays = [0 3 10]*1e-6; % Discrete delays of three-path channel (s)
avgPathGains = [0 -3 -6]; %Average path gains (dB)
```

By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 m difference in path length. The path delays and path gains specify the average delay profile of the channel.

Create a Rayleigh channel using the defined parameters. Set the Visualization property to display the impulse and frequency responses.

```
channel = comm.RayleighChannel(...
    'SampleRate',fsamp, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',0.01, ...
    'Visualization','Impulse and frequency responses', ...
    'SamplesToDisplay','10%');
```

Generate random data symbols and apply FSK modulation.

```
data = randi([0 3],2000,1);
modSig = fskMod(data);
```

Plot the spectrum of the FSK modulated signal.

```
spectrum = dsp.SpectrumAnalyzer('SampleRate',fsamp);
spectrum(modSig)
```



The modulated signal is composed of four tones each having approximately 20 dBm peak power separated by 200 Hz .

Pass the signal through the Rayleigh fading channel and apply AWGN having a 25 dB signal-to-noise ratio.
snrdB = 25;
rxSig = awgn(channel(modSig),snrdB);



The impulse and frequency responses show that the channel behaves as though it were flat. This is because the signal bandwidth, 800 Hz , is much smaller than the coherence bandwidth, 50 kHz .

Plot the received signal spectrum.
spectrum(rxSig)


The four tones comprising the FSK signal maintain the same frequency separation and peak power levels relative to each other. The absolute peak power levels have decreased due to the fading channel.

## FSK Modulation in Frequency-Selective Fading

Increase the symbol rate to 45 kbps and the frequency separation to 200 kHz . Calculate the new samples per symbol and sample rate parameters.
symbolRate $=45 \mathrm{e} 3$;
freqSep = 200e3;
sampPerSym = ceil(M*freqSep/symbolRate);
fsamp = sampPerSym*symbolRate;
Update the FSK modulator properties.
release(fskMod)
fskMod.SymbolRate = symbolRate;
fskMod.FrequencySeparation $=$ freqSep;
Update the spectrum analyzer sample rate property, sa. SampleRate. Apply FSK modulation and plot the resulting spectrum.
release(spectrum)
spectrum. SampleRate = sampPerSym*symbolRate;

```
modSig = fskMod(data);
spectrum(modSig)
```



The spectrum has the same shape as in the flat-fading case but the four tones are now separated by 200 kHz.

Update the channel sample rate property. Pass the signal through the Rayleigh fading channel and apply AWGN.
release(channel)
channel. SampleRate = fsamp;
rxSig = awgn(channel(modSig),25);


Processing


The impulse and frequency responses show that the multipath fading is frequency selective.
Plot the received signal spectrum.
spectrum(rxSig)


There are still four identifiable tones but their relative peak power levels differ due to the frequencyselective fading. The signal bandwidth, 800 kHz , is larger than the coherence bandwidth, 50 kHz .

## QPSK Modulation in Fading Channel

Pass a QPSK signal through a Rayleigh multipath fading channel. Change the signal bandwidth to observe the impact of the fading channel on the QPSK constellation.

## QPSK Modulation in Flat Fading

Set the symbol rate parameter to 500 bps .
symbolRate $=500$;
Generate random data symbols and apply QPSK modulation.

```
data = randi([0 3],10000,1);
modSig = pskmod(data,4,pi/4,'gray');
```

Set the path delays and average path gains for the fading channel.

```
pathDelays = [0 3 10]*1e-6; % Discrete delays of three-path channel (s)
avgPathGains = [0 -3 -6]; %Average path gains (dB)
```

By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 meter difference in path length. The path delays and path gains specify the average delay profile of the channel.

Create a Rayleigh channel using the defined parameters. Set the Visualization property to display the impulse and frequency responses.
fsamp = symbolRate;
channel = comm.RayleighChannel(...
'SampleRate',fsamp, ...
'PathDelays', pathDelays, ...
'AveragePathGains',avgPathGains, ...
'MaximumDopplerShift',0.01, ...
'Visualization','Impulse and frequency responses');
Pass the signal through the Rayleigh channel and apply AWGN.
rxSig $=$ awgn(channel(modSig),25);



The impulse and frequency responses show that the channel behaves as though it were flat. This is because the signal bandwidth, 500 Hz , is much smaller than the coherence bandwidth, 50 kHz . Alternatively, the delay span of the channel ( 10 microseconds) is much smaller than the QPSK symbol period (2 milliseconds) so the resultant bandlimited impulse response is approximately flat.

Plot the constellation.

```
constDiagram = comm.ConstellationDiagram;
constDiagram(rxSig)
```



The QPSK constellation shows the effects of the fading channel; however, the signal still has four identifiable states.

## QPSK Modulation in Frequency-Selective Fading

Increase the symbol rate to 500 kbps and update the related channel property. Pass the signal through the Rayleigh channel and apply AWGN.

```
symbolRate = 500e3;
```

release(channel)
channel.SampleRate = symbolRate;
rxSig = awgn(channel(modSig),25);



The impulse and frequency responses show that the multipath fading is frequency selective.
Plot the constellation.
constDiagram(rxSig)


As the signal bandwidth is increased from 500 Hz to 500 kHz , the signal becomes highly distorted. This distortion is due to the intersymbol interference (ISI) that comes from time dispersion of the wideband signal. The delay span of the channel ( 10 microseconds) is now larger than the QPSK symbol period ( 2 microseconds) so the resultant bandlimited impulse response is no longer flat. Alternatively, the signal bandwidth is much larger than the coherence bandwidth, 50 kHz .

## Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization

Compensation of significant phase and frequency offsets for a 16-QAM signal in an AWGN channel is accomplished in two steps. First, correct the coarse frequency offset using the estimate provided by the coarse frequency compensator, and then fine-tune the correction using carrier synchronization. Because of the coarse frequency correction, the carrier synchronizer converges quickly even though the normalized bandwidth is set to a low value. Lower normalized bandwidth values enable better correction for small residual carrier offsets. After applying phase and frequency offset corrections to the received signal, resolve phase ambiguity using the preambles.

Define the simulation parameters.

```
fs = 10000; % Sample rate (Hz)
sps = 4; % Samples per symbol
M = 16; % Modulation order
k = log2(M); % Bits per symbol
rng(1996) % Set seed for repeatable results
barker = comm.BarkerCode(...
    'Length',13,'SamplesPerFrame',13); % For preamble
msgLen = 1e4;
numFrames = 10;
frameLen = msgLen/numFrames;
```

Generate data payloads and add the preamble to each frame. The preamble is later used for phase ambiguity resolution.

```
preamble = (1+barker())/2; % Length 13, unipolar
data = zeros(msgLen, 1);
for idx = 1 : numFrames
    payload = randi([0 M-1],frameLen-barker.Length,1);
    data((idx-1)*frameLen + (1:frameLen)) = [preamble; payload];
end
```

Create a System object for the transmit pulse shape filtering, the receive pulse shape filtering, the QAM coarse frequency compensation, the carrier synchronization, and a constellation diagram.

```
txFilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',sps);
rxFilter = comm.RaisedCosineReceiveFilter(...
    'InputSamplesPerSymbol',sps,'DecimationFactor',sps);
coarse = comm.CoarseFrequencyCompensator('SampleRate',fs, ...
    'FrequencyResolution',10);
fine = comm.CarrierSynchronizer( ...
    'DampingFactor',0.4,'NormalizedLoopBandwidth',0.001, ...
    'SamplesPerSymbol',1,'Modulation','QAM' );
axislimits = [-6 6];
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',qammod(0:M-1,M), ...
    'ChannelNames',{'Before convergence','After convergence'}, ...
    'ShowLegend',true,'XLimits',axislimits,'YLimits',axislimits);
```

Also create a System object for the AWGN channel, and the phase and frequency offset to add impairments to the signal. A phase offset greater than 90 degrees is added to induce a phase ambiguity that results in a constellation quadrant shift.

```
ebn0 = 8;
freqoffset = 110;
```

```
phaseoffset = 110;
awgnChannel = comm.AWGNChannel('EbNo',ebn0, ...
    'BitsPerSymbol',k,'SamplesPerSymbol',sps);
pfo = comm.PhaseFrequency0ffset('Frequency0ffset',freqoffset, ...
    'PhaseOffset',phaseoffset,'SampleRate',fs);
```

Generate random data symbols, apply 16-QAM modulation, and pass the modulated signal through the transmit pulse shaping filter.

```
txMod = qammod(data,M);
txSig = txFilter(txMod);
```

Apply phase and frequency offsets using the pfo System object, and then pass the signal through an AWGN channel to add white Gaussian noise.

```
txSigOffset = pfo(txSig);
rxSig = awgnChannel(txSigOffset);
```

The coarse frequency compensator System object provides a rough correction for the frequency offset. For the conditions in this example, correcting the frequency offset of the received signal correction to within 10 Hz of the transmitted signal is sufficient.

```
syncCoarse = coarse(rxSig);
```

Pass the signal through the receive pulse shaping filter, and apply fine frequency correction.

```
rxFiltSig = fine(rxFilter(syncCoarse));
```

Display the constellation diagram of the first and last 1000 symbols in the signal. Before convergence of the synchronization loop, the spiral nature of the diagram indicates that the frequency offset is not corrected. After the carrier synchronizer has converged to a solution, the symbols are aligned with the reference constellation.

```
constDiagram([rxFiltSig(1:1000) rxFiltSig(9001:end)])
```



Demodulate the signal. Account for the signal delay caused by the transmit and receive filters to align the received data with the transmitted data. Compute and display the total bit errors and BER. When checking the bit errors, use the later portion of the received signal to be sure the synchronization loop has converged.
rxData = qamdemod(rxFiltSig,M);
delay $=($ txFilter.FilterSpanInSymbols + rxFilter.FilterSpanInSymbols) / 2;
idxSync $=2000$; \% Check BER for the received signal after the synchronization loop has converged [syncDataTtlErr,syncDataBER] = biterr(data(idxSync:end-delay),rxData(idxSync+delay:end))
syncDataTtlErr = 16116
syncDataBER $=0.5042$
Depending on the random data used, there may be bit errors resulting from phase ambiguity in the received signal after the synchronization loop converges and locks. In this case, you can use the preamble to determine and then remove the phase ambiguity from the synchronized signal to reduce bit errors. If phase ambiguity is minimal, the number of bit errors may be unchanged.

```
idx = 9000 + (1:barker.Length);
phOffset = angle(txMod(idx) .* conj(rxFiltSig(idx+delay)));
ph0ffsetEst = mean(ph0ffset);
disp(['Phase offset = ',num2str(rad2deg(phOffsetEst)),' degrees'])
Phase offset = -90.1401 degrees
resPhzSig = exp(1i*phOffsetEst) * rxFiltSig;
Demodulate the signal after resolving the phase ambiguity. Recompute the total bit errors and BER.
resPhzData = qamdemod(resPhzSig,M);
[resPhzTtlErr,resPhzBER] = biterr(data(idxSync:end-delay),resPhzData(idxSync+delay:end))
resPhzTtlErr = 5
resPhzBER = 1.5643e-04
```


## Adjust Carrier Synchronizer Damping Factor to Correct Frequency Offset

Attempt to correct for a frequency offset using the carrier synchronizer object. Increase the damping factor of the synchronizer and determine if the offset was corrected.

Set the modulation order, sample rate, frequency offset, and signal-to-noise ratio parameters.

```
M = 8;
fs = le6;
foffset = 1000;
snrdb = 20;
```

Create a phase frequency offset object to introduce a frequency offset to a modulated signal. Create a constellation diagram object.

```
pfo = comm.PhaseFrequency0ffset('SampleRate',fs, ...
    'Frequency0ffset',foffset);
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',pskmod(0:M-1,M,pi/M));
```

Create a carrier synchronizer object to correct for the frequency offset.

```
carriersync = comm.CarrierSynchronizer('Modulation','8PSK', ...
    'DampingFactor',0.05,'NormalizedLoopBandwidth',0.01);
```

The main processing loop includes these steps:

- Generate random data.
- Apply 8-PSK modulation.
- Introduce a frequency offset.
- Pass the signal through an AWGN channel.
- Correct for the frequency offset.
- Display the constellation diagram.

```
for k = 1:200
    data = randi([0 M-1],1000,1);
    modSig = pskmod(data,M);
    txSig = pfo(modSig);
    rxSig = awgn(txSig,snrdb);
    syncOut = carriersync(rxSig);
    constDiagram(syncOut)
end
```



The constellation points cannot be clearly identified indicating that the carrier synchronizer is unable to compensate for the frequency offset.

Determine the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay by using the info function.

```
syncInfo = info(carriersync)
syncInfo =
    struct with fields:
        NormalizedPullInRange: 0.0044
        MaxFrequencyLockDelay: 78.9568
            MaxPhaseLockDelay: 130
```

Convert the normalised pull-in range from radians to cycles. Compare the normalized frequency offset to the pull-in range.

```
[foffset/fs syncInfo.NormalizedPullInRange/(2*pi)]
ans =
    1.0e-03 *
    1.0000 0.7071
```

The offset is greater than the pull-in range. This is reason that the carrier synchronizer failed to correct the frequency offset.

Change the damping factor of the synchronizer to 0.707.
carriersync.DampingFactor $=0.707$;
Repeat the main processing loop.

```
for k = 1:200
    data = randi([0 M-1],1000,1);
    modSig = pskmod(data,M);
    txSig = pfo(modSig);
    rxSig = awgn(txSig,snrdb);
    syncOut = carriersync(rxSig);
    constDiagram(syncOut)
end
```



There are now eight observable clusters, which shows that the frequency offset was corrected.
Determine the new pull-in range. The normalized offset is less than the pull-in range. This explains why the carrier synchronizer was able to correct the offset.

```
syncInfo = info(carriersync);
[foffset/fs syncInfo.NormalizedPullInRange/(2*pi)]
ans =
    0.0010 0.0100
```


## Modulate and Demodulate 8-PSK Signal

Open the 8-PSK model. The model generates an 8-PSK signal, applies white noise, displays the resulting constellation diagram, and computes the error statistics.


Run the model.


The error statistics are collected in vector ErrorVec. Because Eb/No is 15 dB , there are no measured symbol errors.

```
Number of symbol errors = 0
```

Change the Eb/No of the AWGN Channel block from 15 dB to 5 dB . The increase in the noise is shown in the constellation diagram.


Because of the increase in the noise level, the number of symbol errors is greater than zero.

```
Number of symbol errors = 13
```


## Binary to Gray Conversion in Simulink

Open the Binary-to-Gray model. The model converts a binary sequence to a Gray-coded sequence and vice versa by using Data Mapper blocks.

Run the model.
The Display blocks show the natural binary and Gray-coded sequence ordering.


## Bluetooth Tutorials

- "What Is Bluetooth?" on page 5-2
- "Bluetooth Protocol Stack" on page 5-7
- "Bluetooth Parameterization" on page 5-16
- "Create Configuration Objects" on page 5-17
- "Bluetooth Packet Structure" on page 5-23


## What Is Bluetooth?

Bluetooth wireless technology is the air interface intended to replace the cables connecting portable and fixed electronic equipment. Bluetooth device manufacturers have the flexibility to include optional core specification features to optimize and differentiate product offers.

Bluetooth is equated with the implementation specified by the Bluetooth Core Specification group of standards maintained by the Bluetooth Special Interest Group (SIG) industry consortium.
Communications Toolbox Library for the Bluetooth Protocol functionality enables you to model Bluetooth Low Energy (BLE) communications system links, as specified in the Core System Package [Low Energy Controller volume], Specification Volume 6. It also enables you to explore variations on implementations for future evolution of the standard.

## Network Architecture

- "Point-to-Point and Point-to-Multipoint Connection Topology" on page 5-2
- "Broadcast Connection Topology" on page 5-3
- "Mesh Connection Topology" on page 5-3

Network topologies supported in Bluetooth include point-to-point, broadcast, and mesh connectivity. Point-to-point connectivity is available for devices that operate in classic Bluetooth or BLE. The broadcast and mesh connection topologies are only supported for BLE devices.

## Point-to-Point and Point-to-Multipoint Connection Topology

Devices using point-to-point communication operate in a piconet. Point-to-point piconets define one-to-one device communication links. Examples of point-to-point links are links between PCs or mobile phones and peripherals such as headsets, printers, and fitness trackers. Multiple piconets connect to one another in a scatternet topology. Point-to-multipoint piconets define one to more than one device communication links.

This image shows a scatternet of three piconets. Each piconet shows one device in the role of master (M), with other devices in the slave (S) or idle (I) roles. The image also shows one device (M/S) assigned the master role in one piconet and slave in another piconet.


- A scatternet is an ad hoc network consisting of two or more piconets.
- A piconet is defined as a connection between two or more Bluetooth devices. Piconets nets are limited to a maximum of eight devices, with one master taking the master role at any given time and seven slaves.
- The individual Bluetooth devices assume the role of master, slave, or idle peer devices in a given piconet. An individual Bluetooth device can take the role of a slave in one piconet while taking the role of master in another piconet.
- The master device provides the synchronization reference.
- The slaves are other devices that synchronize to the clock and frequency hopping pattern of the master.
- Other idle devices may be located in a piconet but are not active.


## Broadcast Connection Topology

Broadcast piconets establish one-to-many communication links for BLE devices. Examples of broadcast links are retail point-of-interest information, indoor navigation, and asset tracking.

## Mesh Connection Topology

Mesh networks establish the option of many-to-many communication links for BLE devices. Mesh topology enables the creation of large-scale device networks. Mesh is ideally suited for control, monitoring, and automation systems that require reliable and secure communication between thousands of devices.

The Bluetooth SIG specifies mesh networking requirements to enable an interoperable many-to-many ( $\mathrm{m}: \mathrm{m}$ ) mesh networking solution for Bluetooth Low Energy (LE) wireless technology. Mesh networks are ideally suited for large-scale device networks supporting building automation, sensor networks, asset tracking, and other solutions requiring reliable and secure communication between multiple devices. Bluetooth SIG adopted these specifications.

- Mesh Profile Specification - Defines fundamental requirements to enable an interoperable mesh networking solution for Bluetooth LE wireless technology.
- Mesh Model Specification - Introduces models, used to define basic functionality of nodes on a mesh network.
- Mesh Device Properties - Defines device properties required for the Mesh Model specification.

The Bluetooth Mesh Model Specification defines categories of BLE mesh models. Model categories include

- Foundation models
- Generic models
- Sensors
- Time and scenes
- Lighting

All devices must implement the foundation models for configuration server and health server. All other models in the model categories are optional and implemented based on the service the BLE device performs.

As described in the Mesh Profile Bluetooth specification, mesh networks operate as managed-floodbased networks. Devices use broadcast channels to transmit messages to other devices, and the messages are relayed forward to other devices extending the range of the original message.


A device that is not a member of a mesh network is referred to as an unprovisioned device. A device that is a member of a mesh network is known as a node. Devices are added to a mesh network by a Provisioner. Nodes in a mesh network share network keys that enable them to receive and to relay messages from other nodes in their network or subnet. Network keys are used to secure and authenticate messages at the network layer. Unprovisioned devices cannot receive messages because they do not have the network key to recover the message.

## Bluetooth Products

The Bluetooth SIG defines a Bluetooth product as any product containing an implementation of Bluetooth wireless technology. Bluetooth products are classified as:

- Bluetooth End Product
- Bluetooth Host Subsystem Product
- Bluetooth Controller Subsystem Product
- Bluetooth Profile Subsystem Product
- Bluetooth Component Product
- Bluetooth Development Tool
- Bluetooth Test Equipment

The Communications Toolbox Library for the Bluetooth Protocol provides features enabling you to model Bluetooth Host and Controller Subsystem Products fully compliant Bluetooth links with the low energy (LE) core configuration.

## Bluetooth Low Energy Core Configuration

The Bluetooth Core Specification, Volume 0, Part B, Section 4.4 specifies a set of required features that must be implemented to model fully compliant Bluetooth links with the low energy (LE) core configuration.

The LE core configuration defines three main layers - Application, Host, and Controller. The Communications Toolbox Library for the Bluetooth Protocol provides features to model the host and controller layers. Requirements defined in the Bluetooth Core Specification for the host and controller include

| Layer | Sublayer | Bluetooth <br> Specification Volume | Required Features <br> HostLogical link control and <br> adaptation protocol <br> (L2CAP) |
| :--- | :--- | :--- | :--- |
|  | Volume 3, part A | If the GAP Peripheral or <br> Central role is <br> supported, L2CAP LE <br> Signaling Channel (CID <br> 0x0005) and all <br> mandatory features <br> associated with it. |  |
|  | Generic access profile <br> (GAP) | Volume 3, part C | All mandatory features <br> for at least one of the <br> LE GAP roles |
| (Broadcaster, Observer, |  |  |  |
| Peripheral, or Central) |  |  |  |
| in sections 9-12 and |  |  |  |
| section 15. |  |  |  |$|$| Attribute profile (ATT) |
| :--- |
|  |

For a description of the mapping between Bluetooth protocol stack functionality and the OSI reference model, see "Bluetooth Protocol Stack" on page 5-7.

## References

[1] https://www.bluetooth.com/
[2] "Bluetooth Core Specification." Bluetooth Special Interest Group (SIG).
[3] "Supplement to the Bluetooth Core Specification, CSS Version 7." Bluetooth Special Interest Group (SIG).
[4] "Bluetooth Core Specification Addendum 6." Bluetooth Special Interest Group (SIG).
[5] "Mesh Profile Bluetooth Specification." Bluetooth Special Interest Group (SIG).
[6] "Mesh Model Bluetooth Specification." Bluetooth Special Interest Group (SIG).
[7] "Mesh Device Properties Bluetooth Specification." Bluetooth Special Interest Group (SIG).

## Bluetooth Protocol Stack

The Bluetooth Special Interest Group (SIG) [1] and [2] defines the protocol stack for Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR) technology. The fundamental objectives of these specifications is to develop interactive services and applications over interoperable radio components and data communication protocols.

This figure shows the architecture of the Bluetooth stack.


Bluetooth devices can be one of these two types:

- Single mode - Supports a BR/EDR or LE profile
- Dual mode - Supports BR/EDR and LE profiles

The subsequent sections provide details about the architecture of "BLE Protocol Stack" on page 57 and "Bluetooth BR/EDR Protocol Stack" on page 5-12.

## BLE Protocol Stack

This figure compares the BLE protocol stack to the Open System Interconnection (OSI) reference model.


In the preceding figure, the BLE protocol stack is shown along with the OSI reference model.

- There is one-to-one mapping at the physical layer (PHY)
- The OSI data link layer (DLL) maps to the BLE logical link control and adaptation protocol (L2CAP) and link layer (LL)
- In the BLE stack, the higher layers provide application layer services, device roles and modes, connection management, and security protocol

The functionality of the BLE protocol stack is divided between three main layers: the Controller, the Host, and Application Profiles and Services.


## Controller

The controller layer includes the BLE PHY, the LL, and the controller-side host controller interface (HCI).

## BLE PHY

The BLE PHY air interface operates in the same unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band as $\mathrm{Wi}-\mathrm{Fi}^{\circledR}$. The BLE PHY air interface also includes these characteristics:

- Operating radio frequency (RF) is in the range 2.4000 GHz to 2.4835 GHz , inclusive.
- The channel bandwidth is 2 MHz . The operating band is divided into 40 channels, $k=0, \ldots, 39$. The center frequency of the $k$ th channel is $2402+k \times 2 \mathrm{MHz}$.
- User data packets are transmitted using channels in the range [0, 36].
- Advertising data packets are transmitted in channels 37, 38, and 39.
- Gaussian frequency shift-keying (GFSK) modulation scheme is implemented.
- The BLE PHY uses frequency-hopping spread spectrum (FHSS) to reduce interference and to counter the impact of fading channels. The time between frequency hops can vary from 7.5 ms to 4 s and is set at the connection time for each slave.
- Support for throughput at 1 Mbps is mandatory for specification version 4.x compliant devices. At a data rate of 1 Mbps , the transmission is uncoded.
- Optionally, devices compliant with the Bluetooth Core Specification version 5.1 support these additional data rates:
- Coded transmission at bit rates of 500 kbps or 125 kbps
- Uncoded transmission at a bit rate of 2 Mbps


## LL

The LL performs tasks similar to the medium access control (MAC) layer of the OSI model. In Bluetooth, the LL interfaces directly with the BLE PHY and manages the link state of the radio to define the role of a device as master, slave, advertiser, or, scanner.

## Controller-Side HCI

The HCI on the controller side handles the interface between the host and the controller. The HCI defines a set of commands and events for transmission and reception of packet data. When receiving packets from the controller, the HCI extracts raw data at the controller to send to the host.

## Host

The host includes the host-side HCI, L2CAP, attribute protocol (ATT), generic attribute profile (GATT), security manager protocol (SMP), and generic access profile (GAP).

## Host-Side HCI

The HCI on the host side handles the interface between the host and the controller. The HCI defines a set of commands and events for transmission and reception of packet data. When transmitting data, the HCI translates raw data into packets to send them from the host to the controller.

## L2CAP

The L2CAP encapsulates data from the BLE higher layers into the standard BLE packet format for transmission or extracts data from the standard BLE LL packet on reception) according to the link configuration specified at the ATT and SMP layers.

## ATT

The ATT transfers attribute data between clients and servers in GATT-based profiles. The ATT defines the roles of the client-server architecture. The roles typically correspond to the master and the slave as defined in the link layer. In general, a device could be a client, a server, or both, irrespective of whether it is a master or a slave. The ATT also performs data organization into attributes as shown in this figure.


Device attributes are represented as:

- The attribute handle is a 16 -bit identifier value assigned by the server to enable a client to reference those attributes.
- The attribute type is a universally unique identifier (UUID) defined by Bluetooth SIG. For example, UUID 0x2A37 represents a heart-rate measurement.
- The attribute value is a variable length field. The UUID associated with and the service class of the service record containing the attribute value, determine the length of the attribute value field.
- Attribute permissions are sets of permission values associated with each attribute. These permissions specify read and write privileges for an attribute, and the security level required for read and write permission.


## GATT

The GATT provides a reference framework for all GATT-based profiles. The GATT encapsulates the ATT and is responsible for coordinating the exchange of profiles in a BLE link. Profiles include information and data such as handle assignment, a UUID, and a set of permissions.

For devices that implement the GATT profile,

- The client is the device that initiates commands and requests toward the server. The client can receive responses, indications, and notifications.
- The server is the device that accepts incoming commands and requests from the client. The server sends responses, indications, and notifications to the client.

The GATT uses client-server architecture. The roles are not fixed and are determined when a device initiates a defined procedure. Roles are released when the procedure ends.

The terminology used in the GATT includes:

- Service - A collection of data and associated behaviors used to accomplish a particular function or feature
- Characteristic - A value used in a service along with appropriate permissions
- Characteristic descriptor - A description of the associated characteristic behavior
- GATT-Client - A GATT-Client initiates commands and requests towards the server and can receive responses, indications, and notifications sent by the server
- GATT-Server - A GATT-Server accepts incoming commands and requests from a client and sends responses, indications, and notifications to the client


## SMP

The SMP applies security algorithms to encrypt and decrypt data packets. This layer defines the initiator and the responder, corresponding to the master and the slave, once the connection is established.

## GAP

The GAP specifies roles, modes, and procedures of a device. It also manages the connection establishment and security. The GAP interfaces directly with the Application Profiles and Services (App) layer.

## APP Layer

The App layer is the direct user interface defining profiles that afford interoperability between various applications. The Bluetooth core specification enables vendors to define proprietary profiles for use cases not defined by SIG profiles.
$\overline{\text { Note }}$ For more information about the BLE protocol stack architecture, see volume 3, Part C, sections 2 and 2.1 of the Bluetooth Core Specification [1].

## Bluetooth BR/EDR Protocol Stack

This figure compares the block diagram of the Bluetooth BR/EDR protocol stack and with the OSI reference model.


The mapping of BR/EDR stack to the OSI reference model is as shown below:

- The "BR/EDR Radio" on page 5-13 and "Baseband and Link Control" on page 5-13 layers of the Bluetooth BR/EDR stack map to the OSI PHY layer.
- The "Link Manager Protocol (LMP)" on page 5-13, "L2CAP" on page 5-13, "Cable Replacement Protocol" on page 5-13 (RFCOMM), and "PPP" on page 5-14 layers of the Bluetooth BR/EDR stack map to the OSI data link layer.
- The user datagram protocol (UDP), transmission control protocol (TCP), and internet protocol (IP) layers of Bluetooth BR/EDR stack map to a combined, network, transport and session layers of the OSI reference model.
- There is one-to-one mapping at the application layer.


## Core Protocols

The Bluetooth core protocols and the Bluetooth radio are required by most of the Bluetooth devices. The core protocols include these layers.

## BR/EDR Radio

The BR/EDR radio is the lowest defined layer of the Bluetooth specification. The BR mode is mandatory, whereas the EDR mode is optional. This layer defines the requirements of the Bluetooth transceiver device operating in the 2.4 GHz ISM frequency band. It implements a $1600 \mathrm{hops} / \mathrm{sec}$ FHSS technique. The radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth channel has a bandwidth of 1 MHz . Each frequency is located at $(2402+k) \mathrm{MHz}$, where $k$ $=0,1, \ldots 78$. The modulation technique for BR and EDR mode is GFSK and differential phase shift keying (DPSK), respectively. The baud rate is 1 Msymbols/s. The Bluetooth BR/EDR radio uses the time division duplex (TDD) topology in which data transmission occurs in one direction at one time. The transmission alternates in two directions, one after the other.

## Baseband and Link Control

The baseband and link control layer enables the PHY RF link between different Bluetooth devices, forming a piconet. The baseband handles the channel processing and timing, and the link control handles the channel access control. This layer provides these two different types of PHY RF links with their corresponding baseband packets:

- Synchronous connection-oriented (SCO) - Supports real-time audio traffic
- Asynchronous connection-oriented (ACL) - Supports data packet transmission


## Link Manager Protocol (LMP)

The LMP layer is primarily responsible for link setup and link configuration between different Bluetooth devices. These processes include establishing security functions such as authentication and encryption by generating, exchanging, and checking link and encryption keys. Furthermore, this layer controls the power modes and duty cycles of the Bluetooth radio device and the connection states of a Bluetooth unit in a piconet.

## L2CAP

The L2CAP adapts higher-layer protocols over the baseband. It shields the higher-layer protocols from the details of the lower-layer protocols. The L2CAP provides connection-oriented and connectionless services to the higher-layer protocols. This includes protocol multiplexing capability, segmentation and reassembly operations, and group abstractions.

## SDP

Discovery services are an important aspect of the Bluetooth framework. The service discovery protocol (SDP) provides a means for applications to query services and the characteristics of services, following which a connection can be established between two or more Bluetooth devices. The SDP is quite different from service discovery in traditional network-based environments. The SDP is built on top of the L2CAP.

## Cable Replacement Protocol

The cable replacement protocol in the Bluetooth BR/EDR stack uses RFCOMM to provide emulation of serial ports over L2CAP. RFCOMM emulates RS-232 control and data signals over the Bluetooth baseband and provides transport capabilities for higher-layer services that use a serial interface as a transport mechanism. RFCOMM also provides multiple simultaneous connections to one device and enables connections to multiple devices.

## Telephony Control Protocols

The telephony control protocol specification, binary (TCS binary), defines the call control signaling to establish data and voice calls between Bluetooth devices. It is built on top of the L2CAP. Moreover, TCS binary defines mobility management procedures for handling Bluetooth devices.

## Adopted Protocols

In addition to the core protocols, the Bluetooth BR/EDR stack includes protocols adopted from other standard bodies. These adopted protocols are defined in specifications issued by other standardmaking organizations and are incorporated into the Bluetooth framework.

## PPP

The point-to-point protocol (PPP) is an Internet Engineering Task Force (IETF) [3] standard protocol for transporting IP datagrams over a point-to-point link. The PPP runs over the RFCOMM to realize point-to-point connections.

## TCP, UDP, and IP

These layers are the IETF-defined foundation protocols of the TCP/IP protocol suite.

- TCP - This protocol provides a reliable virtual connection between devices to realize data communication. The TCP treats the data as a stream of bytes and transmits them without any errors or duplication.
- UDP - This protocol is an alternative to the TCP and provides an unreliable datagram connection between devices. As there is no end-to-end connection in UDP, data is transmitted link-by-link without any guarantee of service.
- IP - This layer is a network layer protocol that enables a datagram service between devices, supporting both the TCP and UDP.

The use of the TCP, UDP, and IP in the Bluetooth BR/EDR stack enables communication with any other device connected to the Internet.

## OBEX

The object exchange (OBEX) protocol is a session-level protocol developed by the Infrared Data Association (IrDA) to exchange objects. The OBEX protocol provides functionality similar to that of HTTP, but in a simpler manner. HTTP is an application layer protocol and layered above the TCP/IP. The OBEX protocol provides the client with a reliable transport for connecting to a server. It also provides a model for representing objects and operations.

## WAE and WAP

Bluetooth BR/EDR stack incorporates the wireless application environment (WAE) and wireless application protocol (WAP) into its architecture. The advantages of using WAE/WAP features in the Bluetooth stack are:

- Build application gateways that act as an interface between WAP servers and some other application on the PC
- Provide functions such as remote control and data fetching from the PC to the Bluetooth handset
- Reuse the upper software application developed for the WAP application environment


## Application Profiles and Services

For more information, refer "APP Layer" on page 5-11.

## Alternate MAC/PHY

The alternate MAC/PHY (AMP) manager is a secondary controller in the Bluetooth core system. After an L2CAP connection is established between two devices over the BR/EDR radio, the AMP manager can discover the AMPs that are available on the other device. If an AMP is common between two devices, the Bluetooth core system provides mechanisms for moving data traffic from the BR/EDR controller to an AMP controller.

Each AMP manager consists of a protocol adaptation layer (PAL) on top of a MAC and PHY. The PAL maps the Bluetooth protocols to the specific protocols of the underlying MAC and PHY.

L2CAP channels can be created on, or moved to, an AMP. If an AMP physical link has a link supervision timeout, then L2CAP channels can be moved back to BR/EDR radio. To minimize power consumption in the device, AMPs are enabled or disabled as required.

## HCl

The HCI provides a command interface to the BR/EDR radio, baseband controller, and the link manager. It is a single standard interface for accessing the Bluetooth baseband capabilities, the hardware status, and the control registers.

Note For more information about the Bluetooth BR/EDR protocol stack architecture, see volume 1, Part A, sections 2 and 2.1 of the Bluetooth Core Specification [1].

## References

[1] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https:// www.bluetooth.com/.
[2] Bluetooth Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed November 6, 2019. https://www.bluetooth.com/.
[3] IETF. "Internet Standards." Accessed November 6, 2019. https://www.ietf.org/.
[4] "Bluetooth Protocol Stack - an Overview | ScienceDirect Topics." Accessed November 15, 2019. https://www.sciencedirect.com/.

## See Also

## More About

- "What Is Bluetooth?" on page 5-2
- "Bluetooth Packet Structure" on page 5-23


## Bluetooth Parameterization

Communications Toolbox Library for the Bluetooth Protocol configuration objects initialize, store, and validate configuration properties.

## Configuration Objects in Communications Toolbox Library for the Bluetooth Protocol

The configuration objects are designed specifically as containers to store properties. They also provide some level of data validation for the function inputs that they maintain. Functions perform further data validation across input settings based on run-time conditions.

The configuration objects are optimized for iterative computations that process large streams of data, such as communications systems.

Communications Toolbox Library for the Bluetooth Protocol configuration objects define and configure format-specific and function-specific properties. The property page of each object contains descriptions, valid settings, ranges, and other information about the object properties.

- bleLLDataChannelPDUConfig - The Link Layer Data Channel Protocol Data Unit configuration object defines and configures LL data channel PDUs.
- bleLLAdvertisingChannelPDUConfig - The Link Layer Advertising Channel Protocol Data Unit configuration object defines and configures LL advertising channel PDUs.
- bleL2CAPFrameConfig - The Layer 2 CAP Frame configuration object defines and configures Layer 2 CAP frame.
- bleGAPDataBlockConfig - The GAP Data Block configuration object defines and configures GAP data block.
- bleATTPDUConfig - The ATT Protocol Data Unit configuration object defines and configures ATT PDUs.


## See Also

## More About

- "Create Configuration Objects" on page 5-17


## Create Configuration Objects

Communications Toolbox Library for the Bluetooth Protocol uses value objects to organize properties required for generation of higher layer Bluetooth PDUs. After you create the various configuration objects described here, you can use them to generate waveforms.

## Create Link Layer Data Channel PDU Configuration Object

This example shows how to create a BLE link layer data channel PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using Name, Value pairs when creating the object.

## Create Object and Then Modify Properties

Create a BLE link layer data channel PDU configuration object with default settings.

```
lldatapdu = bleLLDataChannelPDUConfig
lldatapdu =
    bleLLDataChannelPDUConfig with properties:
    LLID: 'Data (continuation fragment/empty)'
            NESN: 0
        SequenceNumber: 0
            MoreData: 0
```

Modify the defaults to specify a 'Data (start fragment/complete)' PDU.

```
lldatapdu.LLID = 'Data (start fragment/complete)'
lldatapdu =
    bleLLDataChannelPDUConfig with properties:
                            LLID: 'Data (start fragment/complete)'
            NESN: 0
        SequenceNumber: 0
            MoreData: 0
```


## Overide Default Property Values During Object Creation

Create a BLE link layer data channel PDU configuration object, using Name, Value pairs to specify a 'Control' PDU.

```
lldatapdu2 = bleLLDataChannelPDUConfig('LLID','Control')
lldatapdu2 =
    bleLLDataChannelPDUConfig with properties:
            LLID: 'Control'
                    NESN: 0
        SequenceNumber: 0
            MoreData: 0
        ControlConfig: [1x1 bleLLControlPDUConfig]
```


## Create Link Layer Advertising Channel PDU Configuration Object

This example shows how to create a BLE link layer advertising channel PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using Name, Value pairs when creating the object.

## Create Object and Then Modify Properties

Create a BLE link layer advertising channel PDU configuration object with default settings.

```
lladvertpdu = bleLLAdvertisingChannelPDUConfig
lladvertpdu =
    bleLLAdvertisingChannelPDUConfig with properties:
                            PDUType: 'Advertising indication'
        ChannelSelection: 'Algorithm1'
        AdvertiserAddressType: 'Random'
    AdvertiserAddress: '0123456789AB'
            AdvertisingData: [3x2 char]
```

Modify the defaults to specify a 'Advertising direct indication' PDU.

```
lladvertpdu.PDUType = 'Advertising direct indication'
```

```
lladvertpdu =
    bleLLAdvertisingChannelPDUConfig with properties:
```

                            PDUType: 'Advertising direct indication'
            ChannelSelection: 'Algorithml'
        AdvertiserAddressType: 'Random'
            AdvertiserAddress: '0123456789AB'
            TargetAddressType: 'Random'
                TargetAddress: '0123456789CD'
    
## Overide Default Property Values During Object Creation

Create a BLE link layer advertising channel PDU configuration object, using Name, Value pairs to specify a 'Scan response' PDU using channel section algorithm 2.

```
lldatapdu2 = bleLLAdvertisingChannelPDUConfig('PDUType','Scan response','ChannelSelection','Algo
lldatapdu2 =
    bleLLAdvertisingChannelPDUConfig with properties:
                            PDUType: 'Scan response'
            ChannelSelection: 'Algorithm2'
        AdvertiserAddressType: 'Random'
            AdvertiserAddress: '0123456789AB'
            ScanResponseData: [3x2 char]
```


## Create L2CAP Frame Configuration Object

This example shows how to create a BLE logical link control and adaptation protocol (L2CAP) frame configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using Name, Value pairs when creating the object.

## Create Object and Then Modify Properties

Create a BLE L2CAP frame configuration object with default settings.

```
l2capframe = bleL2CAPFrameConfig
l2capframe =
    bleL2CAPFrameConfig with properties:
            ChannelIdentifier: '0005'
                        CommandType: 'Credit based connection request'
            SignalIdentifier: '01'
        SourceChannelIdentifier: '0040'
                            LEPSM: '001F'
            MaxTransmissionUnit: 23
                MaxPDUPayloadSize: 23
                    Credits: 1
```

Modify the defaults setting the channel identifier to ' 0004 ' to specify an ATT channel.

```
l2capframe.ChannelIdentifier = '0004'
l2capframe =
    bleL2CAPFrameConfig with properties:
```

        ChannelIdentifier: '0004'
    
## Overide Default Property Values During Object Creation

Create a BLE L2CAP frame configuration object, using Name, Value pairs to specify a 'Command reject' signaling channel command with the reject reason 'Invalid CID in request'.

```
l2capframe = bleL2CAPFrameConfig ('CommandType','Command reject','CommandRejectReason','Invalid
l2capframe =
    bleL2CAPFrameConfig with properties:
                            ChannelIdentifier: '0005'
                            CommandType: 'Command reject'
                            SignalIdentifier: '01'
            CommandRejectReason: 'Invalid CID in request'
            SourceChannelIdentifier: '0040'
        DestinationChannelIdentifier: '0040'
```


## Create GAP Data Block Configuration Object

This example shows how to create a BLE GAP data block configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using Name, Value pairs when creating the object.

## Create Object and Then Modify Properties

Create a BLE GAP data block configuration object with default settings.

```
gapDataBlk = bleGAPDataBlockConfig
gapDataBlk =
    bleGAPDataBlockConfig with properties:
    AdvertisingDataTypes: {'Flags'}
        LEDiscoverability: 'General'
            BREDR: 0
```

Modify the defaults to specify for an advertising data block for 'Flags' and 'Tx power level' advertising data types.

```
gapDataBlk.AdvertisingDataTypes = {'Flags'; 'Tx power level'}
gapDataBlk =
    bleGAPDataBlockConfig with properties:
        AdvertisingDataTypes: {2x1 cell}
            LEDiscoverability: 'General'
                    BREDR: 0
                TxPowerLevel: 0
gapDataBlk.AdvertisingDataTypes
ans = 2xl cell
    {'Flags' }
    {'Tx power level'}
```


## Overide Default Property Values During Object Creation

Create a BLE GAP data block configuration object, using Name, Value pairs to specify 'Advertising interval' and 'Local name' advertising data types.

```
gapDataBlk = bleGAPDataBlockConfig ('AdvertisingDataTypes', {'Advertising interval', 'Local name
```

gapDataBlk =
bleGAPDataBlockConfig with properties:
AdvertisingDataTypes: \{2x1 cell\}
LocalName: 'Bluetooth'
LocalNameShortening: 0
AdvertisingInterval: 32
gapDataBlk.AdvertisingDataTypes

```
ans = 2x1 cell
    {'Advertising interval'}
    {'Local name' }
```


## Create Attribute PDU Configuration Object

This example shows how to create a BLE attribute (ATT) PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using Name, Value pairs when creating the object.

## Create Object and Then Modify Properties

Create a BLE ATT PDU configuration object with default settings.

```
attpdu = bleATTPDUConfig
attpdu =
    bleATTPDUConfig with properties:
                            Opcode: 'Read request'
        AttributeHandle: '0001'
```

Modify the defaults to specify a 'Read blob request' operation code.

```
attpdu.Opcode = 'Read blob request'
attpdu =
    bleATTPDUConfig with properties:
            Opcode: 'Read blob request'
        AttributeHandle: '0001'
            Offset: 0
```


## Overide Default Property Values During Object Creation

Create a BLE ATT PDU configuration object, using Name, Value pairs to specify ' Information request ' for the operation code of a request PDU that caused an error.

```
lldatapdu2 = bleATTPDUConfig('RequestedOpcode','Information request')
lldatapdu2 =
    bleATTPDUConfig with properties:
                            Opcode: 'Read request'
        AttributeHandle: '0001'
```


## See Also

## More About

- "Bluetooth Parameterization" on page 5-16


## Bluetooth Packet Structure

The Bluetooth Special Interest Group (SIG) [1] and [2] defines different packet structures for Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR) devices.

## BLE Packet Structure

## Bit Ordering in BLE Packets

When defining packets and messages in the baseband specification, the bit ordering follows the littleendian format. In this format, these rules apply:

- The least significant bit (LSB) corresponds to $b_{0}$.
- LSB is the first bit sent over the air.
- When illustrating the packet structure, the LSB is shown on the left side.

Moreover, data fields generated internally at the baseband level (packet header and payload header length), must be transmitted with the LSB first. For example, a 3-bit parameter is sent as: $\mathrm{b}_{0} \mathrm{~b}_{1} \mathrm{~b}_{2}=$ 110 over the air, where 1 is sent first and 0 is sent last.

The BLE devices use packet formats for: "BLE Uncoded Physical Layer (PHY)" on page 5-23, "BLE Coded PHY" on page 5-24, "Advertising Physical Channel PDU" on page 5-26, "Data Physical Channel PDU" on page 5-27, and "Constant Tone Extension and In-Phase Quadrature (IQ) Sampling" on page 5-30.

Note For more information about BLE packet structure, see volume 6, Part B, Section 2 of the Bluetooth Core Specification [2].

## BLE Uncoded Physical Layer (PHY)

The Bluetooth Core Specification [2] defines two physical layer (PHY) transmission modes (LE 1M and LE 2M) for uncoded PHY. This figure shows the packet structure for the BLE uncoded PHY operating on LE 1 M and LE 2 M .


Each packet contains four mandatory fields (preamble, access-address, protocol data unit (PDU), and cyclic redundancy check (CRC)) and one optional field (constant tone extension (CTE)). The preamble is transmitted first, followed by the access address, PDU, CRC, and CTE (if present) in that order. The entire packet is transmitted at the same symbol rate of $1 \mathrm{Msym} / \mathrm{s}$ or $2 \mathrm{Msym} / \mathrm{s}$.

## Preamble

All link layer (LL) packets contain a preamble, which is used in the receiver to perform frequency synchronization, automatic gain control (AGC) training, and symbol timing estimation. The preamble
is a fixed sequence of alternating 0 and 1 bits. For the BLE packets transmitted on the LE 1M PHY and LE 2M PHY, the preamble size is 1 octet and 2 octets, respectively.

## Access address

The access address is a 4-octet value. Each LL connection between any two devices and each periodic advertising train has a distinct access address. Each time the BLE device needs a new access address, the LL generates a new random value adhering to these requirements:

- The value must not be the access address for any existing LL connection on this device.
- The value must not be the access address for any enabled periodic advertising train.
- The value must have no more than six successive 1 s or 0 s .
- The value must not be the access address for any advertising channel packets.
- The value must not be a sequence that differs from the access address of advertising physical channel packets by only 1 bit.
- All four octets for the value must not be equal.
- The value must have a minimum of two transitions in the most significant 6 bits.

If the random value does not satisfy the above requirements, a new random value is generated until it meets all of the requirements.

## PDU

When a BLE packet is transmitted on either the primary or secondary advertising physical channel or the periodic physical channel, the PDU is defined as the "Advertising Physical Channel PDU" on page $5-26$. When a packet is transmitted on the data physical channel, the PDU is defined as the "Data Physical Channel PDU" on page 5-27.

## CRC

The size of the CRC is 3 octets and is calculated on the PDU of all LL packets. If the PDU is encrypted, then the CRC is calculated after encryption of the PDU is complete. The CRC polynomial has the form $x^{24}+x^{10}+x^{9}+x^{6}+x^{4}+x^{3}+x+1$.

For more information about CRC generation, see volume 6, Part B, Section 3.1.1 of the Bluetooth Core Specification [2].

## CTE

The CTE consists of a constantly modulated series of unwhitened 1s. This field has a variable length that ranges from $16 \mu$ s to $160 \mu \mathrm{~s}$.

For more information about the CTE, see volume 6, Part B, Section 2.5.1 of the Bluetooth Core Specification [2].

Note For more information about BLE uncoded PHY packet structure, refer to Vol 6, Part B, Section 2.1 of Bluetooth Core Specification [2].

## BLE Coded PHY

This figure shows the packet structure for the BLE coded PHY and is implemented for BLE packets on all the physical channels.


Each BLE packet consists of a preamble and these two forward error correcting (FEC) blocks:

- FEC block 1- This block contains three fields: access address, coding indicator (CI), and TERM1. This block implements an $S=8$ coding scheme, where each bit represents eight symbols. This gives a data rate of 125 Kbps .
- FEC block 2- This block contains these three fields: PDU, CRC, and TERM2. This block implements an $S=8$ or $S=2$ coding scheme. In the $S=2$ coding scheme, each bit represents two symbols. Therefore, the data rate is 500 Kbps .

The BLE coded PHY does not contain the CTE.

## Preamble

The BLE coded PHY preamble is 80 symbols in length and contains 10 repetitions of the symbol pattern '00111100' (in the transmission order).

## Access address

The length of BLE coded PHY access address is 256 symbols. For more information, see "Access address" on page 5-24. In addition to the requirements listed in the access address subsection of the "BLE Uncoded Physical Layer (PHY)" on page 5-23 section, the new value for the access address of the BLE coded PHY must also meet these requirements:

- The value must have at least three 1 s in the last significant bits.
- The value must have no more than 11 transitions in the least significant 16 bits.


## CI

The CI field consists of two bits as shown in this table:

| Bits in CI | Description |
| :--- | :--- |
| 00b | FEC block 2 coded using $S=8$ |
| 01b | FEC block 2 coded using $S=2$ |
| All other values | Reserved for future use |

## PDU

The PDU in the BLE coded PHY packet structure has the same formatting as the "PDU" on page 5-24 in the BLE uncoded PHY packet.

## CRC

The CRC in the BLE coded PHY packet structure has the same formatting as the "CRC" on page 5-24 in the BLE uncoded PHY packet.

## TERM1 and TERM2

Each FEC block contains a terminator at the end of the block. That terminator is referred to as TERM1 and TERM2. Each terminator is 3 bits long and forms the termination sequence during the FEC encoding process.
$\overline{\text { Note For more information about BLE coded PHY packet structure, see volume 6, Part B, Section } 2.2}$ of the Bluetooth Core Specification [2].

## Advertising Physical Channel PDU

The packet structure format of the advertising physical channel PDU is shown in this figure.


The advertising physical channel PDU has a 16 -bit header and a variable-size payload. The 16 -bit header field of the advertising physical channel PDU is shown in this figure.


The PDU type field in the advertising channel PDU header defines different types of PDUs that can be transmitted on the BLE coded PHY. This table maps different types of PDUs with the physical channels and the PHYs on which the BLE packet might appear. The table also indicates the PHY transmission modes supported for each type of advertising physical channel PDU.

| PDU Type | PDU Name | Physical Channel | LE 1M <br> Suppor <br> t | LE 2M <br> Support | LE Coded <br> Support |
| :--- | :--- | :--- | :--- | :--- | :--- |
| 0000b | ADV_IND | Primary Advertising | Yes |  |  |
| 0001b | ADV_DIRECT_IND | Primary Advertising | Yes |  |  |
| 0010b | ADV_NONCONN_IND | Primary Advertising | Yes |  |  |
| 0011b | SCAN_REQ | Primary Advertising | Yes |  |  |
|  | AUX_SCAN_REQ | Secondary Advertising | Yes | Yes | Yes |
| 0100b | SCAN_RSP | Primary Advertising | Yes |  |  |
| 0101b | CONNECT_IND | Primary Advertising | Yes |  |  |
|  | AUX_CONNECT_REQ | Secondary Advertising | Yes | Yes | Yes |


| PDU Type | PDU Name | Physical Channel | LE 1M <br> Suppor <br> t | LE 2M <br> Support | LE Coded <br> Support |
| :--- | :--- | :--- | :--- | :--- | :--- |
| 0110b | ADV_SCAN_IND | Primary Advertising | Yes |  |  |
|  | ADV_EXT_IND | Primary Advertising | Yes |  | Yes |
|  | AUX_ADV_IND | Secondary Advertising | Yes | Yes | Yes |
|  | AUX_SCAN_RSP | Secondary Advertising | Yes | Yes | Yes |
|  | AUX_SYNC_IND | Periodic | Yes | Yes | Yes |
|  | AUX_CHAIN_IND | Secondary Advertising <br> and Periodic | Yes | Yes | Yes |
| 1000b | AUX_CONNECT_RSP | Secondary Advertising | Yes | Yes | Yes |
| All other values | Reserved for future use |  |  |  |  |

The RFU field is reserved for future use. The ChSel, TxAdd, and RxAdd fields of the advertising physical channel PDU header contain information specific to the type of PDU defined for each advertising physical channel PDU separately. If the ChSel, TxAdd, or RxAdd fields are not defined as used in a given PDU, then they are considered as reserved for future use.

The Length field of the advertising physical channel PDU header denotes the length of the payload in octets. The valid range of the Length field is 1 to 255 octets.

The Payload field in the advertising physical channel PDU packet structure is specific to the type of PDUs listed in the preceding table.

Note For more information about advertising physical channel PDUs, see volume 6, Part B, Section 2.3 of the Bluetooth Core Specification [2].

## Data Physical Channel PDU

The packet structure format of the data physical channel PDU is shown in this figure.


The data physical channel PDU has a 16 -bit or 24 -bit header, a variable length payload in the range [ 0,251 ] octets, and can include a 32-bit message integrity check (MIC) field. The MIC is not included in an unencrypted LL connection or in an encrypted LL connection with a data channel PDU containing an empty payload. The MIC is included in an encrypted LL connection with a data channel PDU containing a nonzero length payload. In this case, the MIC is calculated as specified in volume 6, Part E, Section 1 of the Bluetooth Core Specification [2].

The header field of the the data physical channel PDU is shown in this figure.


The data physical channel PDU header includes these fields:

- Link layer identifier (LLID) - This field indicates whether the packet is an LL data PDU or LL control PDU.
- 00 b - Reserved for future use
- 01b - LL Data PDU, which can be a continuation fragment of an logical link control and adaptation (L2CAP) message or an empty PDU
- 10b - LL Data PDU, which can be a start of an L2CAP message or a complete L2CAP message with no fragmentation
- 11b - LL control PDU
- Next expected sequence number (NESN): The LL uses this field to either acknowledge the last data physical channel PDU sent by the peer or to request the peer to resend the last data physical channel PDU. For more information about NESN, see volume 6, Part B, Section 4.5.9 of the Bluetooth Core Specification [2].
- Sequence number (SN): The LL uses this field to identify the BLE packets sent by it. For more information about the SN, see volume 6, Part B, Section 4.5.9 of the Bluetooth Core Specification [2].
- More data (MD): This field indicates that the BLE device has more data to send. If neither of master and slave BLE device has set the MD bit in their packets, the packet from the slave closes the connection event. If the master and slave devices have set the MD bit, the master can continue the connection event by sending another packet, and the slave must listen after sending its packet. For more information about MD, see volume 6, Part B, Section 4.5.6 of the Bluetooth Core Specification [2].
- CTEInfo present (CP): This field indicates whether the data physical channel PDU header has a CTEInfo field and, subsequently whether the data physical channel packet has a CTE. For more information about the packet structure of the CTEInfo field, see volume 6, Part B, Section 2.5.2 of the Bluetooth Core Specification [2].
- Length: This field indicates the size, in octets, of the payload and MIC, if present. The size of this field is in the range [ 0,255 ] octets.
- CTEInfo: This field indicates the type and length of the CTE.

The two types of data physical channel PDUs are: "LL Data PDU" on page 5-28 and "LL Control PDU" on page 5-29.

## LL Data PDU

The LL uses the LL data PDU to send L2CAP data. The LLID field in the LL data channel PDU header is set to either 01b or 10b. An LL data PDU is referred to as an empty PDU if

- The LLID field of the LL data channel PDU header is set to 01b.
- The Length field of the LL data channel PDU header is set to 00000000b.

An LL data PDU with the LLID field in the header set to 10b does not have the Length field set to 00000000b.

## LL Control PDU

The LL uses the LL data PDU to control the LL connection. If the LLID field of data physical channel PDU header is set to 11b, the data physical channel PDU contains an LL control PDU. This figure shows the LL control PDU payload.


The Opcode field defines different types of LL control PDUs as shown in this table.

| Opcode | LL Control PDU |
| :--- | :--- |
| $0 \times 00$ | LL_CONNECTION_UPDATE_IND |
| $0 \times 01$ | LL_CHANNEL_MAP_IND |
| $0 \times 02$ | LL_TERMINATE_IND |
| $0 \times 03$ | LL_ENC_REQ |
| $0 \times 04$ | LL_ENC_RSP |
| $0 \times 05$ | LL_START_ENC_REQ |
| $0 \times 06$ | LL_START_ENC_RSP |
| $0 \times 07$ | LL_UNKNOWN_RSP |
| $0 \times 08$ | LL_FEATURE_REQ |
| $0 \times 09$ | LL_FEATURE_RSP |
| $0 \times 0 A$ | LL_PAUSE_ENC_REQ |
| $0 \times 0 B$ | LL_PAUSE_ENC_RSP |
| $0 \times 0 C$ | LL_VERSION_IND |
| $0 \times 0 D$ | LL_REJECT_IND |
| $0 \times 0 E$ | LL_SLAVE_FEATURE_REQ |
| $0 \times 0 F$ | LL_CONNECTION_PARAM_REQ |
| $0 \times 10$ | LL_CONNECTION_PARAM_RSP |
| $0 \times 11$ | LL_REJECT_EXT_IND |
| $0 \times 12$ | LL_PING_REQ |
| $0 \times 13$ | LL_PING_RSP |
| $0 \times 14$ | LL_LENGTH_REQ |
| $0 \times 15$ | LL_LENGTH_RSP |
| $0 \times 16$ | LL_PHY_REQ |
| $0 \times 17$ | LL_PHY_RSP |


| Opcode | LL Control PDU |
| :--- | :--- |
| $0 \times 18$ | LL_PHY_UPDATE_IND |
| $0 \times 19$ | LL_MIN_USED_CHANNELS_IND |
| $0 \times 1$ A | LL_CTE_REQ |
| $0 \times 1 B$ | LL_CTE_RSP |
| $0 \times 1 C$ | LL_PERIODIC_SYNC_IND |
| $0 \times 1 D$ | LL_CLOCK_ACCURACY_REQ |
| $0 \times 1 E$ | LL_CLOCK_ACCURACY_RSP |
| All other values | Reserved for future use |

The CtrData field in the LL control PDU is specific to the value of the Opcode field. For more information about different LL control PDUs and their corresponding CtrData field structure, see volume 6, Part B, Sections 2.4.2.1 to 2.4.2.28 of the Bluetooth Core Specification [2].

## Constant Tone Extension and In-Phase Quadrature (IQ) Sampling

The length of the CTE is variable and in the range [16, 160] $\mu \mathrm{s}$. This field contains a constantly modulated series of 1 s with no whitening applied. The CTE is of two types: antenna switching during CTE transmission (AoD) and antenna switching during CTE reception (AoA). When receiving a packet containing an AoD CTE, the receiver does not need to switch antennae. When receiving a packet containing an AoA CTE, the receiver performs antenna switching at the rate and according to the switching pattern configured by the host. In both cases, the receiver takes an IQ sample at each microsecond during the reference period and an IQ sample each sample slot. The controller reports the IQ samples to the host. The receiver samples the entire CTE regardless of its length, unless this conflicts with other activities. For more information about CTE, see volume 6, Part B, Sections 2.5.1 to 2.5.3 of the Bluetooth Core Specification [2].

When requested by the host, the receiver performs IQ sampling when receiving a valid BLE packet with a CTE. However, when receiving a BLE packet with a CTE and an incorrect CRC, the receiver might perform IQ sampling. For more information about IQ sampling, see volume 6, Part B, Section 2.5.4 of the Bluetooth Core Specification [2].

Note For more information about data physical channel PDUs, see volume 6, Part B, Section 2.4 of the Bluetooth Core Specification [2].

## Bluetooth BR/EDR Packet Structure

## Bit Ordering in Bluetooth BR/EDR Packets

The bit ordering in Bluetooth BR/EDR packets follows the same format as the "Bit Ordering in BLE Packets" on page 5-23.

Bluetooth BR/EDR devices use packet formats for: "BR Mode" on page 5-31, "EDR Mode" on page 531, "Access Code" on page 5-31, "Packet Header" on page 5-33, "Packet Types" on page 5-34, and "Payload Format" on page 5-35.

Note For more information about Bluetooth BR/EDR packet structure, see volume 2, Part B, Section 6 of the Bluetooth Core Specification [2].

## General Format

## BR Mode

The general format of Bluetooth BR packets is shown in this figure. Each packet consists of these fields: the access code ( 68 or 72 bits), header ( 54 bits), and payload in the range [ 0,2790 ] bits.

| Access Code | Header | BR Payload |
| :---: | :---: | :---: |
| 68 or 72 <br> bits | 54 bits | $0-2790$ bits |

The Bluetooth Core Specification [2] defines different types of packets. A packet can consist of:

- The shortened access code only
- The access code and the packet header
- The access code, the packet header, and the payload

EDR Mode
The general format of Bluetooth EDR packets is shown in this figure.


The format and modulation of the access code and the packet header fields are similar to that of BR packets. Following the header field, the EDR packets have a guard time in the range [4.75, 5.25] $\mu \mathrm{s}$, a sync sequence ( $11 \mu \mathrm{~s}$ ), payload in the range [ 0,2790 ] bits, and trailer (two symbols) fields.

## Access Code

Each packet starts with an access code. If a packet header follows, the access code is 72 bits long. Otherwise, the length of the access code is 68 bits. In this case, the access code is referred to as a shortened access code. The shortened access code does not contain a trailer. The access code is used for synchronization, DC offset compensation, and identification of all packets exchanged on the physical channel. The shortened access code is used in paging and inquiry. In this case, the access code itself is used as a signaling message, and neither a header nor a payload is present. This figure shows the packet structure of the access code.


Different access code types use different lower address parts (LAPs) to construct the sync word. A summary of different access code types is shown in this table.

| Access Code <br> Type | LAP | Access Code <br> Length (Bits) | Description |
| :--- | :--- | :--- | :--- |
| Channel access <br> code (CAC) | Master | 72 | This access code is used in the connection <br> state, synchronization train substate, and <br> synchronization scan substate. It is <br> derived from the LAP of the Master's <br> BD_ADDR. |
| Device access <br> code (DAC) | Paged device | 68 or 72 | This access code is used during page, <br> page scan, and page response substates. It <br> is derived from the paged devices's <br> BD_ADDR. |
| Dedicated inquiry <br> access code <br> (DIAC) | Dedicated | 68 or 72 | This access code is used in the inquiry <br> substate for dedicated inquiry operations. |
| General inquiry <br> access code <br> (GIAC) | Reserved | 68 or 72 | This access code is used in the inquiry <br> substate for general inquiry operations. |

For DAC, DIAC, and GIAC access code types, the access code length of 72 bits is used only in combination with frequency hopping sequence (FHS) packets. When used as self-contained messages without a header, the DAC, DIAC and GIAC do not include trailer bits and are of length 68 bits.

The CAC consists of a preamble, sync word, and trailer.

- Preamble: It is a fixed 4 -symbol pattern of 1 s and 0 s that facilitates DC compensation. If the LSB of the following sync word is 1 or 0 , the preamble sequence is 1010 or 0101 (in transmission order), respectively.
- Sync word: It is a 64 -bit code word derived from a 24 -bit LAP address. The construction guarantees a large Hamming distance between sync words based on different LAPs. The autocorrelation properties of the sync word improve timing acquisition.
- Trailer: It is appended to the sync word as soon as the packet header follows the access code. The trailer is a fixed 4 -symbol pattern of 1 s and 0 s . The trailer together with the three MSBs of the sync word form a 7 -bit pattern of alternating 1 s and 0 s which is used for extended DC compensation. The trailer sequence is either 1010 or 0101 (in transmission order) depending on whether the MSB of the sync word is 0 or 1 , respectively.

Note For more information about access code in Bluetooth BR/EDR, see volume 2, Part B, Section 6.3 of the Bluetooth Core Specification [2].

## Packet Header

The structure of the Bluetooth BR/EDR packet header is shown in this figure.


This table provides a brief description about the packet header fields.

| Packet Header Field | Size of the Field <br> (Bits) | Description |
| :--- | :--- | :--- |
| Logical transport <br> address (LT_ADDR) | 3 | This field indicates the destination slave(s) for a <br> packet in a master-to-slave transmission slot and <br> indicates the source slave for a slave-to-master <br> transmission slot. |
| Type | 4 | This field specifies the type of packet used. The <br> Bluetooth Core Specification [2] defines 16 different <br> types of BR/EDR packets. The value in this field <br> depends on the value of LT_ADDR field in the packet. <br> This field determines the number of slots occupied <br> by the current packet. |
| Flow control (FLOW) | 1 | This field implements the flow control of BR/EDR <br> packets over the asynchronous connection-oriented <br> logical (ACL) transport. When the receive buffer for |
| the ACL logical transport is full, a 'STOP' indication |  |  |
| (FLOW = 0) is returned to stop the other device |  |  |
| from transmitting data temporarily. When the |  |  |
| receive buffer can accept data, a 'GO' indication |  |  |
| (FLOW = 1) is returned. |  |  |$|$| This field informs the source of a successful transfer |
| :--- |
| of payload data with the CRC. This field is reserved |
| for future use on the connectionless slave broadcast |
| (CSB) logical transport. |


| Packet Header Field | Size of the Field <br> (Bits) | Description |
| :--- | :--- | :--- |
| Header error check <br> (HEC) | 8 | This field checks the packet header integrity. Before <br> generating the HEC, the HEC generator is initialized <br> with an 8-bit value. These 8 bits correspond to the <br> upper address part (UAP). After the initialization, <br> the HEC generator calculates the HEC value for the <br> 10 header bits. Before checking the HEC, the <br> receiver initializes the HEC check circuitry with the <br> appropriate 8-bit UAP. If the HEC does not check the <br> packet header integrity, the entire packet is <br> discarded. |

Note For more information about packet header used in Bluetooth BR/EDR, see volume 2, Part B, Section 6.4 of the Bluetooth Core Specification [2].

## Packet Types

The packets used in the piconet are related to these logical transports on which they are used.

- Synchronous connection-oriented (SCO): It is a circuit-switched connection that reserved slots between the master and a specific slave.
- Extended SCO (eSCO): Similar to SCO, it reserves slots between the master and a specific slave. eSCO supports a retransmission window following the reserved slots. Together, the reserved slots and the retransmission window form the complete eSCO window.
- ACL: It provides a packet-switched connection between the master and all active slaves participating in the piconet. ACL supports asynchronous and isochronous services. Between a master and a slave, only a single ACL logical transport must exist.
- CSB: It is used to transport profile broadcast data from a master to multiple slaves. A CSB logical transport is unreliable.

This table summarizes the packets defined for the SCO, eSCO, ACL, and CSB logical transport types.

Note The column entries followed by "D" means data field only. "C.1" implies that the MIC value is mandatory when encryption with AES-CCM is enabled. Otherwise, MIC is excluded. For more information about different packet types used in Bluetooth BR/EDR, see volume 2, Part B, Sections 6.5 and 6.7 of the Bluetooth Core Specification [2].

| Packet <br> Type | TYPE <br> Code | Slot <br> Occupancy | Payload <br> Header <br> (Bytes) | User <br> Payload <br> (Bytes) | FEC | MIC | CRC | Logical <br> Transport <br> Types <br> Supported |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| ID | N/A | 1 | N/A | N/A | N/A | N/A | N/A | N/A |
| NULL | 0000 | 1 | N/A | N/A | N/A | N/A | N/A | SCO, eSCO, <br> ACL, CSB |


| Packet <br> Type | TYPE <br> Code | Slot <br> Occupancy | Payload <br> (eader <br> (Bytes) | User <br> Payload <br> (Bytes) | FEC | MIC | CRC | Logical <br> Transport <br> Types <br> Supported |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| POLL | 0001 | 1 | N/A | N/A | N/A | N/A | N/A | SCO, eSCO, <br> ACL |
| FHS | 0010 | 1 | N/A | 18 | $2 / 3$ | N/A | Yes | SCO, ACL |
| DM1 | 0011 | 1 | 1 | $0-17$ | $2 / 3$ | C.1 | Yes | SCO, ACL, <br> CSB |
| DH1 | 0100 | 1 | 1 | $0-27$ | No | C.1 | Yes | ACL, CSB |
| DM3 | 1010 | 3 | 2 | $0-121$ | $2 / 3$ | C.1 | Yes | ACL, CSB |
| DH3 | 1011 | 3 | 2 | $0-183$ | No | C.1 | Yes | ACL, CSB |
| DM5 | 1110 | 5 | 2 | $0-224$ | $2 / 3$ | C.1 | Yes | ACL, CSB |
| DH5 | 1111 | 5 | 2 | $0-339$ | No | C.1 | Yes | ACL, CSB |
| 2-DH1 | 0100 | 1 | 2 | $0-54$ | No | C.1 | Yes | ACL, CSB |
| 2-DH3 | 1010 | 3 | 2 | $0-367$ | No | C.1 | Yes | ACL, CSB |
| 2-DH5 | 1110 | 5 | 2 | $0-679$ | No | C.1 | Yes | ACL, CSB |
| 3-DH1 | 1000 | 1 | 2 | $0-83$ | No | C.1 | Yes | ACL, CSB |
| 3-DH3 | 1011 | 3 | 2 | $0-552$ | No | C.1 | Yes | ACL, CSB |
| 3-DH5 | 1111 | 5 | 2 | $0-1021$ | No | C.1 | Yes | ACL, CSB |
| HV1 | 0101 | 1 | N/A | 10 | $1 / 3$ | No | No | SCO |
| HV2 | 0110 | 1 | N/A | 20 | $2 / 3$ | No | No | SCO |
| HV3 | 0111 | 1 | N/A | 30 | No | No | No | SCO |
| DV | 1000 | 1 | 1 D | $10+(0-9)$ D | $2 / 3$ D | No | Yes D | SCO |
| EV3 | 0111 | 1 | N/A | $1-30$ | No | No | Yes | eSCO |
| EV4 | 1100 | 3 | N/A | $1-120$ | $2 / 3$ | No | Yes | eSCO |
| EV5 | 1101 | 3 | N/A | $1-180$ | No | No | Yes | eSCO |
| 2-EV3 | 0110 | 1 | N/A | $1-60$ | No | No | Yes | eSCO |
| 2-EV5 | 1100 | 3 | $1-360$ | No | No | Yes | eSCO |  |
| 3-EV3 | 0111 | 1 | $1-540$ | No | No | Yes | eSCO |  |
| 3-EV5 | 1101 | 3 | No | No | Yes | eSCO |  |  |

## Payload Format

The Buetooth Core Specification [2] defines two types of payload field formats: synchronous data field (for ACL packets) and asynchronous data field (for SCO and eSCO packets). However, the DV packets contain both the synchronous and asynchronous data fields.

- Synchronous data field: In SCO, which supports only the BR mode, the length of the synchronous data field is fixed. The synchronous data field contains only the synchronous data body portion and does not have a payload header. In BR eSCO, the synchronous data field consists of these two segments: a synchronous data body and a CRC code. In this case, no payload header is present. In

EDR eSCO, the synchronous data field consists of a guard time, synchronization sequence, synchronous data body, CRC code, and trailer. In this case, no payload header is present.

- Asynchronous data field: The BR ACL packets have an asynchronous data field consisting of payload header, payload body, MIC (if applicable), and CRC (if applicable). This figure shows the 8bit payload header format for BR single-slot ACL packets.


EDR ACL packets have an asynchronous data field consisting of guard time, synchronization sequence, payload header, payload body, MIC (if applicable), CRC (if applicable), and trailer. This figure shows the 16-bit payload header format for EDR multislot ACL packets.


Note For more information about the payload format, see volume 2, Part B, Sections 6.6.1 and 6.6.2 of the Bluetooth Core Specification [2].

## References

[1] Bluetooth Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed November 22, 2019. https://www.bluetooth.com/.
[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https:// www.bluetooth.com/.

## See Also

## More About

- "What Is Bluetooth?" on page 5-2
- "Bluetooth Protocol Stack" on page 5-7


## Equalization

- "Equalization" on page 6-2
- "Adaptive Equalizers" on page 6-5
- "MLSE Equalizers" on page 6-35


## Equalization

## In this section...

"Equalizer Structure Options" on page 6-2
"Selected References for Equalizers" on page 6-3
In a multipath fading scattering environment, the receiver typically detects several constantly changing, delayed versions of the transmitted signal. These time-dispersive channels cause intersymbol interference (ISI) that occurs when symbols received from multiple paths are delayed and overlap in time. ISI causes high error rates because the symbols from multiple received paths interfere with each other and become indistinguishable by the receiver.

Equalizers attempt to mitigate ISI and improve the receiver performance. Equalizer structures are filters that attempt to match the propagation channel response. For time-varying propagation channels, adapting the equalization filter tap weights so that they maintain a match to the channel over time improves the error rate performance.

## Equalizer Structure Options

The Communications Toolbox includes System objects and blocks to recover transmitted data using by linear, decision-feedback, or maximum-likelihood sequence estimation (MLSE) equalization structures. For more information, see "Selected References for Equalizers" on page 6-3.

This figure shows the high-level configuration options for each equalization structure.


For each equalizer structure, you can configure structural settings (such as the number of taps and initial set of tap weights), algorithmic settings (such as the step size), and the signal constellation used by the modulator in your design. You also specify adaptability of the equalizer tap weights throughout the simulation.

- Linear and decision-feedback filter equalizer structures adapt tap weights by using the LMS, RLS, or CMA adaptive algorithm. When using these equalizer structures, the number of samples per symbol determines whether symbols are processed using whole or fractional symbol spacing.
- When using LMS and RLS adaptive algorithms, the equalizer begins operating in tap weights training mode. Configure the equalizer to operate adaptively in decision-directed mode or without further adjustment of taps after training is completed.
- When using the CMA adaptive algorithm, the equalizer has no training mode. You can configure the equalizer to operate adaptively in decision-directed mode or in nonadaptive mode.

To explore the linear and decision-feedback filter equalizer capabilities, see "Adaptive Equalizers" on page 6-5.

- Maximum-Likelihood Sequence Estimation (MLSE) equalizers use the Viterbi algorithm. The MLSE equalization structure provides the optimal match to the received symbols but it requires an accurate channel estimate and is the most computationally complex structure. To explore MLSE equalizer capabilities, see "MLSE Equalizers" on page 6-35.

The computational complexity of each equalization structure grows with the length of the channel time dispersion. Considering the Doppler and frequency selectivity characteristics of the channel, use the information in this table when selecting which equalization structure to use in your simulation.

| Equalizer Structure | Doppler Speed | Is Channel Frequency <br> Selective? | Computational <br> Complexity |
| :--- | :--- | :--- | :--- |
| Linear RLS | High | No | Medium |
| Linear LMS | Low | No | Lowest |
| Linear CMA | Low | No | Lowest |
| DFE RLS | High | Yes | Medium |
| DFE LMS | Low | Yes | Lowest |
| DFE CMA | Low | Yes | Lowest |
| MLSE | Low | Yes | Highest |

## Selected References 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.

## See Also

## Objects

comm.DecisionFeedbackEqualizer|comm.LinearEqualizer|comm.MLSEEqualizer

## Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

## More About

- "Adaptive Equalizers" on page 6-5
- "MLSE Equalizers" on page 6-35


## Adaptive Equalizers

## In this section...

"Symbol Tap Spacing" on page 6-5
"Linear Equalizers" on page 6-6
"Decision-Feedback Equalizers" on page 6-7
"Reference Signal and Operating Modes" on page 6-8
"Error Calculation" on page 6-8
"Updating Tap Weights" on page 6-8
"Configuring Adaptive Equalizers" on page 6-9
"Using Adaptive Equalizers in Simulink" on page 6-29
"Adaptive Equalization with Filtering and Fading Channel" on page 6-29
Adaptive equalizer structures provide suboptimal equalization of time variations in the propagation channel characteristics. However, these equalizers are appealing because their computational complexity is lower than "MLSE Equalizers" on page 6-35.

In Communications Toolbox, the comm. LinearEqualizer and comm.DecisionFeedbackEqualizer System objects and the Linear Equalizer and Decision Feedback Equalizer blocks use tap delay line filters to equalize a linearly modulated signal through a dispersive channel. These features output the estimate of the signal by using an estimate of the channel modeled as a finite input response (FIR) filter.

To decode a received signal, the adaptive equalizer:
1 Applies the FIR filter to the symbols in the input signal. The FIR filter tap weights correspond to the channel estimate.
2 Outputs the signal estimate and uses the signal estimate to update the tap weights for the next symbol. The signal estimate and updating of weights depends on the adaptive equalizer structure and algorithm.

Adaptive equalizer structure options are linear or decision-feedback. Adaptive algorithm options are least mean square (LMS), recursive mean square (RMS), or constant modulus algorithm (CMA). For background material on adaptive equalizers, see "Selected References for Equalizers" on page 6-3.

## Symbol Tap Spacing

You can configure the equalizer to operate as a symbol-spaced equalizer or as a fractional symbolspaced equalizer.

- To operate the equalizer at a symbol-spaced rate, specify the number of samples per symbol as 1. Symbol-rate equalizers have taps spaced at the symbol duration. Symbol-rate equalizers are sensitive to timing phase.
- To operate the equalizer at a fractional symbol-spaced rate, specify the number of input samples per symbol as an integer greater than 1 and provide an input signal oversampled at that sampling rate. Fractional symbol-spaced equalizers have taps spaced at an integer fraction of the input symbol duration. Fractional symbol-spaced equalizers are not sensitive to timing phase.

Note The MLSE equalizer supports fractional symbol spacing but using it is not recommended. The MLSE computational complexity and burden grows exponentially with the length of the channel time dispersion. Oversampling the input means multiplying the exponential term by the number of samples per symbol.

## Linear Equalizers

Linear equalizers can remove intersymbol interference (ISI) when the frequency response of a channel has no null. If a null exists in the frequency response of a channel, linear equalizers tend to enhance the noise. In this case, use decision feedback equalizers to avoid enhancing the noise.

A 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.

Linear equalizers can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, $K$, is 1 . The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, $K$, is an integer greater than 1 . Typically, $K$ is 4 for fractionally spaced equalizers. The output sample rate is $1 / T$ and the input sample rate is $K / T$, where $T$ is the symbol period. Tap-weight updating occurs at the output rate.

This schematic shows a linear equalizer with $L$ weights, a symbol period of $T$, and $K$ samples per symbol. If $K$ is 1 , the result is a symbol-spaced linear equalizer instead of a fractional symbol-spaced linear equalizer.


In each symbol period, the equalizer receives $K$ input samples at the tapped delay line. The equalizer then outputs a weighted sum of the values in the tapped delay line and updates the weights to prepare for the next symbol period.

## Decision-Feedback Equalizers

A decision feedback equalizer (DFE) is a nonlinear equalizer that reduces intersymbol interference (ISI) in frequency-selective channels. If a null exists in the frequency response of a channel, DFEs do not enhance the noise. A DFE consists of a tapped delay line that stores samples from the input signal and contains a forward filter and a feedback filter. The forward filter is similar to a linear equalizer. The feedback filter contains a tapped delay line whose inputs are the decisions made on the equalized 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.

DFEs can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, $K$, is 1 . The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, $K$, is an integer greater than 1. Typically, $K$ is 4 for fractional symbol-spaced equalizers. The output sample rate is $1 / T$ and the input sample rate is $K / T$. Tap weight updating occurs at the output rate.

This schematic shows a fractional symbol-spaced DFE with a total of $N$ weights, a symbol period of $T$, and $K$ samples per symbol. The filter has $L$ 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 fractional symbol-spaced DFE.


In each symbol period, the equalizer receives $K$ input samples at the forward filter and 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 Adaptive Algorithm block in the schematic jointly optimizes the forward and feedback weights. Joint optimization is especially important for convergence in the recursive least square (RLS) algorithm.

## Reference Signal and Operating Modes

In default applications, the equalizer first operates in training mode to gather information about the channel. The equalizer later switches to decision-directed mode.

- When the equalizer is operating in training mode, the reference signal is a preset, known transmitted sequence.
- When the equalizer is operating in decision-directed mode, the reference signal is a detected version of the output signal, denoted by $y_{d}$ in the schematic.

The CMA algorithm has no training mode. Training mode applies only when the equalizer is configured to use the LMS or RLS algorithm.

## Error Calculation

The error calculation operation produces a signal given by this expression, where $R$ is a constant related to the signal constellation.

$$
e=\left\{\begin{array}{cc}
d-y & \text { LMS or RLS } \\
y\left(R-|y|^{2}\right) & \text { CMA }
\end{array}\right.
$$

## Updating Tap Weights

- "Least Mean Square Algorithm" on page 6-9
- "Recursive Least Square Algorithm" on page 6-9
- "Constant Modulus Algorithm" on page 6-9

For linear and decision-feedback equalizer structures, the choice of LMS, RLS, or CMA determines the algorithms that are used to set the tap weights and perform the error calculation. The new set of tap weights depends on:

- The current set of tap weights
- The input signal
- The output signal
- The reference signal, $d$, for LMS and RLS adaptive algorithms only. The reference signal characteristics depend on the operating mode of the equalizer.


## Least Mean Square Algorithm

For the LMS algorithm, in the previous schematic, $w$ is a vector of all weights $w_{i}$, and $u$ is a vector of all inputs $u_{i}$. Based on the current set of weights, the LMS algorithm creates the new set of weights as

$$
w_{\text {new }}=w_{\text {current }}+(\text { StepSize }) u^{*} e .
$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed when using the LMS adaptive algorithm, use the maxstep object function. The * operator denotes the complex conjugate and the error calculation $e=d-y$.

## Recursive Least Square Algorithm

For the RLS algorithm, in the previous schematic, $w$ is the vector of all weights $w_{i}$, and $u$ is the vector of all inputs $u_{i}$. Based on the current set of inputs, $u$, and the inverse correlation matrix, $P$, the RLS algorithm first computes the Kalman gain vector, $K$, as

$$
K=\frac{P u}{(\text { ForgettingFactor })+u^{H} P u} .
$$

The forgetting factor used by the adaptive algorithm is specified as a scalar in the range $(0,1]$. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalized output signal to be less stable. $H$ denotes the Hermitian transpose. Based on the current inverse correlation matrix, the new inverse correlation matrix is

$$
P_{\text {new }}=\frac{P_{\text {current }}\left(1-K u^{H}\right)}{\text { ForgettingFactor }} .
$$

Based on the current set of weights, the RLS algorithm creates the new set of weights as

$$
w_{\text {new }}=w_{\text {current }}+K^{*} e .
$$

The * operator denotes the complex conjugate and the error calculation $e=d-y$.

## Constant Modulus Algorithm

For the CMA adaptive algorithm, in the previous schematic, $w$ is the vector of all weights $w_{i}$, and $u$ is the vector of all inputs $u_{i}$. Based on the current set of weights, the CMA adaptive algorithm creates the new set of weights as

$$
w_{\text {new }}=w_{\text {current }}+(\text { StepSize }) u^{*} e .
$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed by the CMA adaptive algorithm, use the maxstep object function. The * operator denotes the complex conjugate and the error calculation $e=y\left(R-|y|^{2}\right)$, where $R$ is a constant related to the signal constellation.

## Configuring Adaptive Equalizers

Choose the linear or decision-feedback equalizer structure. Decide which adaptive algorithm to use LMS, RLS, or CMA. Specify settings for structure and algorithm-specific operation modes.

Configuring an equalizer involves selecting a linear or decision-feedback structure, selecting an adaptive algorithm, and specifying the structure and algorithm specific operation modes.

- "Specify an Adaptive Equalizer" on page 6-10
- "Equalizer Training" on page 6-13
- "Managing Delays When Using Equalizers" on page 6-23

When deciding which adaptive algorithm best fits your needs, consider:

- The LMS algorithm executes quickly but converges slowly. Its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly. Its complexity grows approximately with the square of the number of weights. This algorithm can also be unstable when the number of weights is large.
- The constant modulus algorithm (CMA) is useful when no training signal is available. It works best for constant modulus modulations such as PSK.
- If CMA has no additional side information, it can introduce phase ambiguity. For example, the weights found by the CMA might produce a perfect QPSK constellation but introduce a phase rotation of 90,180 , or 270 degrees. In this case, employ a phase ambiguity correction algorithm or choose a differential modulation scheme. Differential modulation schemes are insensitive to phase ambiguity.

To view or change any properties of an adaptive equalizer, use the syntax described for channel objects in "Displaying and Changing Object Properties" on page 14-14.

For more information about adaptive algorithms, see the references listed in "Selected References for Equalizers" on page 6-3.

## Specify an Adaptive Equalizer

- "Defining an Equalizer Object" on page 6-10
- "Adaptive Algorithm Assignment" on page 6-12

To create an adaptive equalizer object for use in MATLAB, select the comm. LinearEqualizer or comm. DecisionFeedbackEqualizer System object. For Simulink, use the Linear Equalizer or Decision Feedback Equalizer block. Based on the propagation channel characteristics in your simulation, use the criteria in "Equalization" on page 6-2 to select the equalizer structure.

The equalizer object has many properties that record information about the equalizer. Properties can be related to:

- The structure of the equalizer, such as the number of taps.
- The adaptive algorithm that the equalizer uses, such as the step size in the LMS or CMA algorithm.
- Information about the current state of the equalizer. The equalizer object can output the values of the weights.

To view or change any properties of an equalizer object, use the syntax described for channel objects in "Displaying and Changing Object Properties" on page 14-14.

## Defining an Equalizer Object

The code creates equalizer objects for these configurations:

- A symbol-spaced linear RLS equalizer with 10 weights.
- A fractionally spaced linear RLS equalizer with 10 weights, a BPSK constellation, and two samples per symbol.
- A decision-feedback RLS equalizer with three weights in the feedforward filter and two weights in the feedback filter.

All three equalizer objects specify the RLS adaptive algorithm with a forgetting factor of 0.3.
Create equalizer objects of different types. The default settings are used for properties not set using 'Name, Value' pairs.

```
eqlin = comm.LinearEqualizer('Algorithm','RLS','NumTaps',10,'ForgettingFactor',0.3)
eqlin =
    comm.LinearEqualizer with properties:
                                    Algorithm: 'RLS'
                                    NumTaps: 10
            ForgettingFactor: 0.3000
        InitialInverseCorrelationMatrix: 0.1000
                        Constellation: [1x4 double]
                        ReferenceTap: 3
                            InputDelay: 0
            InputSamplesPerSymbol: 1
            TrainingFlagInputPort: false
                AdaptAfterTraining: true
            InitialWeightsSource: 'Auto'
                        WeightUpdatePeriod: 1
```

eqfrac = comm.LinearEqualizer('Algorithm','RLS','NumTaps',10,'ForgettingFactor',0.3, ...
'Constellation',[-1 1],'InputSamplesPerSymbol', 2)
eqfrac =
comm.LinearEqualizer with properties:
Algorithm: 'RLS'
NumTaps: 10
ForgettingFactor: 0.3000
InitialInverseCorrelationMatrix: 0.1000
Constellation: [-1 1]
ReferenceTap: 3
InputDelay: 0
InputSamplesPerSymbol: 2
TrainingFlagInputPort: false
AdaptAfterTraining: true
InitialWeightsSource: 'Auto'
WeightUpdatePeriod: 1
eqdfe = comm.DecisionFeedbackEqualizer('Algorithm','RLS','NumForwardTaps',3, ...
'NumFeedbackTaps',2,'ForgettingFactor',0.3)
eqdfe =
comm.DecisionFeedbackEqualizer with properties:
Algorithm: 'RLS'
NumForwardTaps: 3
NumFeedbackTaps: 2

```
        ForgettingFactor: 0.3000
InitialInverseCorrelationMatrix: 0.1000
    Constellation: [1x4 double]
            ReferenceTap: 3
                InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
    AdaptAfterTraining: true
    InitialWeightsSource: 'Auto'
    WeightUpdatePeriod: 1
```


## Adaptive Algorithm Assignment

Use the Algorithm property to assign the adaptive algorithm used by the equalizer.

## Algorithm Assignment

When creating the equalizer object, assign the adaptive algorithm.

```
eqlms = comm.LinearEqualizer('Algorithm','LMS');
```

Create the equalizer object with default property settings. LMS is the default adaptive algorithm.

```
eqrls = comm.LinearEqualizer;
```

eqrls.Algorithm
ans =
' LMS '

Update eqrls to use the RLS adaptive algorithm.

```
eqrls.Algorithm = 'RLS';
eqrls.Algorithm
ans =
'RLS'
```


## Cloning and Duplicating Objects

Configure a new equalizer object by cloning an existing equalizer object, and then changing its properties. Clone eqlms to create an independent equalizer, eqcma, then update the algorithm to 'CMA'.

```
eqcma = clone(eqlms);
```

eqcma.Algorithm
ans $=$
'LMS'
eqcma.Algorithm = 'CMA';
eqcma.Algorithm
ans $=$
'CMA'

If you want an independent duplicate, use the clone command.
eqlms.NumTaps

```
ans = 5
eq2 = eqlms;
eq2.NumTaps = 6;
eq2.NumTaps
ans = 6
eqlms.NumTaps
ans = 6
```

The clone command creates a copy of eqlms that is independent of eqlms. By contrast, the command eqB $=$ eqA creates eqB as a reference to eqA, so that eqB and eqA always have identical property settings.

## Equalizer Training

- "Linearly Equalize System By Using Different Training Schemes" on page 6-13
- "Linearly Equalize Symbols By Using EVM-Based Training" on page 6-20


## Linearly Equalize System By Using Different Training Schemes

Demonstrate linear equalization by using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel. Apply different equalizer training schemes and show the symbol error magnitude.

## System Setup

Simulate a QPSK-modulated system subject to AWGN. Transmit packets composed of 200 training symbols and 1800 random data symbols. Configure a linear LMS equalizer to recover the packet data.

```
M = 4;
numTrainSymbols = 200;
numDataSymbols = 1800;
SNR = 20;
trainingSymbols = pskmod(randi([0 M-1],numTrainSymbols,1),M,pi/4);
numPkts = 10;
lineq = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',5,'ReferenceTap',3,'StepSize',0.01);
```


## Train the Equalizer at the Beginning of Each Packet With Reset

Use prepended training symbols when processing each packet. After processing each packet, reset the equalizer. This reset forces the equalizer to train the taps with no previous knowledge. Equalizer error signal plots for the first, second, and last packet show higher symbol errors at the start of each packet.

```
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    rx = awgn(packet,SNR);
    [~,err] = lineq(rx,trainingSymbols);
    reset(lineq)
    if (ii ==1 || ii == 2 ||ii == numPkts)
```

```
        subplot(3,1,jj)
        plot(abs(err))
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols')
        ylabel('Error Magnitude')
        axis([0,length(packet),0,1])
        grid on;
        jj = jj+1;
    end
end
```





## Train the Equalizer at the Beginning of Each Packet Without Reset

Process each packet using prepended training symbols. Do not reset the equalizer after each packet is processed. By not resetting after each packet, the equalizer retains tap weights from training prior packets. Equalizer error signal plots for the first, second, and last packet show that after the initial training on the first packet, subsequent packets have fewer symbol errors at the start of each packet.

```
release(lineq)
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    channel = 1;
    rx = awgn(packet*channel,SNR);
```

```
    [~,err] = lineq(rx,trainingSymbols);
    if (ii ==1 || ii == 2 ||ii == numPkts)
        subplot(3,1,jj)
        plot(abs(err))
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols')
        ylabel('Error Magnitude')
        axis([0,length(packet),0,1])
        grid on;
        jj = jj+1;
    end
```

end




## Train the Equalizer Periodically

Systems with signals subject to time-varying channels require periodic equalizer training to maintain lock on the channel variations. Specify a system that has 200 symbols of training for every 1800 data symbols. Between training, the equalizer does not update tap weights. The equalizer processes 200 symbols per packet.

```
Rs = 1e6;
fd = 20;
spp = 200; % Symbols per packet
b = randi([0 M-1],numDataSymbols,1);
dataSym = pskmod(b,M,pi/4);
packet = [trainingSymbols; dataSym];
stream = repmat(packet,10,1);
```

```
tx = (0:length(stream)-1)'/Rs;
channel = exp(li*2*pi*fd*tx);
rx = awgn(stream.*channel,SNR); training phase.
```

```
release(lineq)
```

release(lineq)
lineq.AdaptAfterTraining = false
lineq.AdaptAfterTraining = false
lineq =
lineq =
comm.LinearEqualizer with properties:
comm.LinearEqualizer with properties:
Algorithm: 'LMS'
Algorithm: 'LMS'
NumTaps: 5
NumTaps: 5
StepSize: 0.0100
StepSize: 0.0100
Constellation: [1x4 double]
Constellation: [1x4 double]
ReferenceTap: 3
ReferenceTap: 3
InputDelay: 0
InputDelay: 0
InputSamplesPerSymbol: 1
InputSamplesPerSymbol: 1
TrainingFlagInputPort: false
TrainingFlagInputPort: false
AdaptAfterTraining: false
AdaptAfterTraining: false
InitialWeightsSource: 'Auto'
InitialWeightsSource: 'Auto'
WeightUpdatePeriod: 1

```
                WeightUpdatePeriod: 1
```

Set the AdaptAfterTraining property to false to stop the equalizer tap weight updates after the

Equalize the impaired data. Plot the angular error from the channel, the equalizer error signal, and signal constellation. As the channel varies, the equalizer output does not remove the channel effects. The output constellation rotates out of sync, resulting in bit errors.

```
[y,err] = lineq(rx,trainingSymbols);
figure
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('Time (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(err))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel Without Retraining')
```



[^0]

Set the TrainingInputPort property to true to configure the equalizer to retrain the taps when signaled by the trainFlag input. The equalizer trains only when trainFlag is true. After every 2000 symbols, the equalizer retrains the taps and keeps lock on variations of the channel. Plot the angular error from the channel, equalizer error signal, and signal constellation. As the channel varies, the equalizer output removes the channel effects. The output constellation does not rotate out of sync and bit errors are reduced.

```
release(lineq)
lineq.TrainingFlagInputPort = true;
symbolCnt = 0;
numPackets = length(rx)/spp;
trainFlag = true;
trainingPeriod = 2000;
eVec = zeros(size(rx));
yVec = zeros(size(rx));
for p=1:numPackets
    [yVec((p-1)*spp+1:p*spp,1),eVec((p-1)*spp+1:p*spp,1)] = ...
        lineq(rx((p-1)*spp+1:p*spp,1),trainingSymbols,trainFlag);
    symbolCnt = symbolCnt + spp;
    if symbolCnt >= trainingPeriod
        trainFlag = true;
        symbolCnt = 0;
    else
        trainFlag = false;
    end
end
figure
```

```
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('t (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(eVec))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel With Retraining')
```



Time-Varying Channel With Retraining


```
scatterplot(yVec)
```



## Linearly Equalize Symbols By Using EVM-Based Training

Recover QPSK symbols with a linear equalizer by using the constant modulus algorithm (CMA) and EVM-based taps training. When using blind equalizer algorithms, such as CMA, train the equalizer taps by using the AdaptWeights property to start and stop training. Helper functions are used to generate plots and apply phase correction.

Initialize system variables.

```
rng(123456);
M = 4; % QPSK
numSymbols = 100;
numPackets = 5000;
raylChan = comm.RayleighChannel('PathDelays',[0 1], ...
    'AveragePathGains',[0 -12],'MaximumDopplerShift',1e-5);
SNR = 50;
adaptWeights = true;
```

Create the equalizer and EVM System objects. The equalizer System object specifies a linear equalizer by using the CMA adaptive algorithm. Call the helper function to initialize figure plots.

```
lineq = comm.LinearEqualizer('Algorithm','CMA', ...
    'NumTaps',5,'ReferenceTap',3, ...
    'StepSize',0.03,'AdaptWeightsSource','Input port')
lineq =
    comm.LinearEqualizer with properties:
```

Algorithm: 'CMA'
NumTaps: 5
StepSize: 0.0300
Constellation: [1x4 double]
ReferenceTap: 3
InputSamplesPerSymbol: 1
AdaptWeightsSource: 'Input port'
InitialWeightsSource: 'Auto' WeightUpdatePeriod: 1
info(lineq)

```
ans = struct with fields:
```

    Latency: 2
    evm = comm.EVM('ReferenceSignalSource', ...
'Estimated from reference constellation');
[errPlot,evmPlot,scatSym,adaptState] = initFigures(numPackets,lineq);

## Equalization Loop

To implement the equalization loop:
1 Generate PSK data packets.
2 Apply Rayleigh fading and AWGN to the transmission data.
3 Apply equalization to the received data and phase correction to the equalizer output.
4 Estimate the EVM and toggle the adaptWeights flag to true or false based on the EVM level.
5 Update the figure plots.
for $p=1$ : numPackets
data $=$ randi([0 M-1], numSymbols,1);
tx = pskmod(data, M, pi/4);
$r x=\operatorname{awgn}(r a y l C h a n(t x), S N R)$;
rxDelay = finddelay(rx,tx);
[y,err,wts] = lineq(rx,adaptWeights);
$y=$ phaseCorrection(y);
evmEst = evm(y);
adaptWeights $=$ (evmEst > 20);
updateFigures(errPlot,evmPlot,scatSym, adaptState, ... wts, $y$ (end), evmEst, adaptWeights, $p$, numPackets)
end


The figure plots show that, as the EVM varies, the equalizer toggles in and out of decision-directed weight adaptation mode.

## Helper Functions

This helper function initializes figures that show a quad plot of simulation results.

```
function [errPlot,evmPlot,scatter,adaptState] = initFigures(numPkts,lineq)
yVec = nan(numPkts,1);
evmVec = nan(numPkts,1);
wVec = zeros(lineq.NumTaps,1);
adaptVec = nan(numPkts,1);
figure
subplot(2,2,1)
evmPlot = stem(wVec);
grid on; axis([1 lineq.NumTaps 0 1.8])
xlabel('Taps'); ylabel('|Weights|'); title('Tap Weight Magnitude')
subplot(2,2,2)
scatter = plot(yVec, '.');
axis square; axis([-1.2 1.2 -1.2 1.2]); grid on
xlabel('In-phase'); ylabel('Quadrature'); title('Scatter Plot');
subplot(2,2,3)
```

```
adaptState = plot(adaptVec);
grid on; axis([0 numPkts -0.2 1.2])
ylabel('Training'); xlabel('Symbols'); title('Adapt Weights Signal')
subplot(2,2,4)
errPlot = plot(evmVec);
grid on; axis([1 numPkts 0 100])
xlabel('Symbols'); ylabel('EVM (%)'); title('EVM')
end
```

This helper function updates figures.

```
function updateFigures(errPlot,evmPlot,scatSym, ...
    adaptState,w,y,evmEst, adaptWts, p, numFrames)
persistent yVec evmVec adaptVec
if p == 1
    yVec = nan(numFrames,1);
    evmVec = nan(numFrames,1);
    adaptVec = nan(numFrames,1);
end
yVec(p) = y;
evmVec(p) = evmEst;
adaptVec(p) = adaptWts;
errPlot.YData = abs(evmVec);
evmPlot.YData = abs(w);
scatSym.XData = real(yVec);
scatSym.YData = imag(yVec);
adaptState.YData = adaptVec;
drawnow limitrate
end
```

This helper function applies phase correction.

```
function y = phaseCorrection(y)
a = angle(y((real(y) > 0) & (imag(y) > 0)));
a(a<0.1) = a(a< 0.1) + pi/2;
theta = mean(a) - pi/4;
y = y * exp(-1i*theta);
end
```


## Managing Delays When Using Equalizers

For proper equalization, you must determine and account for system delays. As shown in the following example, you can use the finddelay function to determine the system delay. This example uses LMS linear equalization but the same approach is valid for the RLS and CMA adaptive algorithms and for decision feedback equalizers.

## Linearly Equalize Delayed Signal

Simulate a system with delay between the transmitted symbols and received samples. Typical systems have transmitter and receiver filters that result in a delay. This delay must be accounted for to synchronize the system. In this example, the system delay is introduced without transmit and receive filters. Linear equalization, using the least mean squares (LMS) algorithm, recovers QPSK symbols.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
mpChan = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
systemDelay = dsp.Delay(20);
snr = 24;
```

Generate QPSK-modulated symbols. Apply multipath channel filtering, a system delay, and AWGN to the transmitted symbols.

```
data = randi([0 M-1],numSymbols,1);
tx = pskmod(data,M,pi/4); % OQPSK
delayedSym = systemDelay(filter(mpChan,1,tx));
rx = awgn(delayedSym,snr,'measured');
```

Create equalizer and EVM System objects. The equalizer System object specifies a linear equalizer that uses the LMS algorithm.

```
lineq = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',9,'ReferenceTap',5);
evm = comm.EVM('ReferenceSignalSource', ...
    'Estimated from reference constellation');
```


## Equalize Without Adjusting Input Delay

Equalize the received symbols.

```
[y1,err1,wts1] = lineq(rx,tx(1:numTrainingSymbols,1));
```

Find the delay between the received symbols and the transmitted symbols by using the finddelay function.

```
rxDelay = finddelay(tx,rx)
rxDelay = 20
```

Display the equalizer information. The latency value indicates the delay introduced by the equalizer. Calculate the total delay as the sum of rxDelay and the equalizer latency.

```
eqInfo = info(lineq)
eqInfo = struct with fields:
    Latency: 4
totalDelay = rxDelay + eqInfo.Latency;
```

Until the equalizer output converges, the symbol error rate is high. Plot the error output, err1, to determine when the equalized output converges.

```
plot(abs(err1))
xlabel('Symbols')
ylabel('Error Magnitude')
title('Equalizer Error Signal')
```

Equalizer Error Signal


The plot shows excessive errors beyond the 1000 symbols training period. When demodulating symbols and computing symbol errors, to account for the unconverged output and the system delay between the equalizer output and transmitted symbols, skip the first 2000 symbols.

```
dataRecl = pskdemod(y1(2000+totalDelay:end),M,pi/4);
symErrWithDelay = symerr(data(2000:end-totalDelay),dataRec1)
symErrWithDelay = 5999
evmWithDelay = evm(y1)
evmWithDelay = 29.5795
```

The error rate and EVM are high because the receive delay was not accounted for in the equalizer System object.

## Adjust Input Delay in Equalizer

Equalize the received data by using the delay value to set the InputDelay property. Because InputDelay is a nontunable property, you must release the lineq System object to reconfigure the InputDelay property. Equalize the received symbols.

```
release(lineq)
lineq.InputDelay = rxDelay
lineq =
    comm.LinearEqualizer with properties:
```

```
        Algorithm: 'LMS'
        NumTaps: 9
        StepSize: 0.0100
        Constellation: [1x4 double]
    ReferenceTap: 5
        InputDelay: 20
InputSamplesPerSymbol: 1
TrainingFlagInputPort: false
        AdaptAfterTraining: true
    InitialWeightsSource: 'Auto'
        WeightUpdatePeriod: 1
```

[y2,err2,wts2] = lineq(rx,tx(1:numTrainingSymbols,1));

Plot the tap weights and equalized error magnitude. A stem plot shows the equalizer tap weights before and after the system delay is removed. A 2-D line plot shows the slower equalizer convergence for the delayed signal as compared to the signal with the delay removed.

```
subplot(2,1,1)
stem([real(wts1), real(wts2)])
xlabel('Taps')
ylabel('Tap Weight Real')
legend('rxDelayed','rxDelayRemoved')
grid on
subplot(2,1,2)
stem([imag(wts1),imag(wts2)])
xlabel('Taps')
ylabel('Tap Weight Imaginary')
legend('rxDelayed','rxDelayRemoved')
grid on
```


figure
plot([abs(err1), abs(err2)])
xlabel('Symbols')
ylabel('Error Magnitude')
legend('rxDelayed','rxDelayRemoved')
grid on


Plot error output of the equalized signals, rxDelayed and rxDelayRemoved. For the signal that has the delay removed, the equalizer converges during the 1000 symbol training period. When demodulating symbols and computing symbol errors, to account for the unconverged output and the system delay between the equalizer output and transmitted symbols, skip the first 500 symbols.
Reconfiguring the equalizer to account for the system delay enables better equalization of the signal, and reduces symbol errors and the EVM.

```
eqInfo = info(lineq)
eqInfo = struct with fields:
    Latency: 4
totalDelay = rxDelay + eqInfo.Latency;
dataRec2 = pskdemod(y2(500+totalDelay:end),M,pi/4);
symErrDelayRemoved = symerr(data(500:end-totalDelay),dataRec2)
symErrDelayRemoved = 0
evmDelayRemoved = evm(y2(500+totalDelay:end))
```


## Using Adaptive Equalizers in Simulink

## Adaptive Equalization with Filtering and Fading Channel

This model shows the behavior of the selected adaptive equalizer in a communication link that has a fading channel. The transmitter and receiver have root raised cosine pulse shaped filtering. A subsystem block enables you to select between linear or decision feedback equalizers that usie the least mean square (LMS) or recursive least square (RLS) adaptive algorithm.

## Model Structure

- The transmitter generates 16QAM random signal data that includes a training sequence and applies root raised cosine pulse shaped filtering.
- Channel impairments include multipath fading, Doppler shift, carrier frequency offset, variable integer delay, free space path loss, and AWGN.
- The receiver applies root raised cosine pulse shaped filtering, AGC, includes equalizer mode control to enable training and enables you to select the equalizer algorithm from these choices.

| Selection | Equalizer Algorithm |
| :--- | :--- |
| LMS Linear | Linear least mean square equalizer |
| LMS DFE | Decision feedback least mean square equalizer |
| RLS Linear | Linear recursive least square equalizer |
| RLS DFE | Decision feedback recursive least square equalizer |

- Scopes help you understand how the different equalizers and adaptive algorithms behave.

Explore Example Model


## Experimenting with the model

This model provides several ways for you to change settings and observe the results. The InitFcn found in File>Model Properties>Callbacks calls cm_ex_adaptive_eq_with_fading_init to initialize the model. This file enables you to vary settings in the model, including:

- System parameters, such as SNR.
- Pulse shaping filter parameters, such as rolloff and filter length
- Path loss value.
- Channel conditions: Rayleigh or Rician fading, channel path gains, channel path delays, and Doppler shift.
- Equalizer choice and configuration.


## Model Considerations

This non-standards-based communication link is representative of a modern communications system.

- The optimal equalizer configuration depends on the channel conditions. The initialization file sets the Doppler shift and multipath fading channel parameters that highlight the capabilities of different equalizers.

- The decision feedback equalizer structure performs better than the linear equalizer structure for higher intersymbol interference.
- The RLS algorithm performs better than the LMS algorithm for higher Doppler frequencies.
- The LMS algorithm executes quickly, converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, its complexity grows approximately as the square of the number of weights. It can be unstable when the number of weights is large.
- The channels exercised for different equalizers have the following characteristics.

| Selection | Channel Characteristics |
| :--- | :--- |
| LMS Linear | 3 tap multipath fading channel with 10 Hz Doppler shift |
| LMS DFE | 5 tap multipath fading channel with 25 Hz Doppler shift |
| RLS Linear | 3 tap multipath fading channel with 50 Hz Doppler shift |
| RLS DFE | 5 tap multipath fading channel with 100 Hz Doppler shift |

- Initial settings for other channel impairments are the same for all equalizers. Carrier frequency offset value is set to 50 Hz . Free space path loss is set to 60 dB . Variable integer delay is set to 2 samples, which requires the equalizers to perform some timing recovery.

Deep channel fades and path loss can cause the equalizer input signal level to be much less than the desired output signal level and result in unacceptably long equalizer convergence time. The AGC block adjusts the magnitude of received signal to reduce the equalizer convergence time. You must adjust the optimal AGC output power level based on the modulation scheme selected. For 16QAM, a desired output power of 10 W is used.

Training of the equalizer is performed at the beginning of the simulation.

## Running the Simulation

Running the simulation computes symbol error statistics and produces these figures:

- A constellation diagram of the signal after the receive filter.
- A constellation diagram of the signal after AGC.
- A constellation diagram of the signal after equalization with signal quality measurements shown.
- An equalizer error plot.

For the plots shown here, the equalizer algorithm selected is RLS Linear. Monitoring these figures, you can see that the received signal quality fluctuates as simulation time progresses.

The After Rx Filter and After AGC constellation plots show the signal before equalization. After AGC shows the impact of the channel conditions on the transmitted signal. The After Eq plot shows the signal after equalization. The signal plotted in the constellation diagram after equalization shows the variation in signal quality based on the effectiveness of the equalization process. Throughout the simulation, the signal constellations plotted before equalization deviate noticeably from a 16QAM signal constellation. The After Eq constellation improves or degrades as the equalizer error signal varies. The Eq error plotted in the Eq Error plot, indicates poor equalization at the start of the simulation. The error degrades at first then improves as the equalizer converges.



## Further Exploration

Double-click the Equalizer Selector block and select a different equalizer. Run the simulation to see the performance of the various equalizer options. You can use the signal logger to compare the results from this experimentation. In the block diagram, right-click on signal wires and select Log Selected Signals. If you have enabled signal logging, after the simulation run finishes, open the Simulation Data Inspector to view the logged signals.

At the MATLAB ${ }^{\text {TM }}$ command prompt, enter edit cm_ex_adaptive_eq_with_fading_init.m to open the initialization file, then modify a parameter and rerun the simulation. For example, adjust the channel characteristics (params.maxDoppler|, params. pathDelays, and params. pathGains). The RLS adaptive algorithm performs better than the LMS adaptive algorithm as the maximum Doppler is increased.

## See Also

Objects
comm. DecisionFeedbackEqualizer|comm. LinearEqualizer|comm.MLSEEqualizer

## Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

## More About

- "Equalization" on page 6-2
- "MLSE Equalizers" on page 6-35


## MLSE Equalizers

## In this section...

"Equalize a Vector Signal in MATLAB" on page 6-35
"Equalizing Signals in Continuous Operation Mode" on page 6-36
"Use a Preamble or a Postamble" on page 6-39
"Using MLSE Equalizers in Simulink" on page 6-40
"MLSE Equalization with Dynamically Changing Channel" on page 6-40
Maximum-Likelihood Sequence Estimation (MLSE) equalizers provide optimal equalization of time variations in the propagation channel characteristics. However, MLSE equalizers are sometimes less appealing because their computational complexity is higher than "Adaptive Equalizers" on page 6-5.

In Communications Toolbox, the mlseeq function, comm.MLSEEqualizer System object, and MLSE Equalizer block use the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. These features output the maximum likelihood sequence estimate of the signal by using an estimate of the channel modeled as a finite input response (FIR) filter.

To decode a received signal, the MLSE equalizer:
1 Applies the FIR filter to the symbols in the input signal. The FIR filter tap weights correspond to the channel estimate.
2 Uses the Viterbi algorithm to compute the traceback paths and the state metric. These values are 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 theoretically possible performance, but is computationally intensive.

For background material on MLSE equalizers, see "Selected References for Equalizers" on page 6-3.

## Equalize a Vector Signal in MATLAB

You can use the mlseeq function or comm.MLSEEqualizer System object for MLSE equalization in MATLAB. The examples in this section call the ml seeq function. A similar workflow applies when using the comm.MLSEEqualizer System object. For examples that use the System object, see the comm.MLSEEqualizer System object reference page.

The ml seeq function has two operation modes:

- Continuous operation mode enables you to process a series of vectors by using repeated calls to ml seeq. The function saves its internal state information from one call to the next. To learn more, see Equalizing in Continuous Operation Mode on page 6-36.
- 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 6-39.

If you are not processing a series of vectors and do not need to specify a preamble or postamble, the operational modes are nearly identical. They differ in that continuous operation mode incurs a delay,
while reset operation mode does not. The following example uses reset operation mode. If you modify the example to run using continuous operation mode, there will be delay in the equalized output. To learn more about this delay, see Delays in Continuous Operation Mode on page 6-37.

## Use mlseeq to Equalize a Vector Signal

In its simplest form, the ml seeq 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.
- The Viterbi algorithm traceback depth. Larger values for the traceback depth can improve the results from the equalizer but increase the computation time.

Generate a PSK modulated signal with modulation order set to four.

```
M = 4;
msg = pskmod([1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]',M);
```

Filter the modulated signal through a distortion channel.

```
chcoeffs = [.986; .845; .237; .12345+.31i];
filtmsg = filter(chcoeffs,1,msg);
```

Define the reference constellation, traceback length, and channel estimate for the MLSE equalizer. In this example, the exact channel is provided as the channel estimate.

```
const = pskmod([0:M-1],M);
tblen = 10;
chanest = chcoeffs;
```

Equalize the received signal.

```
msgEq = mlseeq(filtmsg,chanest,const,tblen,'rst');
isequal(msg,msgEq)
ans = logical
    1
```


## Equalizing Signals 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, ml seeq can save its internal state information for use in a subsequent invocation and can initialize by using previously stored state information. To choose continuous operation mode when invoking ml seeq, specify 'cont ' as an input argument.

Note Continuous operation mode incurs a delay, as described in Delays in Continuous Operation Mode on page 6-37. This mode cannot accommodate a preamble or postamble.

## Procedure for Continuous Operation Mode

When using continuous operation mode within a loop, preallocate three empty matrix variables to store the state metrics, traceback states, and traceback inputs for the equalizer before the equalization loop starts. Inside the loop, invoke mlseeq using a syntax such as:

```
sm = [];
ts = [];
ti = [];
    [y,sm,ts,ti] = mlseeq(x,chcoeffs,const,tblen,'cont',nsamp,sm,ts,ti);
end
```

Using $\mathrm{sm}, \mathrm{ts}$, and ti as input arguments causes ml seeq to continue operating from where it finished in the previous iteration. Using $\mathrm{sm}, \mathrm{ts}$, and ti as output arguments causes ml seeq to update the state information at the end of the current iteration. In the first iteration, $\mathrm{sm}, \mathrm{ts}$, and ti start as empty matrices, so the first invocation of the ml seeq 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. The first tblen output symbols are unrelated to the input signal and the last tblen input symbols are unrelated to the output signal. For example, this command uses a traceback depth of 3. The first three output symbols are unrelated to the input signal of ones $(1,10)$.

```
\(y=m l\) seeq \((\) ones \((1,10), 1,[-7: 2: 7], 3\), 'cont')
\(\begin{array}{lllllllllll}y=7 & -7 & -7 & 1 & 1 & 1 & 1 & 1 & 1 & 1\end{array}\)
```

It is important to keep track of delays introduced by different portions of the communications system. The "Use mlseeq to Equalize a Vector in Continuous Operation Mode" on page 6-37 example illustrates how to account for the delay when computing an error rate.

## Use mlseeq to Equalize a Vector in Continuous Operation Mode

This example shows the procedure for using the continuous operation mode of the ml seeq function within a loop.

## Initialize Variables

Specify run-time variables.

```
numsym = 200; % Number of symbols in each iteration
numiter = 25; % Number of iterations
M = 4; % Use 4-PSK modulation
qpskMod = comm.QPSKModulator('PhaseOffset',0);
chcoeffs = [1 ; 0.25]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
```

To initialize the equalizer, define parameters for the reference constellation, traceback length, number of samples per symbol, and the state variables sm, ts, and ti.

```
const = qpskMod((0:M-1)');
tblen = 10;
nsamp = 1;
sm = [];
```

```
ts = [];
ti = [];
```

Define variables to accumulate results from each iteration of the loop.

```
fullmodmsg = [];
fullfiltmsg = [];
fullrx = [];
```


## Simulate the System by Using a Loop

Run the simulation in a loop that generates random data, modulates the data by using baseband PSK modulation, and filters the data. The ml seeq function equalizes the filtered data. The loop also updates the variables that accumulate results from each iteration of the loop.

```
for jj = 1:numiter
    msg = randi([0 M-1],numsym,1); % Random signal vector
    modmsg = qpskMod(msg); % PSK-modulated signal
    filtmsg = filter(chcoeffs,1,modmsg); % Filtered signal
    % Equalize the signal.
    [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

Compute 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, 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.

Taking the delay into account, compute the total number of symbol errors.

```
hErrorCalc = comm.ErrorRate('ReceiveDelay',10);
err = step(hErrorCalc, fullmodmsg, fullrx);
numsymerrs = err(1)
numsymerrs = 0
```

Plot the signal constellation before and after equalization. The points in the equalized signal coincide with the points of the ideal signal constellation for 4-PSK.

```
h = scatterplot(fullfiltmsg);
hold on;
scatterplot(fullrx,1,0,'r*',h);
legend('Filtered signal before equalization','Equalized signal',...
    'Location','NorthOutside');
hold off;
```



## Use a Preamble or a 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 ml seeq 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 [14 4] and a 4-PSK signal constellation of [1 $\left.\begin{array}{lll}1 & -1 & -j\end{array}\right]$ indicates that the modulated signal begins with [ $1-\mathrm{j}-\mathrm{j}]$.

If your system uses a preamble without a postamble, use a postamble vector of [] when invoking ml seeq. If your system uses a postamble without a preamble, use a preamble vector of [].

## Recover Message Containing Preamble

Recover a message that includes a preamble, equalize the signal, and check the symbol error rate.
Specify the modulation order, equalizer traceback depth, number of samples per symbol, preamble, and message length.

```
M = 4;
```

tblen = 16;
nsamp = 1;
preamble = [3;1];
msgLen = 500;

Generate the reference constellation.
const $=$ pskmod (0:3,4);
Generate a message by using random data and prepend the preamble to the message. Modulate the random data.

```
msgData = randi([0 M-1],msgLen,1);
msgData = [preamble; msgData];
msgSym = pskmod(msgData,M);
```

Filter the data through a distortion channel and add Gaussian noise to the signal.

```
chcoeffs = [0.623; 0.489+0.234i; 0.398i; 0.21];
chanest = chcoeffs;
msgFilt = filter(chcoeffs,1,msgSym);
msgRx = awgn(msgFilt,9,'measured');
```

Equalize the received signal. To configure the equalizer, provide the channel estimate, reference constellation, equalizer traceback depth, operating mode, number of samples per symbol, and preamble. The same preamble symbols appear at the beginning of the message vector and in the syntax for ml seeq. Because the system does not use a postamble, an empty vector is specified as the last input argument in this ml seeq syntax.

Check the symbol error rate of the equalized signal. Run-to-run results vary due to use of random numbers.

```
eqSym = mlseeq(msgRx,chanest,const,tblen,'rst',nsamp,preamble,[]);
[nsymerrs,ser] = symerr(msgSym,eqSym)
nsymerrs = 8
ser = 0.0159
```


## Using MLSE Equalizers in Simulink

The MLSE Equalizer block uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The block outputs the maximum likelihood sequence estimate (MLSE) of the signal by using your estimate of the channel modeled as a finite input response (FIR) filter. When using the MLSE Equalizer block, you specify the channel estimate and the signal constellation of the input signal. You can also specify an expected input signal preamble and postamble as input parameters to the MLSE Equalizer block.

## MLSE Equalization with Dynamically Changing Channel

Use a Maximum Likelihood Sequence Estimation (MLSE) equalizer to equalize the effects of a multipath Rayleigh fading channel. The MLSE equalizer inputs data that has passed through a time varying dispersive channel and an estimate of the channel. The channel estimate contains dynamically evolving channel coefficients of a two-path Rayleigh fading channel.

## Model Structure

- The transmitter generates QPSK random signal data.
- Channel impairments include multipath fading and AWGN.
- The receiver applies MLSE equalization and QPSK demodulation.
- The model uses scopes and a BER calculation to show the system behavior.


## Explore Example Model



## Experimenting with the model

The Bernoulli Binary Generator block sample time of 5e-6 seconds corresponds to a bit rate of 200 kbps and a QPSK symbol rate of $100 \mathrm{ksym} / \mathrm{sec}$.

The Multipath Rayleigh Fading Channel block settings are:

- Maximum Doppler shift is 30 Hz .
- Discrete path delay is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This delay reflects the simplest delay vector for a two-path channel.
- Average path gain is [0-10].
- Average path gains are normalized to 0 dB so that the average power input to the AWGN block is 1 W.

The MLSE Equalizer block has the Traceback depth set to 10. Vary this depth to study its effect on Bit Error rate (BER).

The QPSK demodulator accepts an N-by-1 input frame and generates a 2 N -by-1 output frame. This output frame and a traceback depth of 10 results in a delay of 20 bits. The model performs framebased processing on frames that have 100 samples per frame. Due to the frame-based processing,
there is a inherent delay of 100 bits in the model. The combined receive delay of 120 is set in the Receive delay parameter of the Error Rate Calculation block, aligning the samples.

The computed BER is displayed. Constellation plots show the constellation before and after equalization.
$B E R=0.033645$



## See Also

## Objects

comm. DecisionFeedbackEqualizer|comm. LinearEqualizer|comm.MLSEEqualizer

## Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

## More About

- "Equalization" on page 6-2
- "Adaptive Equalizers" on page 6-5


## Equalizer Examples (new \& old)

- "DF Equalize QPSK-Modulated Signal in Simulink" on page 7-2
- "Linearly Equalize QPSK-Modulated Signal in Simulink" on page 7-6
- "Adaptive Equalization with Filtering and Fading Channel" on page 7-10
- "MLSE Equalization with Dynamically Changing Channel" on page 7-16
- "Adaptive Equalization" on page 7-19
- "Equalize BSPK Signal" on page 7-27
- "Compare RLS and LMS Algorithms" on page 7-31


## DF Equalize QPSK-Modulated Signal in Simulink

Apply decision feedback equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel.


Equalizer converges around
1000 symbols. Set
"Computation Delay" to 1000
Copyright 2018 The MathWorks, Inc.

The slex_dfeq_qpsk_signal model generates an $\mathrm{M}=4$ sequence using the Random Integer Generator block. The sequence is modulated with the M-PSK Modulator Baseband block, filtered with the Discrete FIR Filter block, and then impaired with the AWGN block. The Decision Feedback Equalizer block equalizes the data sequence, the data is demodulated with the M-PSK Demodulator Baseband block, and the bit error rate is computed. The signal path out of the modulator is split to a Selector block, which provides the first 1000 symbols of the modulated signal to the equalizer as an initial training sequence.

No delay is introduced between the transmitted and received signal because the maximum tap value is the first tap of the discrete FIR filter and the equalizer reference tap is 1 . The equalizer converges after around 1000 symbols so this value is used for the computation delay of the Error Rate Calculation block.

The computed error rate is displayed and plots show the equalized constellation, equalized tap weights, and signal error magnitude.

Computed error rate $=0.0011111$




## Linearly Equalize QPSK-Modulated Signal in Simulink

Apply linear equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel.


[^1]Copyright 2018 The MathWorks, Inc.

The slex_lineq_qpsk_signal model generates an $\mathrm{M}=4$ sequence using the Random Integer Generator block. The sequence is modulated with the M- PSK Modulator Baseband block, filtered with the Discrete FIR Filter block, and then impaired with the AWGN block. The Linear Equalizer block equalizes the data sequence, the data is demodulated with the M-PSK Demodulator Baseband block, and the bit error rate is computed. The signal path out of the modulator is split to a Selector block, which provides the first 1000 symbols of the modulated signal to the equalizer as an initial training sequence.

No delay is introduced between the transmitted and received signal because the maximum tap value is the first tap of the discrete FIR filter and the equalizer reference tap is 1 . The equalizer converges after around 1000 symbols so this value is used for the computation delay of the Error Rate Calculation block.

The computed error rate is displayed and plots show the equalized constellation, equalized tap weights, and signal error magnitude.

Computed error rate $=0.0024444$




## Adaptive Equalization with Filtering and Fading Channel

This model shows the behavior of the selected adaptive equalizer in a communication link that has a fading channel. The transmitter and receiver have root raised cosine pulse shaped filtering. A subsystem block enables you to select between linear or decision feedback equalizers that usie the least mean square (LMS) or recursive least square (RLS) adaptive algorithm.

## Model Structure

- The transmitter generates 16QAM random signal data that includes a training sequence and applies root raised cosine pulse shaped filtering.
- Channel impairments include multipath fading, Doppler shift, carrier frequency offset, variable integer delay, free space path loss, and AWGN.
- The receiver applies root raised cosine pulse shaped filtering, AGC, includes equalizer mode control to enable training and enables you to select the equalizer algorithm from these choices.

| Selection | Equalizer Algorithm |
| :--- | :--- |
| LMS Linear | Linear least mean square equalizer |
| LMS DFE | Decision feedback least mean square equalizer |
| RLS Linear | Linear recursive least square equalizer |
| RLS DFE | Decision feedback recursive least square equalizer |

- Scopes help you understand how the different equalizers and adaptive algorithms behave.


## Explore Example Model



## Experimenting with the model

This model provides several ways for you to change settings and observe the results. The InitFcn found in File>Model Properties>Callbacks calls cm_ex_adaptive_eq_with_fading_init to initialize the model. This file enables you to vary settings in the model, including:

- System parameters, such as SNR.
- Pulse shaping filter parameters, such as rolloff and filter length
- Path loss value.
- Channel conditions: Rayleigh or Rician fading, channel path gains, channel path delays, and Doppler shift.
- Equalizer choice and configuration.


## Model Considerations

This non-standards-based communication link is representative of a modern communications system.

- The optimal equalizer configuration depends on the channel conditions. The initialization file sets the Doppler shift and multipath fading channel parameters that highlight the capabilities of different equalizers.

- The decision feedback equalizer structure performs better than the linear equalizer structure for higher intersymbol interference.
- The RLS algorithm performs better than the LMS algorithm for higher Doppler frequencies.
- The LMS algorithm executes quickly, converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, its complexity grows approximately as the square of the number of weights. It can be unstable when the number of weights is large.
- The channels exercised for different equalizers have the following characteristics.

| Selection | Channel Characteristics |
| :--- | :--- |
| LMS Linear | 3 tap multipath fading channel with 10 Hz Doppler shift |
| LMS DFE | 5 tap multipath fading channel with 25 Hz Doppler shift |
| RLS Linear | 3 tap multipath fading channel with 50 Hz Doppler shift |
| RLS DFE | 5 tap multipath fading channel with 100 Hz Doppler shift |

- Initial settings for other channel impairments are the same for all equalizers. Carrier frequency offset value is set to 50 Hz . Free space path loss is set to 60 dB . Variable integer delay is set to 2 samples, which requires the equalizers to perform some timing recovery.

Deep channel fades and path loss can cause the equalizer input signal level to be much less than the desired output signal level and result in unacceptably long equalizer convergence time. The AGC block adjusts the magnitude of received signal to reduce the equalizer convergence time. You must adjust the optimal AGC output power level based on the modulation scheme selected. For 16QAM, a desired output power of 10 W is used.

Training of the equalizer is performed at the beginning of the simulation.

## Running the Simulation

Running the simulation computes symbol error statistics and produces these figures:

- A constellation diagram of the signal after the receive filter.
- A constellation diagram of the signal after AGC.
- A constellation diagram of the signal after equalization with signal quality measurements shown.
- An equalizer error plot.

For the plots shown here, the equalizer algorithm selected is RLS Linear. Monitoring these figures, you can see that the received signal quality fluctuates as simulation time progresses.

The After Rx Filter and After AGC constellation plots show the signal before equalization. After AGC shows the impact of the channel conditions on the transmitted signal. The After Eq plot shows the signal after equalization. The signal plotted in the constellation diagram after equalization shows the variation in signal quality based on the effectiveness of the equalization process. Throughout the simulation, the signal constellations plotted before equalization deviate noticeably from a 16QAM signal constellation. The After Eq constellation improves or degrades as the equalizer error signal varies. The Eq error plotted in the Eq Error plot, indicates poor equalization at the start of the simulation. The error degrades at first then improves as the equalizer converges.



## Further Exploration

Double-click the Equalizer Selector block and select a different equalizer. Run the simulation to see the performance of the various equalizer options. You can use the signal logger to compare the results from this experimentation. In the block diagram, right-click on signal wires and select Log Selected Signals. If you have enabled signal logging, after the simulation run finishes, open the Simulation Data Inspector to view the logged signals.

At the MATLAB ${ }^{\text {TM }}$ command prompt, enter edit cm_ex_adaptive_eq_with_fading_init.m to open the initialization file, then modify a parameter and rerun the simulation. For example, adjust the channel characteristics (params.maxDoppler|, params. pathDelays, and params. pathGains). The RLS adaptive algorithm performs better than the LMS adaptive algorithm as the maximum Doppler is increased.

## MLSE Equalization with Dynamically Changing Channel

Use a Maximum Likelihood Sequence Estimation (MLSE) equalizer to equalize the effects of a multipath Rayleigh fading channel. The MLSE equalizer inputs data that has passed through a time varying dispersive channel and an estimate of the channel. The channel estimate contains dynamically evolving channel coefficients of a two-path Rayleigh fading channel.

## Model Structure

- The transmitter generates QPSK random signal data.
- Channel impairments include multipath fading and AWGN.
- The receiver applies MLSE equalization and QPSK demodulation.
- The model uses scopes and a BER calculation to show the system behavior.


## Explore Example Model



## Experimenting with the model

The Bernoulli Binary Generator block sample time of 5e-6 seconds corresponds to a bit rate of 200 kbps and a QPSK symbol rate of $100 \mathrm{ksym} / \mathrm{sec}$.

The Multipath Rayleigh Fading Channel block settings are:

- Maximum Doppler shift is 30 Hz .
- Discrete path delay is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This delay reflects the simplest delay vector for a two-path channel.
- Average path gain is [0-10].
- Average path gains are normalized to 0 dB so that the average power input to the AWGN block is 1 W.

The MLSE Equalizer block has the Traceback depth set to 10. Vary this depth to study its effect on Bit Error rate (BER).

The QPSK demodulator accepts an N-by-1 input frame and generates a 2 N -by-1 output frame. This output frame and a traceback depth of 10 results in a delay of 20 bits. The model performs framebased processing on frames that have 100 samples per frame. Due to the frame-based processing, there is a inherent delay of 100 bits in the model. The combined receive delay of 120 is set in the Receive delay parameter of the Error Rate Calculation block, aligning the samples.

The computed BER is displayed. Constellation plots show the constellation before and after equalization.
$B E R=0.033645$



## Adaptive Equalization

This example shows how to a model a communication link with PSK modulation, raised cosine pulse shaping, multipath fading, and adaptive equalization.

The example sets up three equalization scenarios, and calls a separate script to execute the processing loop multiple times for each scenario. Each call corresponds to a transmission block. The pulse shaping and multipath fading channel retain state information from one block to the next. For visualizing the impact of channel fading on adaptive equalizer convergence, the simulation resets the equalizer state every block.

To experiment with different simulation settings, you can edit the example. For instance, you can set the ResetBeforeFiltering property of the equalizer object to 0 , which will cause the equalizer to retain state from one block to the next.

## Transmission Block

Set parameters related to the transmission block which is composed of three parts: training sequence, payload, and tail sequence. All three use the same PSK scheme; the training and tail sequences are used for equalization. We use the default random number generator to ensure the repeatability of the results.

```
Rsym = 1e6; % Symbol rate (Hz)
nTrain = 100; % Number of training symbols
nPayload = 400; % Number of payload symbols
nTail = 20; % Number of tail symbols
% Set random number generator for repeatability
hStream = RandStream.create('mt19937ar', 'seed', 12345);
```


## PSK Modulation

Configure the PSK modulation and demodulation System objects.

```
bitsPerSym = 2; % Number of bits per PSK symbol
M = 2^bitsPerSym; % Modulation order
hPSKMod = comm.PSKModulator(M, ...
    'Phase0ffset',0, ...
    'SymbolMapping','Binary');
hPSKDemod = comm.PSKDemodulator(M, ...
    'Phase0ffset',0, ...
    'SymbolMapping','Binary');
PSKConstellation = constellation(hPSKMod).'; % PSK constellation
```


## Training and Tail Sequences

Generate the training and tail sequences.

```
xTrainData = randi(hStream, [0 M-1], nTrain, 1);
xTailData = randi(hStream, [0 M-1], nTail, 1);
xTrain = step(hPSKMod,xTrainData);
xTail = step(hPSKMod,xTailData);
```


## Transmit and Receive Filters

Configure raised cosine transmit and receive filter System objects. The filters incorporate upsampling and downsampling, respectively.

```
chanFilterSpan = 8; % Filter span in symbols
sampPerSymChan = 4; % Samples per symbol through channels
hTxFilt = comm.RaisedCosineTransmitFilter( ...
    'RolloffFactor',0.25, ...
    'FilterSpanInSymbols',chanFilterSpan, ...
    'OutputSamplesPerSymbol',sampPerSymChan);
hRxFilt = comm.RaisedCosineReceiveFilter( ...
    'RolloffFactor',0.25, ...
    'FilterSpanInSymbols',chanFilterSpan, ...
    'InputSamplesPerSymbol',sampPerSymChan, ...
    'DecimationFactor',sampPerSymChan);
% Calculate the samples per symbol after the receive filter
sampPerSymPostRx = sampPerSymChan/hRxFilt.DecimationFactor;
% Calculate the delay in samples from both channel filters
chanFilterDelay = chanFilterSpan*sampPerSymPostRx;
```


## AWGN Channel

Configure an AWGN channel System object with the NoiseMethod property set to Signal to noise ratio (Es/No) and Es/No set to 20 dB .

```
hAWGNChan = comm.AWGNChannel( ...
    'NoiseMethod','Signal to noise ratio (Es/No)', ...
    'EsNo',20, ...
    'SamplesPerSymbol',sampPerSymChan);
```


## Simulation 1: Linear Equalization for Frequency-Flat Fading

Begin with single-path, frequency-flat fading channel. For this channel, the receiver uses a simple 1tap LMS (least mean square) equalizer, which implements automatic gain and phase control.

The script commadapteqloop.m runs multiple times. Each run corresponds to a transmission block. The equalizer resets its state and weight every transmission block. To retain state from one block to the next, you can set the ResetBeforeFiltering property of the equalizer object to false.

Before the first run, commadapteqloop.m displays the Rayleigh channel System object and the properties of the equalizer object. For each run, a MATLAB figure shows signal processing visualizations. The red circles in the signal constellation plots correspond to symbol errors. In the "Weights" plot, blue and magenta lines correspond to real and imaginary parts, respectively.

```
simName = 'Linear equalization for frequency-flat fading'; % Used to label figure window
% Configure a frequency-flat Rayleigh channel System object with the
% RandomStream property set to 'mt19937ar with seed' for repeatability.
hRayleighChan = comm.RayleighChannel( ...
    'SampleRate',Rsym*sampPerSymChan, ...
    'MaximumDopplerShift',30);
% Configure an adaptive equalizer object
nWeights = 1; % Single weight
stepSize = 0.1; % Step size for LMS algorithm
alg = lms(stepSize); % Adaptive algorithm object
eqObj = lineareq(nWeights,alg,PSKConstellation); % Equalizer object
% Delay in symbols from the equalizer
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;
```

```
% Link simulation
nBlocks = 50; % Number of transmission blocks in simulation
for block = 1:nBlocks
    commadapteqloop;
end
System: comm.RayleighChannel
    Properties:
            SampleRate: 4000000
            PathDelays: 0
        AveragePathGains: 0
        NormalizePathGains: true
        MaximumDopplerShift: 30
        DopplerSpectrum: [1xl struct]
            RandomStream: 'mt19937ar with seed'
                Seed: }7
    PathGainsOutputPort: false
                EqType: 'Linear Equalizer'
                AlgType: 'LMS'
                nWeights: 1
                nSampPerSym: 1
                    RefTap: 1
                SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i -1.0000 + 0.0000i -0.0000 - 1.0000i]
                StepSize: 0.1000
            LeakageFactor: 1
                    Weights: 0
            WeightInputs: 0
    ResetBeforeFiltering: 1
    NumSamplesProcessed: 0
```



## Simulation 2: Linear Equalization for Frequency-Selective Fading

Simulate a three-path, frequency-selective Rayleigh fading channel. The receiver uses an 8 -tap linear RLS (recursive least squares) equalizer with symbol-spaced taps.

```
simName = 'Linear equalization for frequency-selective fading';
% Reset transmit and receive filters
reset(hTxFilt);
reset(hRxFilt);
% Set the Rayleigh channel System object to be frequency-selective
release(hRayleighChan);
hRayleighChan.PathDelays = [0 0.9 1.5]/Rsym;
hRayleighChan.AveragePathGains = [0 -3 -6];
% Configure an adaptive equalizer
nWeights = 8;
forgetFactor = 0.99; % RLS algorithm forgetting factor
alg = rls(forgetFactor); % RLS algorithm object
eqObj = lineareq(nWeights,alg,PSKConstellation);
eqObj.RefTap = 3; %Reference tap
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;
```

```
% Link simulation and store BER values
BERvect = zeros(1,nBlocks);
for block = 1:nBlocks
    commadapteqloop;
    BERvect(block) = BEREq;
end
avgBER2 = mean(BERvect)
System: comm.RayleighChannel
    Properties:
            SampleRate: 4000000
                PathDelays: [0 9e-07 1.5e-06]
            AveragePathGains: [0 -3 -6]
        NormalizePathGains: true
        MaximumDopplerShift: 30
            DopplerSpectrum: [1x1 struct]
                RandomStream: 'mt19937ar with seed'
                    Seed: 73
        PathGainsOutputPort: false
                            EqType: 'Linear Equalizer'
                        AlgType: 'RLS'
                                nWeights: 8
                    nSampPerSym: 1
                            RefTap: 3
                            SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i -1.0000 + 0.0000i -0.0000 - 1.0000i]
                            ForgetFactor: 0.9900
                            InvCorrInit: 0.1000
                InvCorrMatrix: [8x8 double]
                    Weights: [00 0 0 0 0 0 0 0)]
                WeightInputs: [000 0 0 0 0 0 0 0 ]
        ResetBeforeFiltering: 1
        NumSamplesProcessed: 0
avgBER2 =
    3.0000e-04
```



## Simulation 3: Decision feedback Equalization (DFE) for Frequency-Selective Fading

The receiver uses a DFE with a six-tap fractionally spaced forward filter (two samples per symbol) and two feedback weights. The DFE uses the same RLS algorithm as in Simulation 2. The receive filter structure is reconstructed to account for the increased number of samples per symbol.

```
simName = 'Decision feedback equalization (DFE) for frequency-selective fading';
% Reset transmit filter and adjust receive filter decimation factor
reset(hTxFilt);
release(hRxFilt);
hRxFilt.DecimationFactor = 2;
sampPerSymPostRx = sampPerSymChan/hRxFilt.DecimationFactor;
chanFilterDelay = chanFilterSpan*sampPerSymPostRx;
% Reset fading channel
reset(hRayleighChan);
% Configure an adaptive equalizer object
nFwdWeights = 6; % Number of feedforward equalizer weights
nFbkWeights = 2; % Number of feedback filter weights
eqObj = dfe(nFwdWeights, nFbkWeights,alg,PSKConstellation,sampPerSymPostRx);
eqObj.RefTap = 3;
```

```
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;
for block = 1:nBlocks
    commadapteqloop;
    BERvect(block) = BEREq;
end
avgBER3 = mean(BERvect)
System: comm.RayleighChannel
    Properties:
            SampleRate: 4000000
            PathDelays: [0 9e-07 1.5e-06]
            AveragePathGains: [0 0-3 -6]
        NormalizePathGains: true
        MaximumDopplerShift: 30
            DopplerSpectrum: [1x1 struct]
                RandomStream: 'mt19937ar with seed'
                Seed: 73
        PathGainsOutputPort: false
                                    EqType: 'Decision Feedback Equalizer'
                                    AlgType: 'RLS'
                                    nWeights: [6 2]
                                    nSampPerSym: 2
                                    RefTap: 3
                                    SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i-1.0000 + 0.0000i -0.0000 - 1.0000i]
                    ForgetFactor: 0.9900
                    InvCorrInit: 0.1000
            InvCorrMatrix: [8x8 double]
                Weights: [0 0 0 0 0 0 0 0 0 0)]
                WeightInputs: [0 0 0 0 0 0 0 0]
        ResetBeforeFiltering: 1
        NumSamplesProcessed: 0
avgBER3 =
    0
```



## Summary

This example showed the relative performance of linear and decision feedback equalizers in both frequency-flat and frequency-selective fading channels. It showed how a one-tap equalizer is sufficient to compensate for a frequency-flat channel, but that a frequency-selective channel requires an equalizer with multiple taps. Finally, it showed that a decision feedback equalizer is superior to a linear equalizer in a frequency-selective channel.

## Appendix

This example uses the following script and helper functions:

- commadapteqloop.m
- commadapteq_checkvars.m
- commadapteq_graphics.m


## Equalize BSPK Signal

Equalize a BPSK signal using a linear equalizer with an least mean square (LMS) algorithm.
Generate random binary data and apply BPSK modulation.

```
data = randi([0 1],1000,1);
modData = pskmod(data,2);
```

Apply two-tap static fading to the modulated signal.

```
rxSig = conv(modData,[0.5 0.05]);
```

Create an LMS adaptive algorithm object with a step size of 0.06 .

```
alg = lms(0.06);
```

Warning: LMS will be removed in a future release. Use comm.LinearEqualizer or comm.DecisionFeedb
Create a linear equalizer object having 8 taps using the previously created algorithm object. Set the reference tap index to 4.

```
eqlms = lineareq(8,alg);
```

Warning: LINEAREQ will be removed in a future release. Use comm.LinearEqualizer instead. See <a
eqlms.RefTap = 4;

Equalize the received signal, rxSig, while using the first 200 data bits as a training sequence.

```
trSeq = data(1:200);
[eqSig,~,e] = equalize(eqlms,rxSig,trSeq);
```

Warning: EQUALIZE will be removed in a future release. Use comm.LinearEqualizer or comm.Decision
Filter and plot the power of the received (nonequalized) signal. The magnitude of the signal has been attenuated by the channel.

```
rxSigPwr = filter(0.1*ones(10,1),1,abs(rxSig)).^2;
plot(rxSigPwr)
title('Received Signal')
xlabel('Bits')
ylabel('Power (W)')
```



Plot the equalized signal. The signal reaches the intended power level of 1 W .

```
eqSigPwr = filter(0.1*ones(10,1),1,abs(eqSig)).^2;
plot(eqSigPwr)
title('Equalized Signal')
xlabel('Bits')
ylabel('Power (W)')
```

Equalized Signal


Plot the magnitude of the error estimate, e. The error decreases until it is nearly zero after 400 bits.


## Compare RLS and LMS Algorithms

Equalize a QAM signal passed through a frequency-selective fading channel using RLS and LMS algorithms. Compare the performance of the two algorithms.

Specify the modulation order. Generate the corresponding QAM reference constellation.

```
M = 16;
sigConst = qammod(0:M-1,M,'UnitAveragePower',true);
```

Create a frequency-selective static channel having three taps.

```
rchan = comm.RayleighChannel('SampleRate',1000, ...
    'PathDelays',[0 1e-3 2e-3],'AveragePathGains',[0 -3 -6], ...
    'MaximumDopplerShift',0, ...
    'RandomStream','mt19937ar with seed','Seed',73);
```


## RLS Equalizer

Create an RLS equalizer object.

```
eqrls = lineareq(6,rls(0.99,0.1));
eqrls.SigConst = sigConst;
eqrls.ResetBeforeFiltering = 0;
```

Generate and QAM modulate a random training sequence. Pass the sequence through the Rayleigh fading channel. Pass the received signal and the training signal through the equalizer to set the equalizer tap weights.

```
trainData = randi([0 M-1],200,1);
trainSig = qammod(trainData,M,'UnitAveragePower',true);
rxSig = rchan(trainSig);
[~,~,errorSig] = equalize(eqrls,rxSig,trainSig);
```

Plot the magnitude of the error estimate.

```
plot(abs(errorSig))
title('Error Estimate, RLS Equalizer')
xlabel('Symbols')
ylabel('Amplitude')
```

The error is nearly eliminated within 200 symbols.
Transmit a QAM signal through a frequency-selective channel. Equalize the received signal using the previously 'trained' RLS equalizer. Measure the time required to execute the processing loop.

```
tic
for k = 1:20
    data = randi([0 M-1],1000,1); % Random message
    txSig = qammod(data,M,'UnitAveragePower',true);
    % Introduce channel distortion.
    rxSig = rchan(txSig);
    % Equalize the received signal.
    eqSig = equalize(eqrls,rxSig);
```

```
end
rlstime = toc;
```

Plot the constellation diagram of the received and equalized signals.

```
h = scatterplot(rxSig,1,0,'c.');
hold on
scatterplot(eqSig,1,0,'b.',h)
legend('Received Signal','Equalized Signal')
title('RLS Equalizer')
hold off
```

The equalizer removed the effects of the fading channel.

## LMS Equalizer

Repeat the equalization process with an LMS equalizer. Create an LMS equalizer object.

```
eqlms = lineareq(6,lms(0.03));
eqlms.SigConst = sigConst;
eqlms.ResetBeforeFiltering = 0;
```

Train the LMS equalizer.

```
trainData = randi([0 M-1],1000,1);
trainSig = qammod(trainData,M,'UnitAveragePower',true);
rxSig = rchan(trainSig);
[~,~,errorSig] = equalize(eqlms,rxSig,trainSig);
```

Plot the magnitude of the error estimate.

```
plot(abs(errorSig))
title('Error Estimate, LMS Equalizer')
xlabel('Symbols')
ylabel('Amplitude')
```

Training the LMS equalizer requires 1000 symbols.
Transmit a QAM signal through the same frequency-selective channel. Equalize the received signal using the previously 'trained' LMS equalizer. Measure the time required to execute the processing loop.

```
tic
for k = 1:20
    data = randi([0 M-1],1000,1); % Random message
    txSig = qammod(data,M,'UnitAveragePower',true);
    % Introduce channel distortion.
    rxSig = rchan(txSig);
    % Equalize the received signal.
    eqSig = equalize(eqlms,rxSig);
end
lmstime = toc;
```

Plot the constellation diagram of the received and equalized signals.

```
h = scatterplot(rxSig,1,0,'c.');
hold on
```

```
scatterplot(eqSig,1,0,'b.',h)
legend('Received Signal','Equalized Signal')
title('LMS Equalizer')
```

The equalizer removes the effects of the fading channel.
Compare the loop execution time for the two equalizer algorithms.
[rlstime lmstime]
The LMS algorithm is more computationally efficient as it took $50 \%$ of the time to execute the processing loop. However, the training sequence required by the LMS algorithm is 5 times longer.

## System Design

- "Source Coding" on page 8-2
- "Error Detection and Correction" on page 8-13
- "Interleaving" on page 8-115
- "Digital Modulation" on page 8-128
- "Analog Passband Modulation" on page 8-148
- "Phase-Locked Loops" on page 8-153
- "Multiple-Input Multiple-Output (MIMO)" on page 8-156
- "Huffman Coding" on page 8-158
- "Differential Pulse Code Modulation" on page 8-160
- "Compand a Signal" on page 8-163
- "Arithmetic Coding" on page 8-165
- "Quantization" on page 8-167


## Source Coding

```
In this section...
"Represent Partitions" on page 8-2
"Represent Codebooks" on page 8-2
"Determine Which Interval Each Input Is In" on page 8-3
"Optimize Quantization Parameters" on page 8-3
"Differential Pulse Code Modulation" on page 8-4
"Optimize DPCM Parameters" on page 8-6
"Compand a Signal" on page 8-7
"Huffman Coding" on page 8-8
"Arithmetic Coding" on page 8-9
"Quantize a Signal" on page 8-10
```


## Represent Partitions

Scalar quantization is a process that maps all inputs within a specified range to a common value. This process 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 on page $8-2$ and a codebook on page 8-2.

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.


## Represent 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 $[0,1,3]$.

## Determine 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);
```


## Optimize Quantization Parameters

- "Section Overview" on page 8-3
- "Example: Optimizing Quantization Parameters" on page 8-3


## Section Overview

Quantization distorts a signal. You can reduce 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 =
```

$$
0.0148 \quad 0.0024
$$

## Differential Pulse Code Modulation

- "Section Overview" on page 8-4
- "DPCM Terminology" on page 8-4
- "Represent Predictors" on page 8-5
- "Example: DPCM Encoding and Decoding" on page 8-5


## Section Overview

The quantization in the section "Quantize a Signal" on page 8-10 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 "Represent Partitions" on page 8-2 and "Represent Codebooks" on page 8-2, but also a predictor. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form
$y(k)=p(1) x(k-1)+p(2) x(k-2)+\ldots+p(m-1) x(k-m+1)+p(m) x(k-m)$
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.

## Represent Predictors

If the guess for the kth value of the signal $x$, based on earlier values of $x$, is
$y(k)=p(1) x(k-1)+p(2) x(k-2)+\ldots+p(m-1) x(k-m+1)+p(m) \times(k-m)$
then the corresponding predictor vector for toolbox functions is
predictor $=[0, p(1), p(2), p(3), \ldots, p(m-1), p(m)]$

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
    

## Optimize DPCM Parameters

- "Section Overview" on page 8-6
- "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 8-6


## Section Overview

The section "Optimize Quantization Parameters" on page 8-3 describes how to use training data with the $l$ loyds 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

## Compand a Signal

- "Section Overview" on page 8-7
- "Example: $\mu$-Law Compander" on page 8-7


## 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: $\mu$-law and A-law companders. Its reference page lists both compressor laws.

## Example: $\mu$-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 $\mu$-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" on page 8-8
- "Create a Huffman Code Dictionary in MATLAB" on page 8-9
- "Create and Decode a Huffman Code Using MATLAB" on page 8-9


## 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 8-9.

## Create a Huffman Code Dictionary in MATLAB

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 1 s with probability $0.1,2 \mathrm{~s}$ with probability 0.1 , and 3 s 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
```


## Create and Decode a Huffman Code Using MATLAB

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" on page 8-10
- "Represent Arithmetic Coding Parameters" on page 8-10
- "Create and Decode an Arithmetic Code Using MATLAB" on page 8-10


## 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.

## Represent 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];
```


## Create and Decode an Arithmetic Code Using MATLAB

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


## Quantize a Signal

- "Scalar Quantization Example 1" on page 8-10
- "Scalar Quantization Example 2" on page 8-11


## 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])
```



## See Also

## Functions

dpcmdeco|dpcmenco |dpcmopt|lloyds|quantiz

## Error Detection and Correction

In this section...<br>"Cyclic Redundancy Check Codes" on page 8-13<br>"Block Codes" on page 8-16<br>"Convolutional Codes" on page 8-29<br>"Linear Block Codes" on page 8-53<br>"Hamming Codes" on page 8-61<br>"BCH Codes" on page 8-67<br>"Reed-Solomon Codes" on page 8-71<br>"LDPC Codes" on page 8-80<br>"Galois Field Computations" on page 8-80<br>"Galois Fields of Odd Characteristic" on page 8-103

## Cyclic Redundancy Check Codes

- "CRC-Code Features" on page 8-13
- "CRC Non-Direct Algorithm" on page 8-13
- "Example Using CRC Non-Direct Algorithm" on page 8-14
- "CRC Direct Algorithm" on page 8-15
- "Selected Bibliography for CRC Coding" on page 8-15


## CRC-Code Features

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 a communications system detects an error 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 block library. Open the CRC sublibrary by double-clicking on its icon in the Error Detection and Correction library.

Communications Toolbox supports CRC Coding using Simulink blocks, System objects, or MATLAB objects.

For a list of the CRC coding routines the Communications Toolbox supports see Cyclic Redundancy Check Coding.

## CRC Non-Direct Algorithm

The CRC non-direct 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 $T=M^{*} \chi^{r}+C$, since multiplying by $\chi^{r}$ 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 $T=Q^{*} P+R$, 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 [ $\left.\begin{array}{llll}1 & 1 & 0 & 1\end{array}\right]$ 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 $\mathrm{d}(1)$ 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.

## Example Using CRC Non-Direct Algorithm

Suppose the input frame is [ $\left.\begin{array}{lllllll}1 & 1 & 0 & 0 & 1 & 1 & 0\end{array}\right]^{\prime}$, corresponding to the polynomial $M=x^{6}+x^{5}+x^{2}+$ $x$, and the generator polynomial is $\mathrm{P}=x^{3}+x^{2}+1$, of degree $\mathrm{r}=3$. By polynomial division, $\mathrm{M}^{*} \mathrm{x}^{3}=\left(x^{6}\right.$ $\left.+x^{3}+x\right)^{*} \mathrm{P}+x$. The remainder is $\mathrm{R}=x$, so that the checksum is then $\left[\begin{array}{lll}0 & 1 & 0\end{array}\right]$ '. An extra 0 is added on the left to make the checksum have length 3 .

## CRC Direct Algorithm


where
Message Block Input is

## $m_{0}, m_{1}, \ldots, m_{k-1}$

Code Word Output is

$$
c_{0}, c_{1}, \ldots, c_{n-1}=\underset{X}{\neq}, m_{1}, \ldots, m_{k-1}, d_{0}, d_{1}, \ldots, d_{n-k-1}
$$

The initial step of the direct CRC encoding occurs with the three switches in position X. The algorithm feeds $k$ message bits to the encoder. These bits are the first $k$ bits of the code word output. Simultaneously, the algorithm sends $k$ bits to the linear feedback shift register (LFSR). When the system completely feeds the $k$ th message bit to the LFSR, the switches move to position Y. Here, the LFSR contains the mathematical remainder from the polynomial division. These bits are shifted out of the LFSR and they are the remaining bits (checksum) of the code word output.

## 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.

## Block Codes

- "Block-Coding Features" on page 8-16
- "Terminology" on page 8-17
- "Data Formats for Block Coding" on page 8-17
- "Using Block Encoders and Decoders Within a Model" on page 8-19
- "Examples of Block Coding" on page 8-19
- "Notes on Specific Block-Coding Techniques" on page 8-22
- "Shortening, Puncturing, and Erasures" on page 8-24
- "Reed-Solomon Code in Integer Format" on page 8-26
- "Find a Generator Polynomial" on page 8-27
- "Performing Other Block Code Tasks" on page 8-27
- "Selected Bibliography for Block Coding" on page 8-28


## Block-Coding Features

Error-control coding techniques detect, and possibly correct, errors that occur when messages are transmitted in a digital communication system. To accomplish this, the encoder transmits not only the information symbols but also extra redundant symbols. The decoder interprets what it receives, using the redundant symbols to detect and possibly correct whatever errors occurred during transmission. You might use error-control coding if your transmission channel is very noisy or if your data is very sensitive to noise. Depending on the nature of the data or noise, you might choose a specific type of error-control coding.

Block coding is a special case of error-control coding. Block-coding techniques map a fixed number of message symbols to a fixed number of code symbols. A block coder treats each block of data independently and is a memoryless device. Communications Toolbox contains block-coding capabilities by providing Simulink blocks, System objects, and MATLAB functions.

The class of block-coding techniques includes categories shown in the diagram below.


Communications Toolbox supports general linear block codes. It also process cyclic, BCH, Hamming, and Reed-Solomon codes (which are all special kinds of linear block codes). Blocks in the product can encode or decode a message using one of the previously mentioned techniques. The Reed-Solomon and BCH decoders indicate how many errors they detected while decoding. The Reed-Solomon coding blocks also let you decide whether to use symbols or bits as your data.

Note The blocks and functions in Communications Toolbox are designed for error-control codes that use an alphabet having 2 or $2^{\mathrm{m}}$ symbols.

## Communications Toolbox Support Functions

Functions in Communications Toolbox can support simulation blocks by

- Determining characteristics of a technique, such as error-correction capability or possible message lengths
- Performing lower-level computations associated with a technique, such as
- Computing a truth table
- Computing a generator or parity-check matrix
- Converting between generator and parity-check matrices
- Computing a generator polynomial

For more information about error-control coding capabilities, see Block Codes on page 8-16.

## Terminology

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

Each block of $K$ message symbols is encoded into a codeword that consists of $N$ message symbols. $K$ is called the message length, $N$ is called the codeword length, and the code is called an $[N, K]$ code.

## Data Formats for Block Coding

Each message or codeword is an ordered grouping of symbols. Each block in the Block Coding sublibrary processes one word in each time step, as described in the following section, "Binary Format (All Coding Methods)" on page 8-17. Reed-Solomon coding blocks also let you choose between binary and integer data, as described in "Integer Format (Reed-Solomon Only)" on page 818.

## Binary Format (All Coding Methods)

You can structure messages and codewords as binary vector signals, where each vector represents a message word or a codeword. At a given time, the encoder receives an entire message word, encodes it, and outputs the entire codeword. The message and code signals operate over the same sample time.

This example illustrates the encoder receiving a four-bit message and producing a five-bit codeword at time 0 . It repeats this process with a new message at time 1.


For all coding techniques except Reed-Solomon using binary input, the message vector must have length K and the corresponding code vector has length N. For Reed-Solomon codes with binary input, the symbols for the code are binary sequences of length $M$, corresponding to elements of the Galois field $\operatorname{GF}\left(2^{\mathrm{M}}\right)$. In this case, the message vector must have length $\mathrm{M}^{*} \mathrm{~K}$ and the corresponding code vector has length $\mathrm{M}^{*} \mathrm{~N}$. The Binary-Input RS Encoder block and the Binary-Output RS Decoder block use this format for messages and codewords.

If the input to a block-coding block is a frame-based vector, it must be a column vector instead of a row vector.

To produce sample-based messages in the binary format, you can configure the Bernoulli Binary Generator block so that its Probability of a zero parameter is a vector whose length is that of the signal you want to create. To produce frame-based messages in the binary format, you can configure the same block so that its Probability of a zero parameter is a scalar and its Samples per frame parameter is the length of the signal you want to create.

## Using Serial Signals

If you prefer to structure messages and codewords as scalar signals, where several samples jointly form a message word or codeword, you can use the Buffer and Unbuffer blocks. Buffering involves latency and multirate processing. If your model computes error rates, the initial delay in the codingbuffering combination influences the Receive delay parameter in the Error Rate Calculation block.

You can display the sample times of signals in your model. On the Debug tab, expand Information Overlays. In the Sample Time section, select Colors. Alternatively, you can attach Probe blocks to connector lines to help evaluate sample timing, buffering, and delays.

## Integer Format (Reed-Solomon Only)

A message word for an $[\mathrm{N}, \mathrm{K}]$ Reed-Solomon code consists of $\mathrm{M} * \mathrm{~K}$ bits, which you can interpret as K symbols from 0 to $2^{\mathrm{M}}$. The symbols are binary sequences of length M , corresponding to elements of the Galois field GF $\left(2^{\mathrm{M}}\right)$, in descending order of powers. The integer format for Reed-Solomon codes lets you structure messages and codewords as integer signals instead of binary signals. (The input must be a frame-based column vector.)

Note In this context, Simulink expects the first bit to be the most significant bit in the symbol. "First" means the smallest index in a vector or the smallest time for a series of scalars.

The following figure illustrates the equivalence between binary and integer signals for a ReedSolomon encoder. The case for the decoder is similar.


To produce sample-based messages in the integer format, you can configure the Random Integer Generator block so that M-ary number and Initial seed parameters are vectors of the desired length and all entries of the $\mathbf{M}$-ary number vector are $2^{\mathrm{M}}$. To produce frame-based messages in the integer format, you can configure the same block so that its M-ary number and Initial seed parameters are scalars and its Samples per frame parameter is the length of the signal you want to create.

## Using Block Encoders and Decoders Within a Model

Once you have configured the coding blocks, a few tips can help you place them correctly within your model:

- If a block has multiple outputs, the first one is always the stream of coding data.

The Reed-Solomon and BCH blocks have an error counter as a second output.

- Be sure that the signal sizes are appropriate for the mask parameters. For example, if you use the Binary Cyclic Encoder block and set Message length K to 4, the input signal must be a vector of length 4.

You can display the size of signals in your model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions.

## Examples of Block Coding

Example: Reed-Solomon Code in Integer Format
This example uses a Reed-Solomon code in integer format. It illustrates the appropriate vector lengths of the code and message signals for the coding blocks. It also exhibits error correction, using a simple way of introducing errors into each codeword.


Open the model by typing doc_rscoding at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Comm Sources library
- Set M-ary number to 15.
- Set Initial seed to a positive number, randn is chosen here.
- Check the Frame-based outputs check box.
- Set Samples per frame to 5 .
- Integer-Input RS Encoder
- Set Codeword length $\mathbf{N}$ to 15.
- Set Message length K to 5 .
- Gain, in the Simulink Math Operations library
- Set Gain to $[0 ; 0 ; 0 ; 0 ; 0 ;$ ones $(10,1)]$.
- Integer-Output RS Decoder
- Set Codeword length $\mathbf{N}$ to 15.
- Set Message length K to 5 .
- Scope, in the Simulink Sinks library. Get two copies of this block.
- Sum, in the Simulink Math Operations library
- Set List of signs to |-+

Connect the blocks as shown in the preceding figure. On the Simulation tab, in the Simulate section, set Stop time to 500. The Simulate section appears on multiple tabs.

You can display the vector length of signals in your model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions.

The encoder accepts a vector of length 5 (which is K in this case) and produces a vector of length 15 (which is N in this case). The decoder does the opposite.

Running the model produces the following scope images. The plotted error count will vary based on the Initial Seed value used in the Random Integer Generator block. You can adjust the axis range exactly match that of the first scope. Right-click the plot area in the second scope and select
Configuration Properties. On the Display tab, adjust the axes limits.


## Number of Errors Before Correction

The second plot is the number of errors that the decoder detected while trying to recover the message. Often the number is five because the Gain block replaces the first five symbols in each codeword with zeros. However, the number of errors is less than five whenever a correct codeword contains one or more zeros in the first five places.

The first plot is the difference between the original message and the recovered message; since the decoder was able to correct all errors that occurred, each of the five data streams in the plot is zero.

## Notes on Specific Block-Coding Techniques

Although the Block Coding sublibrary is somewhat uniform in its look and feel, the various coding techniques are not identical. This section describes special options and restrictions that apply to parameters and signals for the coding technique categories in this sublibrary. Coding techniques discussed below include - Generic Linear Block code, Cyclic code, Hamming code, BCH code, and Reed-Solomon code.

## Generic Linear Block Codes

Encoding a message using a generic linear block code requires a generator matrix. Decoding the code requires the generator matrix and possibly a truth table. To use the Binary Linear Encoder and Binary Linear Decoder blocks, you must understand the Generator matrix and Error-correction truth table parameters.

Generator Matrix - The process of encoding a message into an [ $\mathrm{N}, \mathrm{K}$ ] linear block code is determined by a K-by-N generator matrix $G$. Specifically, a 1 -by-K message vector $v$ is encoded into the 1 -by-N codeword vector $v G$. If $G$ has the form $\left[I_{\mathrm{k}}, P\right]$ or $\left[P, I_{\mathrm{k}}\right]$, where $P$ is some K -by-( $\mathrm{N}-\mathrm{K}$ ) matrix and $I_{\mathrm{k}}$ is the K-by-K identity matrix, $G$ is said to be in standard form. (Some authors, such as Clark and Cain [2], use the first standard form, while others, such as Lin and Costello [3], use the second.) The linear block-coding blocks in this product require the Generator matrix mask parameter to be in standard form.

Decoding Table - A decoding table tells a decoder how to correct errors that may 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.

The Binary Linear Decoder block allows you to specify a decoding table in the Error-correction truth table parameter. Represent a decoding table as a matrix with N columns and $2^{\text {N-K }}$ rows. Each row gives a correction vector for one received codeword vector.

You can avoid specifying a decoding table explicitly, by setting the Error-correction truth table parameter to 0. When Error-correction truth table is 0, the block computes a decoding table using the syndtable function.

## Cyclic Codes

For cyclic codes, the codeword length $N$ must have the form $2^{\mathrm{M}}-1$, where M is an integer greater than or equal to 3 .

Generator Polynomials - Cyclic codes have special 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 $x^{\mathrm{N}}-1$. Van Lint [5] explains how a generator polynomial determines a cyclic code.

The Binary Cyclic Encoder and Binary Cyclic Decoder blocks allow you to specify a generator polynomial as the second mask parameter, instead of specifying $K$ there. The blocks represent a generator polynomial using a vector that lists the coefficients of the polynomial in order of ascending powers of the variable. You can find generator polynomials for cyclic codes using the cyclpoly function.

If you do not want to specify a generator polynomial, set the second mask parameter to the value of K.

## Hamming Codes

For Hamming codes, the codeword length $N$ must have the form $2^{\mathrm{M}}-1$, where M is an integer greater than or equal to 3 . The message length K must equal $\mathrm{N}-\mathrm{M}$.

Primitive Polynomials - Hamming codes rely on algebraic fields that have $2^{\mathrm{M}}$ elements (or, more generally, $p^{\mathrm{M}}$ elements for a prime number $p$ ). Elements of such fields are named relative to a distinguished element of the field that is called a primitive element. The minimal polynomial of a primitive element is called a primitive polynomial. The Hamming Encoder and Hamming Decoder blocks allow you to specify a primitive polynomial for the finite field that they use for computations. If you want to specify this polynomial, do so in the second mask parameter field. The blocks represent a primitive polynomial using a vector that lists the coefficients of the polynomial in order of ascending powers of the variable. You can find generator polynomials for Galois fields using the gfprimfd function.

If you do not want to specify a primitive polynomial, set the second mask parameter to the value of K .

## BCH Codes

BCH codes are cyclic error-correcting codes that are constructed using finite fields. For these codes, the codeword length N must have the form $2^{\mathrm{M}}-1$, where M is an integer from 3 to 9 . The message length K is restricted to particular values that depend on N . To see which values of K are valid for a given N, see the comm. BCHEncoder System object reference page. No known analytic formula describes the relationship among the codeword length, message length, and error-correction capability for BCH codes.

## Narrow-Sense BCH Codes

The narrow-sense generator polynomial is LCM[m_1(x), $\left.m_{-} 2(x), \ldots, m_{-} 2 t(x)\right]$, where:

- LCM represents the least common multiple,
- mid x ) represents the minimum polynomial corresponding to $\alpha^{\mathrm{i}}, \alpha$ is a root of the default primitive polynomial for the field $\mathrm{GF}(\mathrm{n}+1)$,
- and trepresents the error-correcting capability of the code.


## Reed-Solomon Codes

Reed-Solomon codes are useful for correcting errors that occur in bursts. In the simplest case, the length of codewords in a Reed-Solomon code is of the form $\mathrm{N}=2^{\mathrm{M}}-1$, where the $2^{\mathrm{M}}$ is the number of symbols for the code. The error-correction capability of a Reed-Solomon code is floor ( (N-K)/2), where K is the length of message words. The difference N-K must be even.

It is sometimes convenient to use a shortened Reed-Solomon code in which N is less than $2^{\mathrm{M}}-1$. In this case, the encoder appends $2^{\mathrm{M}}-1-\mathrm{N}$ zero symbols to each message word and codeword. The errorcorrection capability of a shortened Reed-Solomon code is also floor ( ( $\mathrm{N}-\mathrm{K}$ )/2). The Communications Toolbox Reed-Solomon blocks can implement shortened Reed-Solomon codes.

Effect of Nonbinary Symbols - One difference between Reed-Solomon codes and the other codes supported in this product is that Reed-Solomon codes process symbols in GF( $2^{\mathrm{M}}$ ) instead of GF(2). M bits specify each symbol. The nonbinary nature of the Reed-Solomon code symbols causes the ReedSolomon blocks to differ from other coding blocks in these ways:

- You can use the integer format, via the Integer-Input RS Encoder and Integer-Output RS Decoder blocks.
- The binary format expects the vector lengths to be an integer multiple of $\mathrm{M}^{*} \mathrm{~K}$ (not K ) for messages and the same integer $\mathrm{M}^{*} \mathrm{~N}$ (not N ) for codewords.

Error Information - The Reed-Solomon decoding blocks (Binary-Output RS Decoder and IntegerOutput RS Decoder) return error-related information during the simulation. The second output signal indicates the number of errors that the block detected in the input codeword. A -1 in the second output indicates that the block detected more errors than it could correct using the coding scheme.

## Shortening, Puncturing, and Erasures

Many standards utilize punctured codes, and digital receivers can easily output erasures. BCH and RS performance improves significantly in fading channels where the receiver generates erasures.

A punctured codeword has only parity symbols removed, and a shortened codeword has only information symbols removed. A codeword with erasures can have those erasures in either information symbols or parity symbols.

## Reed Solomon Examples with Shortening, Puncturing, and Erasures

In this section, a representative example of Reed Solomon coding with shortening, puncturing, and erasures is built with increasing complexity of error correction.

## Encoder Example with Shortening and Puncturing

The following figure shows a representative example of a $(7,3)$ Reed Solomon encoder with shortening and puncturing.


In this figure, the message source outputs two information symbols, designated by $\mathrm{I}_{1} \mathrm{I}_{2}$. (For a BCH example, the symbols are binary bits.) Because the code is a shortened $(7,3)$ code, a zero must be added ahead of the information symbols, yielding a three-symbol message of $0 \mathrm{I}_{1} \mathrm{I}_{2}$. The modified message sequence is RS encoded, and the added information zero is then removed, which yields a result of $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{2} \mathrm{P}_{3} \mathrm{P}_{4}$. (In this example, the parity bits are at the end of the codeword.)

The puncturing operation is governed by the puncture vector, which, in this case, is 1011 . Within the puncture vector, a 1 means that the symbol is kept, and a 0 means that the symbol is thrown away. In
this example, the puncturing operation removes the second parity symbol, yielding a final vector of $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{3} \mathrm{P}_{4}$.

## Decoder Example with Shortening and Puncturing

The following figure shows how the RS decoder operates on a shortened and punctured codeword.


This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a $(5,2)$ codeword, because it has been shortened from a $(7,3)$ codeword by one symbol, and one symbol has also been punctured.

As a first step, the decoder adds an erasure, designated by E, in the second parity position of the codeword. This corresponds to the puncture vector 1011. Adding a zero accounts for shortening, in the same way as shown in the preceding figure. The single erasure does not exceed the erasurecorrecting capability of the code, which can correct four erasures. The decoding operation results in the three-symbol message $\mathrm{DI}_{1} \mathrm{I}_{2}$. The first symbol is truncated, as in the preceding figure, yielding a final output of $\mathrm{I}_{1} \mathrm{I}_{2}$.

## Decoder Example with Shortening, Puncturing, and Erasures

The following figure shows the decoder operating on the punctured, shortened codeword, while also correcting erasures generated by the receiver.


In this figure, demodulator receives the $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{3} \mathrm{P}_{4}$ vector that the encoder sent. The demodulator declares that two of the five received symbols are unreliable enough to be erased, such that symbols 2 and 5 are deemed to be erasures. The 01001 vector, provided by an external source, indicates these erasures. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered.

The decoder blocks receive the codeword and the erasure vector, and perform the erasures indicated by the vector 01001. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered. The resulting codeword vector is $\mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{P}_{3} \mathrm{E}$, where E is an erasure symbol.

The codeword is then depunctured, according to the puncture vector used in the encoding operation (i.e., 1011). Thus, an erasure symbol is inserted between $P_{1}$ and $P_{3}$, yielding a codeword vector of $\mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{EP}_{3} \mathrm{E}$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0 \mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{EP}_{3} \mathrm{E}$, such that a $(7,3)$ codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of $\mathrm{DI}_{1} \mathrm{I}_{2}$ (where D refers to a dummy symbol). Finally, the removal of the D symbol from the message vector accounts for the shortening and yields the original $I_{1} \mathrm{I}_{2}$ vector.

For additional information, see the "Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink" on page 11-17 example.

## Reed-Solomon Code in Integer Format

To open an example model that uses a Reed-Solomon code in integer format, type doc_rscoding at the MATLAB command line. For more information about the model, see "Example: Reed-Solomon Code in Integer Format" on page 8-19

## Find 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+\mp@subsup{x}{}{\wedge}8+\mp@subsup{x}{}{\wedge}5+\mp@subsup{x}{}{\wedge}4+\mp@subsup{x}{}{\wedge}2+x+1
genpolyRS = rsgenpoly(15,5)
```

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

```
genpolyCyclic =
    \(\begin{array}{lllllllllll}1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1\end{array}\)
genpolyBCH \(=\) GF(2) array.
Array elements =
\begin{tabular}{lllllllllll}
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1
\end{tabular}
genpolyRS \(=G F\left(2^{\wedge} 4\right)\) array. Primitive polynomial \(=D^{\wedge} 4+D+1\) (19 decimal)
Array elements =
\begin{tabular}{lllllllllll}
1 & 4 & 8 & 10 & 12 & 9 & 4 & 2 & 12 & 2 & 7
\end{tabular}
```

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 8-83 and "Polynomials over Galois Fields" on page 8-97.


## 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 ( $\mathrm{X}-\mathrm{A}^{\mathrm{b}}$ ) $\left(\mathrm{X}-\mathrm{A}^{\mathrm{b}}\right.$ $\left.{ }^{+1}\right) \ldots\left(\mathrm{X}-\mathrm{A}^{\mathrm{b}+2 t-1}\right)$, 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.

## 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.

## - Error Correction Versus Error Detection for Linear Block Codes

You can use a linear block code to detect $d_{\min }-1$ errors or to correct $t=\left[\frac{1}{2}\left(d_{\min }-1\right)\right]$ 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 $2^{\wedge} m-1$, 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 Codes

- "Convolutional Code Features" on page 8-29
- "Polynomial Description of a Convolutional Code" on page 8-30
- "Trellis Description of a Convolutional Code" on page 8-32
- "Create and Decode Convolutional Codes" on page 8-35
- "Design a Rate-2/3 Feedforward Encoder Using MATLAB" on page 8-42
- "Design a Rate $2 / 3$ Feedforward Encoder Using Simulink" on page 8-43
- "Puncture a Convolutional Code Using MATLAB" on page 8-46
- "Implement a Systematic Encoder with Feedback Using Simulink" on page 8-46
- "Soft-Decision Decoding" on page 8-47
- "Tailbiting Encoding Using Feedback Encoders" on page 8-52
- "Selected Bibliography for Convolutional Coding" on page 8-53


## Convolutional Code Features

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.

Communications Toolbox provides convolutional coding capabilities as Simulink blocks, System objects, and MATLAB functions. This product supports feedforward and 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.

The product also includes an a posteriori probability decoder, which can be used for soft output decoding of convolutional codes.

For background information about convolutional coding, see the works listed in Selected Bibliography for Convolutional Coding on page 8-53.

## Block Parameters for Convolutional Coding

To process convolutional codes, use the Convolutional Encoder, Viterbi Decoder, and/or APP Decoder blocks in the Convolutional sublibrary. If a mask parameter is required in both the encoder and the decoder, use the same value in both blocks.

The blocks in the Convolutional sublibrary assume that you use one of two different representations of a convolutional encoder:

- If you design your encoder using a diagram with shift registers and modulo-2 adders, you can compute the code generator polynomial matrix and subsequently use the poly2trellis function (in Communications Toolbox) to generate the corresponding trellis structure mask parameter automatically. For an example, see "Design a Rate 2/3 Feedforward Encoder Using Simulink" on page 8-43.
- If you design your encoder using a trellis diagram, you can construct the trellis structure in MATLAB and use it as the mask parameter.

For more information about these representations, see Polynomial Description of a Convolutional Code on page 8-30 and Trellis Description of a Convolutional Code on page 8-32.

## Using the Polynomial Description in Blocks

To use the polynomial description with the Convolutional Encoder, Viterbi Decoder, or APP Decoder blocks, use the utility function poly2trellis from Communications Toolbox. This function accepts a polynomial description and converts it into a trellis description. For example, the following command computes the trellis description of an encoder whose constraint length is 5 and whose generator polynomials are 35 and 31:
trellis = poly2trellis(5,[35 31]);
To use this encoder with one of the convolutional-coding blocks, simply place a poly2trellis command such as

```
poly2trellis(5,[35 31]);
```

in the Trellis structure parameter field.

## Polynomial Description of a Convolutional Code

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 on page 8-30
- Generator polynomials on page 8-31
- Feedback connection polynomials on page 8-31 (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 8-31 below for details.

In other situations, you can determine the ( $\mathrm{i}, \mathrm{j}$ ) 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 001101010 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 8-28, 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.

## Use Trellis Structure for Rate 1/2 Feedback Convolutional Encoder

Create a trellis structure to represent the rate $1 / 2$ systematic convolutional encoder with feedback shown in this diagram.


This encoder has 5 for its constraint length, [37 33] as its generator polynomial matrix, and 37 for its feedback connection polynomial.

The first generator polynomial is octal 37. The second generator polynomial is octal 33. The feedback polynomial is octal 37. The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits.

The binary vector [1 1011111$]$ represents octal 37 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 10111 ] represents octal 33 and corresponds to the lower row of binary digits in the diagram. These binary digits indicate connections from the outputs of the registers to the two adders in the diagram. The initial 1 corresponds to the input bit.

Convert the polynomial to a trellis structure by using the poly2trellis function. When used with a feedback polynomial, poly2trellis makes a feedback connection to the input of the trellis.

```
trellis = poly2trellis(5,[37 33],37)
trellis = struct with fields:
    numInputSymbols: 2
    numOutputSymbols: 4
            numStates: 16
        nextStates: [16x2 double]
            outputs: [16x2 double]
```

Generate random binary data. Convolutionally encode the data by using the specified trellis structure. Decode the coded data by using the Viterbi algorithm with the specified trellis structure, 34 for its traceback depth, truncated operation mode, and hard decisions.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
tbdepth = 34; % Traceback depth for Viterbi decoder
decodedData = vitdec(codedData,trellis,tbdepth,'trunc','hard');
```

Verify the decoded data has zero bit errors.

```
biterr(data,decodedData)
ans = 0
```


## 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 Code on page 8-30.
trellis = poly2trellis(3,[67]);
The MATLAB structure trellis is a suitable input argument for convenc and vitdec.

## Trellis Description of a Convolutional Code

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 on page 8-33 trellises in MATLAB, and gives an example of a MATLAB trellis on page 8-34.

The figure below depicts a trellis for the convolutional encoder from the previous section. The encoder has four states (numbered in binary from 00 to 11), a one-bit input, and a two-bit output. (The ratio of input bits to output bits makes this encoder a rate-1/2 encoder.) Each solid arrow shows how the encoder changes its state if the current input is zero, and each dashed arrow shows how the encoder changes its state if the current input is one. The octal numbers above each arrow indicate the current output of the encoder.


As an example of interpreting this trellis diagram, if the encoder is in the 10 state and receives an input of zero, 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 <br> encoder: $2^{\mathrm{k}}$ |
| numOutputsymbols | Scalar | Number of output symbols from <br> the encoder: $2^{\mathrm{n}}$ |
| numStates | Scalar | Number of states in the encoder |
| nextStates | numStates-by-2 ${ }^{\mathrm{k}}$ matrix | Next states for all combinations <br> of current state and current <br> input |
| outputs | numStates-by-2 ${ }^{\mathrm{k}}$ matrix | Outputs (in octal) for all <br> combinations of current state <br> 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 $\mathrm{j}-1$. 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 $\{0, \ldots, 0,1\}$. 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 $\mathrm{i}-1$ and the input bits have decimal representation $\mathrm{j}-1$. 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,'num0utputSymbols',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. For more information, see Polynomial Description of a Convolutional Code on page 8-30.

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

## Example: A MATLAB Trellis Structure

Consider the trellis shown below.


To build a trellis structure that describes it, use the command below.

```
trellis = struct('numInputSymbols',2,'numOutputSymbols',4,...
'numStates',4,'nextStates',[0 2;0 2;1 3;1 3],...
'outputs',[0 3;1 2;3 0;2 1]);
```

The number of input symbols is 2 because the trellis diagram has two types of input path: the solid arrow and the dashed arrow. The number of output symbols is 4 because the numbers above the arrows can be either $0,1,2$, or 3 . The number of states is 4 because there are four bullets on the left side of the trellis diagram (equivalently, four on the right side). To compute the matrix of next states, create a matrix whose rows correspond to the four current states on the left side of the trellis, whose columns correspond to the inputs of 0 and 1 , and whose elements give the next states at the end of the arrows on the right side of the trellis. To compute the matrix of outputs, create a matrix whose rows and columns are as in the next states matrix, but whose elements give the octal outputs shown above the arrows in the trellis.

## Create and Decode 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.
Define a trellis.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]);
```

Encode a vector of ones.

```
x = ones(100,1);
code = convenc(x,t);
```

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.

Define a trellis.
$\mathrm{t}=$ poly2trellis([4 3],[4 5 17;7 4 2]);
Encode a vector of ones.
code = convenc(ones(100,1),t);
Set the traceback length for decoding and decode using vitdec.

```
tb = 2;
decoded = vitdec(code,t,tb,'trunc','hard');
```

Verify that the decoded data is a vector of 100 ones.

```
isequal(decoded,ones(100,1))
ans = logical
    1
```


## 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 $2^{\wedge} \mathrm{nsdec}-1$.

An input of 0 represents the most confident 0 , while an input of $2^{\wedge}$ 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 |

## Implement Soft-Decision Decoding Using MATLAB

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 softdecision decoding, the example uses quantiz to map the noisy data values to appropriate decisionvalue 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).

```
s = RandStream.create('mt19937ar', 'seed',94384);
prevStream = RandStream.setGlobalStream(s);
msg = randi([0 1],4000,1); % Random data
t = poly2trellis(7,[171 133]); % Define trellis.
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(t);
% Create an AWGNChannel System object.
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
```

```
    'SNR', 6);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(t, 'InputFormat', 'Soft', ...
    'SoftInputWordLength', 3, 'TracebackDepth', 48, ...
    'TerminationMethod', 'Continuous');
% Create a ErrorRate Calculator System object. Account for the receive
% delay caused by the traceback length of the viterbi decoder.
hErrorCalc = comm.ErrorRate('ReceiveDelay', 48);
ber = zeros(3,1); % Store BER values
code = step(hConvEnc,msg); % Encode the data.
hChan.SignalPower = (code'*code)/length(code);
ncode = step(hChan,code); % 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 = step(hVitDec,qcode); % Decode.
% Compute bit error rate.
ber = step(hErrorCalc, msg, decoded);
ratio = ber(1)
number = ber(2)
RandStream.setGlobalStream(prevStream);
```

The output is below.
number $=$
5
ratio =
0.0013

## Implement Soft-Decision Decoding Using Simulink

This example creates a rate $1 / 2$ convolutional code. It uses a quantizer and the Viterbi Decoder block to perform soft-decision decoding. To open the model, enter doc_softdecision at the MATLAB command line. For a description of the model, see Overview of the Simulation on page 8-47.

## Defining the Convolutional Code

The feedforward convolutional encoder in this example is depicted below.


The encoder's constraint length is a scalar since the encoder has one input. The value of the constraint length is the number of bits stored in the shift register, including the current input. There are six memory registers, and the current input is one bit. Thus the constraint length of the code is 7 .

The code generator is a 1-by-2 matrix of octal numbers because the encoder has one input and two outputs. The first element in the matrix indicates which input values contribute to the first output, and the second element in the matrix indicates which input values contribute to the second output.

For example, the first output in the encoder diagram is the modulo-2 sum of the rightmost and the four leftmost elements in the diagram's array of input values. The seven-digit binary number 1111001 captures this information, and is equivalent to the octal number 171. The octal number 171 thus becomes the first entry of the code generator matrix. Here, each triplet of bits uses the leftmost bit as the most significant bit. The second output corresponds to the binary number 1011011, which is equivalent to the octal number 133. The code generator is therefore [171 133].

The Trellis structure parameter in the Convolutional Encoder block tells the block which code to use when processing data. In this case, the poly2trellis function, in Communications Toolbox, converts the constraint length and the pair of octal numbers into a valid trellis structure.

While the message data entering the Convolutional Encoder block is a scalar bit stream, the encoded data leaving the block is a stream of binary vectors of length 2 .

## Mapping the Received Data

The received data, that is, the output of the AWGN Channel block, consists of complex numbers that are close to -1 and 1. In order to reconstruct the original binary message, the receiver part of the model must decode the convolutional code. The Viterbi Decoder block in this model expects its input data to be integers between 0 and 7. The demodulator, a custom subsystem in this model, transforms the received data into a format that the Viterbi Decoder block can interpret properly. More specifically, the demodulator subsystem

- Converts the received data signal to a real signal by removing its imaginary part. It is reasonable to assume that the imaginary part of the received data does not contain essential information, because the imaginary part of the transmitted data is zero (ignoring small roundoff errors) and because the channel noise is not very powerful.
- Normalizes the received data by dividing by the standard deviation of the noise estimate and then multiplying by -1 .
- Quantizes the normalized data using three bits.

The combination of this mapping and the Viterbi Decoder block's decision mapping reverses the BPSK modulation that the BPSK Modulator Baseband block performs on the transmitting side of this model. To examine the demodulator subsystem in more detail, double-click the icon labeled SoftOutput BPSK Demodulator.

## Decoding the Convolutional Code

After the received data is properly mapped to length-2 vectors of 3-bit decision values, the Viterbi Decoder block decodes it. The block uses a soft-decision algorithm with $2^{3}$ different input values because the Decision type parameter is Soft Decision and the Number of soft decision bits parameter is 3 .

## Soft-Decision Interpretation of Data

When the Decision type parameter is set to Soft Decision, the Viterbi Decoder block requires input values between 0 and $2^{\text {b }}-1$, where $b$ is the Number of soft decision bits parameter. The block
interprets 0 as the most confident decision that the codeword bit is a 0 and interprets $2^{\mathrm{b}}-1$ as the most confident decision that the codeword bit is a 1 . The values in between these extremes represent less confident decisions. The following table lists the interpretations of the eight possible input values for this example.

| Decision 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 |

## Traceback and Decoding Delay

The traceback depth influences the decoding delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

- For the continuous operating mode, the decoding delay is equal to the number of traceback depth symbols.
- For the truncated or terminated operating mode, the decoding delay is zero. In this case, the traceback depth must be less than or equal to the number of symbols in each input.


## Traceback Depth Estimate

As a general estimate, a typical traceback depth value is approximately two to three times (ConstraintLength - 1) / (1-coderate). The constraint length of the code, ConstraintLength, is equal to $(\log 2($ trellis.numStates $)+1)$. The coderate is equal to $(K / N) \times($ length $($ PuncturePattern $) /$ sum(PuncturePattern).
$K$ is the number of input symbols, $N$ is the number of output symbols, and PuncturePattern is the puncture pattern vector.

For example, applying this general estimate, results in these approximate traceback depths.

- A rate $1 / 2$ code has a traceback depth of 5 (ConstraintLength - 1 ).
- A rate $2 / 3$ code has a traceback depth of 7.5(ConstraintLength -1 ).
- A rate $3 / 4$ code has a traceback depth of 10 (ConstraintLength -1 ).
- A rate $5 / 6$ code has a traceback depth of 15 (ConstraintLength -1 ).

The Traceback depth parameter in the Viterbi Decoder block represents the length of the decoding delay. Some hardware implementations offer options of 48 and 96 . This example chooses 48 because that is closer to the estimated target for a rate $1 / 2$ code with a constraint length of 7 .

## Delay in Received Data

The Receive delay parameter of the Error Rate Calculation block is nonzero because a given message bit and its corresponding recovered bit are separated in time by a nonzero amount of
simulation time. The Receive delay parameter tells the block which elements of its input signals to compare when checking for errors.

In this case, the Receive delay value is equal to the Traceback depth value (48).

## Comparing Simulation Results with Theoretical Results

This section describes how to compare the bit error rate in this simulation with the bit error rate that would theoretically result from unquantized decoding. The process includes these steps

## - Computing Theoretical Bounds for the Bit Error Rate

To calculate theoretical bounds for the bit error rate $P_{\mathrm{b}}$ of the convolutional code in this model, you can use this estimate based on unquantized-decision decoding:

$$
P_{b}<\sum_{d=f}^{\infty} c_{d} P_{d}
$$

In this estimate, $c_{\mathrm{d}}$ is the sum of bit errors for error events of distance $d$, and $f$ is the free distance of the code. The quantity $P_{\mathrm{d}}$ is the pairwise error probability, given by

$$
P_{d}=\frac{1}{2} \operatorname{erfc}\left(\sqrt{d R \frac{E_{b}}{N_{0}}}\right)
$$

where $R$ is the code rate of $1 / 2$, and erfc is the MATLAB complementary error function, defined by

$$
\operatorname{erfc}(x)=\frac{2}{\sqrt{\Pi}} \int_{x}^{\infty} e^{-t^{2}} d t
$$

Values for the coefficients $c_{\mathrm{d}}$ and the free distance $f$ are in published articles such as Frenger, P., P. Orten, and T. Ottosson, "Convolution Codes with Optimum Distance Spectrum," IEEE Communications Letters, vol. 3, pp. 317-319, November 1999. [3]. The free distance for this code is $f=10$.

The following commands calculate the values of $P_{\mathrm{b}}$ for $E_{\mathrm{b}} / N_{0}$ values in the range from 1 to 4 , in increments of 0.5 :

```
EbNoVec = [1:0.5:4.0];
R = 1/2;
% Errs is the vector of sums of bit errors for
% error events at distance d, for d from 10 to 29.
Errs = [36 0 211 0 1404 0 11633 0 77433 0 502690 0,...
    3322763 0 21292910 0 134365911 0 843425871 0];
% P is the matrix of pairwise error probilities, for
% Eb/No values in EbNoVec and d from 10 to 29.
P = zeros(20,7); % Initialize.
for d = 10:29
    P(d-9,:) = (1/2)*erfc(sqrt(d*R*10.^(EbNoVec/10)));
end
% Bounds is the vector of upper bounds for the bit error
% rate, for Eb/No values in EbNoVec.
Bounds = Errs*P;
```


## - Simulating Multiple Times to Collect Bit Error Rates

You can efficiently vary the simulation parameters by using the sim function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB , in increments of 0.5 dB . It collects all bit error rates from these simulations in the matrix BERVec. It also plots the bit error rates in a figure window along with the theoretical bounds computed in the preceding code fragment.

Note First open the model by clicking here in the MATLAB Help browser. Then execute these commands, which might take a few minutes.

```
% Plot theoretical bounds and set up figure.
figure;
semilogy(EbNoVec,Bounds,'bo',1,NaN,'r*');
xlabel('Eb/No (dB)'); ylabel('Bit Error Rate');
title('Bit Error Rate (BER)');
legend('Theoretical bound on BER','Actual BER');
axis([1 4 le-5 1]);
hold on;
BERVec = [];
% Make the noise level variable.
set param('doc softdecision/AWGN Channel',...
    'EsNodB','EbNodB+10*log10(1/2)');
% Simulate multiple times.
for n = 1:length(EbNoVec)
    EbNodB = EbNoVec(n);
    sim('doc softdecision',5000000);
    BERVec(n,:) = BER_Data;
    semilogy(EbNoVec(n),BERVec(n,1),'r*'); % Plot point.
    drawnow;
end
hold off;
```

Note The estimate for $P_{\mathrm{b}}$ assumes that the decoder uses unquantized data, that is, an infinitely fine quantization. By contrast, the simulation in this example uses 8 -level (3-bit) quantization. Because of this quantization, the simulated bit error rate is not quite as low as the bound when the signal-to-noise ratio is high.

The plot of bit error rate against signal-to-noise ratio follows. The locations of your actual BER points might vary because the simulation involves random numbers.


Design a Rate-2/3 Feedforward Encoder Using MATLAB
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 [54].

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 [23 350; 0513 ].

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 = randi([0 1],2*len,1); % Random binary message of 2-bit symbols
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Trellis
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(trel);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(trel, 'InputFormat', 'hard', ...
    'TracebackDepth', 34, 'TerminationMethod', 'Continuous');
% Create a ErrorRate Calculator System object. Since each symbol represents
% two bits, the receive delay for this object is twice the traceback length
% of the viterbi decoder.
hErrorCalc = comm.ErrorRate('ReceiveDelay', 68);
ber = zeros(3,1); % Store BER values
code = step(hConvEnc,msg); % Encode the message.
ncode = rem(code + randerr(3*len,1,[0 1;.96 .04]),2); % Add noise.
decoded = step(hVitDec, ncode); % Decode.
ber = step(hErrorCalc, msg, decoded);
```

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 0 s , while subsequent elements represent the decoded messages.

## Design a Rate 2/3 Feedforward Encoder Using Simulink

This example uses the rate $2 / 3$ feedforward convolutional encoder depicted in the following figure. The description explains how to determine the coding blocks' parameters from a schematic of a rate 2/3 feedforward encoder. This example also illustrates the use of the Error Rate Calculation block with a receive delay.


## How to Determine Coding Parameters

The Convolutional Encoder and Viterbi Decoder blocks can implement this code if their parameters have the appropriate values.

The encoder's constraint length is a vector of length 2 since 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 [54].

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, notice that the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13 . The full value of the code generator matrix is [27 33 0; 05 13].

To use the constraint length and code generator parameters in the Convolutional Encoder and Viterbi Decoder blocks, use the poly2trellis function to convert those parameters into a trellis structure.

## How to Simulate the Encoder

The following model simulates this encoder.


To open the completed model, enter doc_convcoding at the MATLAB command line. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Comm Sources library
- Set Probability of a zero to . 5 .
- Set Initial seed to any positive integer scalar, preferably the output of the randn function.
- Set Sample time to . 5.
- Check the Frame-based outputs check box.
- Set Samples per frame to 2.
- Convolutional Encoder
- Set Trellis structure to poly2trellis ([5 4],[23 350 ; 0 ( 5 13]).
- Binary Symmetric Channel, in the Channels library
- Set Error probability to 0.02.
- Set Initial seed to any positive integer scalar, preferably the output of the randn function.
- Clear the Output error vector check box.
- Viterbi Decoder
- Set Trellis structure to poly2trellis([5 4],[23 35 0; 0 5 13]).
- Set Decision type to Hard decision.
- Error Rate Calculation, in the Comm Sinks library
- Set Receive delay to 68.
- Set Output data to Port.
- Check the Stop simulation check box.
- Set Target number of errors to 100.
- Display, in the Simulink Sinks library
- Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as shown in the preceding figure. On the Simulation tab, in the Simulate section, set Stop time to inf. The Simulate section appears on multiple tabs.

## Notes on the Model

You can display the matrix size of signals in your model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions.

The encoder accepts a 2-by-1 column vector and produces a 3-by-1 column vector, while the decoder does the opposite. The Samples per frame parameter in the Bernoulli Binary Generator block is 2 because the block must generate a message word of length 2 .

The Receive delay parameter in the Error Rate Calculation block is 68, which is the vector length (2) of the recovered message times the Traceback depth value (34) in the Viterbi Decoder block. If you examine the transmitted and received signals as matrices in the MATLAB workspace, you see that the first 34 rows of the recovered message consist of zeros, while subsequent rows are the decoded messages. Thus the delay in the received signal is 34 vectors of length 2 , or 68 samples.

Running the model produces display output consisting of three numbers: the error rate, the total number of errors, and the total number of comparisons that the Error Rate Calculation block makes during the simulation. (The first two numbers vary depending on your Initial seed values in the Bernoulli Binary Generator and Binary Symmetric Channel blocks.) The simulation stops after 100
errors occur, because Target number of errors is set to 100 in the Error Rate Calculation block. The error rate is much less than 0.02, the Error probability in the Binary Symmetric Channel block.

## Puncture a Convolutional Code Using MATLAB

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 11001 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.
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(t, ...
        'PuncturePatternSource', 'Property', ...
        'PuncturePattern', [1;1;1;0;0;1]);
% Create an AWGNChannel System object.
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
    'SNR', 3);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(t, 'InputFormat', 'Unquantized', ...
    'TracebackDepth', 96, 'TerminationMethod', 'Truncated', ...
    'PuncturePatternSource', 'Property', ...
    'PuncturePattern', [1;1;1;0;0;1]);
% Create a ErrorRate Calculator System object.
hErrorCalc = comm.ErrorRate;
berP = zeros(3,1); berPE = berP; % Store BER values
punctcode = step(hConvEnc,msg); % Length is (2*len)*2/3.
tcode = 1-2*punctcode; % Map "0" bit to 1 and "1" bit to -1
hChan.SignalPower = (tcode'*tcode)/length(tcode);
ncode = step(hChan,tcode); % Add noise.
% Decode the punctured code
decoded = step(hVitDec,ncode); % Decode.
berP = step(hErrorCalc, msg, decoded);% Bit error rate
% Erase the least reliable 100 symbols, then decode
release(hVitDec); reset(hErrorCalc)
hVitDec.ErasuresInputPort = true;
[dummy idx] = sort(abs(ncode));
erasures = zeros(size(ncode)); erasures(idx(1:100)) = 1;
decoded = step(hVitDec,ncode, erasures); % Decode.
berPE = step(hErrorCalc, msg, decoded);% Bit error rate
fprintf('Number of errors with puncturing: %d\n', berP(2))
fprintf('Number of errors with puncturing and erasures: %d\n', berPE(2))
```


## Implement a Systematic Encoder with Feedback Using Simulink

This section explains how to use the Convolutional Encoder block to implement a systematic encoder with feedback. A code is systematic if the actual message words appear as part of the codewords. The following diagram shows an example of a systematic encoder.


To implement this encoder, set the Trellis structure parameter in the Convolutional Encoder block to poly2trellis(5, [37 33], 37). This setting corresponds to

- Constraint length: 5
- Generator polynomial pair: [37 33]
- Feedback polynomial: 37

The feedback polynomial is represented by the binary vector [1411117], corresponding to the upper row of binary digits. 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.

To implement a systematic code, set the first generator polynomial to be the same as the feedback polynomial in the Trellis structure parameter of the Convolutional Encoder block. In this example, both polynomials have the octal representation 37.

The second generator polynomial is represented by the binary vector [1 100111 ], corresponding to the lower row of binary digits. The octal number corresponding to the binary number 11011 is 33.

For more information on setting the mask parameters for the Convolutional Encoder block, see Polynomial Description of a Convolutional Code on page 8-30.

## Soft-Decision Decoding

This example creates a rate $1 / 2$ convolutional code. It uses a quantizer and the Viterbi Decoder block to perform soft-decision decoding. This description covers these topics:

- "Overview of the Simulation" on page 8-47
- "Defining the Convolutional Code" on page 8-48
- "Mapping the Received Data" on page 8-49
- "Decoding the Convolutional Code" on page 8-49
- "Delay in Received Data" on page 8-50
- "Comparing Simulation Results with Theoretical Results" on page 8-50


## Overview of the Simulation

The model is in the following figure. To open the model, enter doc_softdecision at the MATLAB command line. The simulation creates a random binary message signal, encodes the message into a convolutional code, modulates the code using the binary phase shift keying (BPSK) technique, and adds white Gaussian noise to the modulated data in order to simulate a noisy channel. Then, the simulation prepares the received data for the decoding block and decodes. Finally, the simulation
compares the decoded information with the original message signal in order to compute the bit error rate. The Convolutional encoder is configured as a rate $1 / 2$ encoder. For every 2 bits, the encoder adds another 2 redundant bits. To accommodate this, and add the correct amount of noise, the Eb/No (dB) parameter of the AWGN block is in effect halved by subtracting $10 * \log 10(2)$. The simulation ends after processing 100 bit errors or $10^{7}$ message bits, whichever comes first.


## Defining the Convolutional Code

The feedforward convolutional encoder in this example is depicted below.


The encoder's constraint length is a scalar since the encoder has one input. The value of the constraint length is the number of bits stored in the shift register, including the current input. There are six memory registers, and the current input is one bit. Thus the constraint length of the code is 7 .

The code generator is a 1-by-2 matrix of octal numbers because the encoder has one input and two outputs. The first element in the matrix indicates which input values contribute to the first output, and the second element in the matrix indicates which input values contribute to the second output.

For example, the first output in the encoder diagram is the modulo-2 sum of the rightmost and the four leftmost elements in the diagram's array of input values. The seven-digit binary number 1111001 captures this information, and is equivalent to the octal number 171. The octal number 171 thus becomes the first entry of the code generator matrix. Here, each triplet of bits uses the leftmost bit as
the most significant bit. The second output corresponds to the binary number 1011011, which is equivalent to the octal number 133. The code generator is therefore [171 133].

The Trellis structure parameter in the Convolutional Encoder block tells the block which code to use when processing data. In this case, the poly2trellis function, in Communications Toolbox, converts the constraint length and the pair of octal numbers into a valid trellis structure.

While the message data entering the Convolutional Encoder block is a scalar bit stream, the encoded data leaving the block is a stream of binary vectors of length 2 .

## Mapping the Received Data

The received data, that is, the output of the AWGN Channel block, consists of complex numbers that are close to -1 and 1. In order to reconstruct the original binary message, the receiver part of the model must decode the convolutional code. The Viterbi Decoder block in this model expects its input data to be integers between 0 and 7. The demodulator, a custom subsystem in this model, transforms the received data into a format that the Viterbi Decoder block can interpret properly. More specifically, the demodulator subsystem

- Converts the received data signal to a real signal by removing its imaginary part. It is reasonable to assume that the imaginary part of the received data does not contain essential information, because the imaginary part of the transmitted data is zero (ignoring small roundoff errors) and because the channel noise is not very powerful.
- Normalizes the received data by dividing by the standard deviation of the noise estimate and then multiplying by -1 .
- Quantizes the normalized data using three bits.

The combination of this mapping and the Viterbi Decoder block's decision mapping reverses the BPSK modulation that the BPSK Modulator Baseband block performs on the transmitting side of this model. To examine the demodulator subsystem in more detail, double-click the icon labeled SoftOutput BPSK Demodulator.

## Decoding the Convolutional Code

After the received data is properly mapped to length-2 vectors of 3-bit decision values, the Viterbi Decoder block decodes it. The block uses a soft-decision algorithm with $2^{3}$ different input values because the Decision type parameter is Soft Decision and the Number of soft decision bits parameter is 3 .

## Soft-Decision Interpretation of Data

When the Decision type parameter is set to Soft Decision, the Viterbi Decoder block requires input values between 0 and $2^{\text {b }}-1$, where $b$ is the Number of soft decision bits parameter. The block interprets 0 as the most confident decision that the codeword bit is a 0 and interprets $2^{\mathrm{b}}-1$ as the most confident decision that the codeword bit is a 1 . The values in between these extremes represent less confident decisions. The following table lists the interpretations of the eight possible input values for this example.

| Decision Value | Interpretation |
| :--- | :--- |
| 0 | Most confident 0 |
| 1 | Second most confident 0 |
| 2 | Third most confident 0 |


| Decision Value | Interpretation |
| :--- | :--- |
| 3 | Least confident 0 |
| 4 | Least confident 1 |
| 5 | Third most confident 1 |
| 6 | Second most confident 1 |
| 7 | Most confident 1 |

## Traceback and Decoding Delay

The Traceback depth parameter in the Viterbi Decoder block represents the length of the decoding delay. Typical values for a traceback depth are about five or six times the constraint length, which would be 35 or 42 in this example. However, some hardware implementations offer options of 48 and 96. This example chooses 48 because that is closer to the targets ( 35 and 42) than 96 is.

## Delay in Received Data

The Receive delay parameter of the Error Rate Calculation block is nonzero because a given message bit and its corresponding recovered bit are separated in time by a nonzero amount of simulation time. The Receive delay parameter tells the block which elements of its input signals to compare when checking for errors.

In this case, the Receive delay value is equal to the Traceback depth value (48).

## Comparing Simulation Results with Theoretical Results

This section describes how to compare the bit error rate in this simulation with the bit error rate that would theoretically result from unquantized decoding. The process includes a few steps, described in these sections:

## Computing Theoretical Bounds for the Bit Error Rate

To calculate theoretical bounds for the bit error rate $P_{\mathrm{b}}$ of the convolutional code in this model, you can use this estimate based on unquantized-decision decoding:

$$
P_{b}<\sum_{d=f}^{\infty} c_{d} P_{d}
$$

In this estimate, $c_{\mathrm{d}}$ is the sum of bit errors for error events of distance $d$, and $f$ is the free distance of the code. The quantity $P_{\mathrm{d}}$ is the pairwise error probability, given by

$$
P_{d}=\frac{1}{2} \operatorname{erfc}\left(\sqrt{d R \frac{E_{b}}{N_{0}}}\right)
$$

where $R$ is the code rate of $1 / 2$, and erfc is the MATLAB complementary error function, defined by

$$
\operatorname{erfc}(x)=\frac{2}{\sqrt{\Pi}} \int_{x}^{\infty} e^{-t^{2}} d t
$$

Values for the coefficients $c_{\mathrm{d}}$ and the free distance $f$ are in published articles such as Frenger, P., P. Orten, and Ottosson, "Convolutional Codes with Optimum Distance Spectrum," IEEE Communications vol. 3, pp. 317-319, November 1999. The free distance for this code is $f=10$.

The following commands calculate the values of $P_{\mathrm{b}}$ for $E_{\mathrm{b}} / N_{0}$ values in the range from 1 to 4 , in increments of 0.5 :

```
EbNoVec = [1:0.5:4.0];
R = 1/2;
% Errs is the vector of sums of bit errors for
% error events at distance d, for d from 10 to 29.
Errs = [36 0 211 0 1404 0 11633 0 77433 0 502690 0,...
    3322763 0 21292910 0 134365911 0 843425871 0];
% P is the matrix of pairwise error probilities, for
% Eb/No values in EbNoVec and d from 10 to 29.
P = zeros(20,7); % Initialize.
for d = 10:29
    P(d-9,:) = (1/2)*erfc(sqrt(d*R*10.^(EbNoVec/10)));
end
% Bounds is the vector of upper bounds for the bit error
% rate, for Eb/No values in EbNoVec.
Bounds = Errs*P;
```


## Simulating Multiple Times to Collect Bit Error Rates

You can efficiently vary the simulation parameters by using the sim function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB , in increments of 0.5 dB . It collects all bit error rates from these simulations in the matrix BERVec. It also plots the bit error rates in a figure window along with the theoretical bounds computed in the preceding code fragment.

Note First open the model by clicking here in the MATLAB Help browser. Then execute these commands, which might take a few minutes.

```
% Plot theoretical bounds and set up figure.
figure;
semilogy(EbNoVec,Bounds,'bo',1,NaN,'r*');
xlabel('Eb/No (dB)'); ylabel('Bit Error Rate');
title('Bit Error Rate (BER)');
legend('Theoretical bound on BER','Actual BER');
axis([1 4 le-5 1]);
hold on;
BERVec = [];
% Make the noise level variable.
set_param('doc_softdecision/AWGN Channel',...
    'EsNodB','EbNodB+10*log10(1/2)');
% Simulate multiple times.
for n = 1:length(EbNoVec)
    EbNodB = EbNoVec(n);
    sim('doc_softdecision',5000000);
    BERVec(n,:) = BER_Data;
    semilogy(EbNoVec(\overline{n}),\operatorname{BERVec}(n,1),'r*'); % Plot point.
    drawnow;
end
hold off;
```

Note The estimate for $P_{\mathrm{b}}$ assumes that the decoder uses unquantized data, that is, an infinitely fine quantization. By contrast, the simulation in this example uses 8 -level (3-bit) quantization. Because of this quantization, the simulated bit error rate is not quite as low as the bound when the signal-tonoise ratio is high.

The plot of bit error rate against signal-to-noise ratio follows. The locations of your actual BER points might vary because the simulation involves random numbers.


## Tailbiting Encoding Using Feedback Encoders

This example demonstrates Tailbiting encoding using feedback encoders. For feedback encoders, the ending state depends on the entire block of data. To accomplish tailbiting, you must calculate for a given information vector (of N bits), the initial state, that leads to the same ending state after the block of data is encoded.

This is achieved in two steps:

- The first step is to determine the zero-state response for a given block of data. The encoder starts in the all-zeros state. The whole block of data is input and the output bits are ignored. After N bits, the encoder is in a state $X_{N}{ }^{[z s]}$. From this state, we calculate the corresponding initial state $X_{0}$ and initialize the encoder with $X_{0}$.
- The second step is the actual encoding. The encoder starts with the initial state $X_{0}$, the data block is input and a valid codeword is output which conforms to the same state boundary condition.

Refer to [8] for a theoretical calculation of the initial state $X_{0}$ from $X_{N}{ }^{[z s]}$ using state-space formulation. This is a one-time calculation which depends on the block length and in practice could be implemented as a look-up table. Here we determine this mapping table by simulating all possible entries for a chosen trellis and block length.


To open the model, type doc_mtailbiting_wfeedback at the MATLAB command line.

```
function mapStValues = getMapping(blkLen, trellis)
% The function returns the mapping value for the given block
length and trellis to be used for determining the initial
state from the zero-state response.
% All possible combinations of the mappings
mapStValuesTab = perms(0:trellis.numStates-1);
% Loop over all the combinations of the mapping entries:
for i = 1:length(mapStValuesTab)
mapStValues = mapStValuesTab(i,:);
% Model parameterized for the Block length
sim('mtailbiting_wfeedback');
% Check the boundary condition for each run
% if ending and starting states match, choose that mapping set
if unique(out)==0
            return
        end
end
```

Selecting the returned mapStValues for the Table data parameter of the Direct Lookup Table ( $\mathrm{n}-\mathrm{D}$ ) block in the Lookup subsystem will perform tailbiting encoding for the chosen block length and trellis.

## 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.
[3] Frenger, P., P. Orten, and T. Ottosson, "Convolution Codes with Optimum Distance Spectrum," IEEE Communications Letters, vol. 3, pp. 317-319, November 1999.

## Linear Block Codes

- "Represent Words for Linear Block Codes" on page 8-54
- "Configure Parameters for Linear Block Codes" on page 8-56
- "Create and Decode Linear Block Codes" on page 8-59


## Represent Words for Linear Block Codes

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

- "Use MATLAB to Create Messages and Codewords in Binary Vector Format" on page 8-54
- "Use MATLAB to Create Messages and Codewords in Binary Matrix Format" on page 8-55
- "Use MATLAB to Create Messages and Codewords in Decimal Vector Format" on page 8-55

To learn how to represent words for BCH or Reed-Solomon codes, see "Represent Words for BCH Codes" on page 8-67 or "Represent Words for Reed-Solomon Codes" on page 8-72.

## Use MATLAB to Create Messages and Codewords in 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
```

        10
    Columns 6 through 10
        \(0 \quad 1\)
    Columns 11 through 12
        \(1 \quad 1\)
    ans =
Columns 1 through 5
$1 \quad 1$
Columns 6 through 10
$1 \quad 0$
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 $\mathrm{k}=4$ ) 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.

## Use MATLAB to Create Messages and Codewords in 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.

## Use MATLAB to Create Messages and Codewords in 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.

## Configure Parameters for Linear Block Codes

This subsection describes the items that you might need in order to process [ $\mathrm{n}, \mathrm{k}$ ] 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 8-56 | Generic linear block |
| "Parity-Check Matrix" on page 8-56 | Generic linear block |
| "Generator Polynomial" on page 8-58 | Cyclic |
| "Decoding Table" on page 8-58 | Generic linear block, Hamming |

## Generator Matrix

The process of encoding a message into an $[\mathrm{n}, \mathrm{k}]$ linear block code is determined by a k -by- n generator matrix G. Specifically, the 1-by-k message vector vis encoded into the 1-by-n codeword vector vG. If G has the form $\left[I_{k} P\right]$ or $\left[P I_{k}\right]$, where $P$ is some $k$-by- $(\mathrm{n}-\mathrm{k})$ 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 8-56.

## Parity-Check Matrix

Decoding an [n,k] linear block code requires an ( $\mathrm{n}-\mathrm{k}$ )-by-n parity-check matrix H. It satisfies $\mathrm{GH}^{\mathrm{tr}}=0$ $(\bmod 2)$, where $\mathrm{H}^{\mathrm{tr}}$ denotes the matrix transpose of $\mathrm{H}, \mathrm{G}$ is the code's generator matrix, and this zero matrix is $\mathrm{k}-\mathrm{by}-(\mathrm{n}-\mathrm{k})$. If $\mathrm{G}=\left[\mathrm{I}_{\mathrm{k}} \mathrm{P}\right]$ then $\mathrm{H}=\left[-\mathrm{P}^{\mathrm{tr}} \mathrm{I}_{\mathrm{n}-\mathrm{k}}\right]$. Most functions in this product 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 [ $\mathrm{n}, \mathrm{k}$ ] binary linear block code.

| Type of Matrix | Standard Form | Dimensions |
| :--- | :--- | :--- |
| Generator | $\left[\mathrm{I}_{\mathrm{k}} \mathrm{P}\right]$ or $\left[\mathrm{P}_{\mathrm{k}}\right]$ | k -by-n |
| Parity-check | $\left[-\mathrm{P}^{\prime} \mathrm{I}_{\mathrm{n}-\mathrm{k}}\right]$ or $\left[\mathrm{I}_{\mathrm{n}-\mathrm{k}}-\mathrm{P}^{\prime}\right]$ | $(\mathrm{n}-\mathrm{k})$-by-n |

$\mathrm{I}_{\mathrm{k}}$ 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 $[\mathrm{n}, \mathrm{k}]=\left[2^{3}-1, \mathrm{n}-3\right]=[7,4]$. genmat has the standard form $\left[\mathrm{I}_{\mathrm{k}}\right]$.

```
[parmat,genmat] = hammgen(3)
parmat =
\begin{tabular}{lllllll}
1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1
\end{tabular}
```

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 8-58.

```
genpoly = cyclpoly(7,3);
[parmat,genmat] = cyclgen(7,genpoly)
parmat =
\begin{tabular}{lllllll}
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
\end{tabular}
```

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 =
    lllll
```

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 $\mathrm{x}^{\mathrm{n}}-1$. 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
```

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 $2^{\wedge}(n-k)$ 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.

## Use a Decoding Table in MATLAB

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

```
corrvect =
    0
correctedcode =
    1 0}0
```


## Create and Decode 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 on page $8-59$ codes, cyclic on page $8-60$ codes, and Hamming on page 8-61 codes.

## Generic Linear Block Codes

Encoding a message using a generic linear block code requires a generator matrix. If you have defined variables $\mathrm{msg}, \mathrm{n}, \mathrm{k}$, 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 [ $\mathrm{n}, \mathrm{k}$ ] code that the generator matrix genmat determines. The /decimal option, suitable when $2^{\wedge} \mathrm{n}$ and $2^{\wedge} \mathrm{k}$ are not very large, indicates that msg contains nonnegative decimal integers rather than their binary representations. See "Represent Words for Linear Block Codes" on page 8-54 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 $t r t$, 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 [ $\mathrm{n}, \mathrm{k}$ ] 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 8-58 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 [ $\mathrm{n}, \mathrm{k}$ ] 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 $2^{\wedge} n$ and $2^{\wedge} k$ are not very large, indicates that msg contains nonnegative decimal integers rather than their binary representations. See "Represent Words for Linear Block Codes" on page $8-54$ 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 $t r t$, 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 $[\mathrm{n}, \mathrm{k}]$ 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( $\mathrm{n}, \mathrm{k}$ ).

## Example

You can modify the example in "Generic Linear Block Codes" on page 8-59 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 8-58 illustrates error correction in a Hamming code.

## Hamming Codes

- "Create a Hamming Code in Binary Format Using Simulink" on page 8-61
- "Reduce the Error Rate Using a Hamming Code" on page 8-62


## Create a Hamming Code in Binary Format Using Simulink

This example shows very simply how to use an encoder and decoder. It illustrates the appropriate vector lengths of the code and message signals for the coding blocks. Because the Error Rate Calculation block accepts only scalars or frame-based column vectors as the transmitted and received signals, this example uses frame-based column vectors throughout. (It thus avoids having to change signal attributes using a block such as Convert 1-D to 2-D.)


Open this model by entering doc_hamming at the MATLAB command line. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Comm Sources library
- Set Probability of a zero to . 5 .
- Set Initial seed to any positive integer scalar, preferably the output of the randn function.
- Check the Frame-based outputs check box.
- Set Samples per frame to 4 .
- Hamming Encoder, with default parameter values
- Hamming Decoder, with default parameter values
- Error Rate Calculation, in the Comm Sinks library, with default parameter values

Connect the blocks as in the preceding figure. You can display the vector length of signals in your model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions. After updating the diagram, if necessary, press $\mathbf{C t r l}+\mathbf{D}$ to compile the model and check error statistics.

The connector lines show relevant signal attributes. The connector lines are double lines to indicate frame-based signals, and the annotations next to the lines show that the signals are column vectors of appropriate sizes.

## Reduce the Error Rate Using a Hamming Code

This section describes how to reduce the error rate by adding an error-correcting code. This figure shows model that uses a Hamming code.


To open a complete version of the model, type doc_hamming at the MATLAB prompt.

## Building the Hamming Code Model

You can build the Hamming code model by following these steps:
1 Type doc_channel at the MATLAB prompt to open the channel noise model.


Then save the model as my_hamming in the folder where you keep your work files.
2 From the Simulink Library Browser drag the Hamming Encoder and Hamming Decoder blocks from the Error Detection and Correction/Block sublibrary into the model window.
3 Click the right border of the model and drag it to the right to widen the model window.
4 Move the Binary Symmetric Channel, Error Rate Calculation, and Display blocks to the right by clicking and dragging.
5 Create enough space between the Bernoulli Binary Generator and Binary Symmetric Channel blocks to fit the Hamming Encoder between them.
6 Click and drag the Hamming Encoder block on top of the line between the Bernoulli Binary Generator block and the Binary Symmetric Channel block, to the right of the branch point, as shown in the following figure. Then release the mouse button. The Hamming Encoder block should automatically connect to the line from the Bernoulli Binary Generator block to the Binary Symmetric Channel block.
7 Move blocks again to create enough space between the Binary Symmetric Channel and the Error Rate Calculation blocks to fit the Hamming Decoder between them.
8 Click and drag the Hamming Decoder block on top of the line between the Binary Symmetric Channel block and the Error Rate Calculation block.

The model should now resemble this figure.


## Using the Hamming Encoder and Decoder Blocks

The Hamming Encoder block encodes the data before it is sent through the channel. The default code is the $[7,4]$ Hamming code, which encodes message words of length 4 into codewords of length 7 . As a result, the block converts frames of size 4 into frames of size 7. The code can correct one error in each transmitted codeword.

For an $[n, k]$ code, the input to the Hamming Encoder block must consist of vectors of size $k$. In this example, $\mathrm{k}=4$.

The Hamming Decoder block decodes the data after it is sent through the channel. If at most one error is created in a codeword by the channel, the block decodes the word correctly. However, if more than one error occurs, the Hamming Decoder block might decode incorrectly.

To learn more about block coding features, see Block Codes on page 8-16.

## Setting Parameters in the Hamming Code Model

Double-click the Bernoulli Binary Generator block and make the following changes to the parameter settings in the block's dialog box, as shown in the following figure:

1 Set Samples per frame to 4. This converts the output of the block into frames of size 4, in order to meet the input requirement of the Hamming Encoder Block. See "Sample-Based and FrameBased Processing" on page 2-4 for more information about frames.

Note Many blocks, such as the Hamming Encoder block, require their input to be a vector of a specific size. If you connect a source block, such as the Bernoulli Binary Generator block, to one of these blocks, set Samples per frame to the required value. For this model the Samples per frame parameter of the Bernoulli Binary Generator block must be a multiple of the Message Length K parameter of the Hamming Encoder block.

## Labeling the Display Block

You can change the label that appears below a block to make it more informative. For example, to change the label below the Display block to 'Error Rate Display', first select the label with the mouse. This causes a box to appear around the text. Enter the changes to the text in the box.

## Running the Hamming Code Model

To run the model, select Simulation $>$ Run. The model terminates after 100 errors occur. The error rate, displayed in the top window of the Display block, is approximately .001. You get slightly different results if you change the Initial seed parameters in the model or run a simulation for a different length of time.

You expect an error rate of approximately . 001 for the following reason: The probability of two or more errors occurring in a codeword of length 7 is

If the codewords with two or more errors are decoded randomly, you expect about half the bits in the decoded message words to be incorrect. This indicates that .001 is a reasonable value for the bit error rate.

To obtain a lower error rate for the same probability of error, try using a Hamming code with larger parameters. To do this, change the parameters Codeword length and Message length in the Hamming Encoder and Hamming Decoder block dialog boxes. You also have to make the appropriate changes to the parameters of the Bernoulli Binary Generator block and the Binary Symmetric Channel block.

## Displaying Frame Sizes

You can display the sizes of data frames in different parts in your model. On the Debug tab, expand Information Overlays. In the Signals section, select Signal Dimensions. The line leading out of the Bernoulli Binary Generator block is labeled [ $4 \times 1$ ], indicating that its output consists of column vectors of size 4. Because the Hamming Encoder block uses a [7,4] code, it converts frames of size 4 into frames of size 7, so its output is labeled [7x1].


## Adding a Scope to the Model

To display the channel errors produced by the Binary Symmetric Channel block, add a Scope block to the model. This is a good way to see whether your model is functioning correctly. The example shown in the following figure shows where to insert the Scope block into the model.


To build this model from the one shown in the figure "Reduce the Error Rate Using a Hamming Code" on page $8-62$, follow these steps:

1 Drag the following blocks from the Simulink Library Browser into the model window:

- Relational Operator block, from the Simulink Logic and Bit Operations library
- Scope block, from the Simulink Sinks library
- Two copies of the Unbuffer block, from the Buffers sublibrary of the Signal Management library in DSP System Toolbox
2 Double-click the Binary Symmetric Channel block to open its dialog box, and select Output error vector. This creates a second output port for the block, which carries the error vector.
3 Double-click the Scope block, under View > Configuration Properties, set Number of input ports to 2 . Select Layout and highlight two blocks vertically. Click OK.
4 Connect the blocks as shown in the preceding figure.


## Setting Parameters in the Expanded Model

Make the following changes to the parameters for the blocks you added to the model.

- Error Rate Calculation Block - Double-click the Error Rate Calculation block and clear the box next to Stop simulation in the block's dialog box.
- Scope Block - The Scope block displays the channel errors and uncorrected errors. To configure the block,

1 Double-click the Scope block, select View > Configuration Properties.
2 Select the Time tab and set Time span to 5000.
3 Select the Logging tab and set Limit data points to last to 30000.
4 Click OK.
5 The scope should now appear as shown.


6 To configure the axes, follow these steps:
a Right-click the vertical axis at the left side of the upper scope.
b In the context menu, select Configuration Properties.
c Set $\mathbf{Y}$-limits (Minimum) to - 1 .
d Set $\mathbf{Y}$-limits (Maximum) to 2, and click $\mathbf{O K}$.
e Repeat the same steps for the vertical axis of the lower scope.
f Widen the scope window until it is roughly three times as wide as it is high. You can do this by clicking the right border of the window and dragging the border to the right, while pressing the left-mouse button.

- Relational Operator - Set Relational Operator to ~= in the block's dialog box. The Relational Operator block compares the transmitted signal, coming from the Bernoulli Random Generator block, with the received signal, coming from the Hamming Decoder block. The block outputs a 0 when the two signals agree and a 1 when they disagree.


## Observing Channel Errors with the Scope

When you run the model, the scope displays the error data. At the end of each 5000 time steps, the scope appears as shown this figure. The scope then clears the displayed data and displays the next 5000 data points.


The upper scope shows the channel errors generated by the Binary Symmetric Channel block. The lower scope shows errors that are not corrected by channel coding.

Click the Stop button on the toolbar at the top of the model window to stop the scope.
You can see individual errors by zooming in on the scope. First click the middle magnifying glass button at the top left of the Scope window. Then click one of the lines in the lower scope. This zooms in horizontally on the line. Continue clicking the lines in the lower scope until the horizontal scale is fine enough to detect individual errors. A typical example of what you might see is shown in the figure below.


The wider rectangular pulse in the middle of the upper scope represents two 1s. These two errors, which occur in a single codeword, are not corrected. This accounts for the uncorrected errors in the
lower scope. The narrower rectangular pulse to the right of the upper scope represents a single error, which is corrected.

When you are done observing the errors, select Simulation > Stop.
"Export Data to MATLAB" on page 1-3 explains how to send the error data to the MATLAB workspace for more detailed analysis.

## BCH Codes

- "Represent Words for BCH Codes" on page 8-67
- "Parameters for BCH Codes" on page 8-67
- "Create and Decode BCH Codes" on page 8-68
- "Algorithms for BCH and RS Errors-only Decoding" on page 8-69


## Represent Words for BCH Codes

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

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

```
msg = [1 0 0 1 0; 1 0 1 1 1]; % Messages in a Galois array
obj = comm.BCHEncoder;
cl = step(obj, msg(1,:)');
c2 = step(obj, msg(2,:)');
cbch = [c1 c2].'
```

The output is
Columns 1 through 5
$1 \quad 0$
0
0
0
1
0
$1 \quad 0$
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
$\begin{array}{ll}0 & 1\end{array}$
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 $2^{\mathrm{m}}-1$ for some integer $\mathrm{m}>2$.
- $k$, 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 BCH Encoder block reference page for a list of some valid values of $k$ corresponding to values of $n$ up to 511 .


## Create and Decode BCH Codes

The BCH Encoder and BCH Decoder System objects create and decode BCH codes, using the data described in "Represent Words for BCH Codes" on page 8-67 and "Parameters for BCH Codes" on page 8-67.

The topics are

- "Example: BCH Coding Syntaxes" on page 8-68
- "Detect and Correct Errors in a BCH Code Using MATLAB" on page 8-68


## Example: BCH Coding Syntaxes

The example below illustrates how to encode and decode data using a [15, 5] BCH code.

```
n = 15; k = 5; % Codeword length and message length
msg = randi([0 1],4*k,1); % Four random binary messages
% Simplest syntax for encoding
enc = comm.BCHEncoder(n,k);
dec = comm.BCHDecoder(n,k);
cl = step(enc,msg); % BCH encoding
d1 = step(dec,c1); % BCH decoding
% Check that the decoding worked correctly.
chk = isequal(d1,msg)
% The following code shows how to perform the encoding and decoding
% operations if one chooses to prepend the parity symbols.
% Steps for converting encoded data with appended parity symbols
% to encoded data with prepended parity symbols
c11 = reshape(c1, n, []);
c12 = circshift(cll,n-k);
c1_prepend = c12(:); % BCH encoded data with prepended parity symbols
% Steps for converting encoded data with prepended parity symbols
% to encoded data with appended parity symbols prior to decoding
c21 = reshape(c1_prepend, n, []);
c22 = circshift(c21,k);
c1_append = c22(:); % BCH encoded data with appended parity symbols
% Check that the prepend-to-append conversion worked correctly.
d1_append = step(dec,c1_append);
ch\overline{k}}=\mathrm{ isequal(msg,d1_append)
```

The output is below.
chk =
1

## Detect and Correct Errors in a BCH Code Using MATLAB

The following example illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and attempts to decode the noisy code using the BCH Decoder System object.

```
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 = randi([0 1], nw*k, 1); % Random k-symbol messages
enc = comm.BCHEncoder(n,k,gp);
dec = comm.BCHDecoder(n,k,gp);
c = step(enc, msgw); % Encode the data.
noise = randerr(nw,n,t); % t errors per codeword
noisy = noise';
noisy = noisy(:);
cnoisy = mod(c + noisy,2); % Add noise to the code.
[dc, nerrs] = step(dec, cnoisy); % Decode cnoisy.
% Check that the decoding worked correctly.
chk2 = isequal(dc,msgw)
nerrs % Find out how many errors have been corrected.
```

Notice that the array of noise values contains binary values, and that the addition operation $\mathrm{c}+$ noise takes place in the Galois field GF(2) because c is a Galois field 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, the BCH Decoder System object corrected all the errors. However, each BCH code has a finite error-correction capability. To learn more about how the BCH Decoder System object behaves when the noise is excessive, see the analogous discussion for Reed-Solomon codes in "Excessive Noise in Reed-Solomon Codewords" on page 8-75.

## Algorithms for BCH and RS 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 [2]).

1 Calculate the first $2 t$ terms of the infinite degree syndrome polynomial, $S(z)$.
2 If those $2 t$ 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

$$
\Lambda(z) S(z)=\Omega(z) \bmod z^{2 t}
$$

5 Correct an error in the codeword according to
$e_{i_{m}}=\frac{\Omega\left(\alpha^{-i_{m}}\right)}{\Lambda^{\prime}\left(\alpha^{-i_{m}}\right)}$
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, $\Lambda^{\prime}(z)$ is the formal derivative [5] of the error locator polynomial, $\Lambda(z)$, and $\alpha$ 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 $2 t$ terms of $S(z)$ are calculated by evaluating the received codeword at successive powers of $\alpha$ (the field's primitive element) from 0 to $2 t-1$. 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 [ $c_{1} c_{1} \ldots c_{N}$ ], 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 [2], 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 $2 t$ terms <br> of $S(z)$ |
| $D(z)$ | Correction polynomial |
| $d$ | Discrepancy |

The following diagram shows the iterative procedure (i.e., the Berlekamp algorithm) used to find $\Lambda(z)$.


## Error Evaluator Polynomial Calculation

The error evaluator polynomial, $\Omega(z)$, is simply the convolution of $\Lambda(z)$ and $S(z)$.

## Reed-Solomon Codes

- "Represent Words for Reed-Solomon Codes" on page 8-72
- "Parameters for Reed-Solomon Codes" on page 8-72
- "Create and Decode Reed-Solomon Codes" on page 8-73
- "Find a Generator Polynomial" on page 8-76
- "Reed Solomon Examples with Shortening, Puncturing, and Erasures" on page 8-77


## Represent Words for Reed-Solomon Codes

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

Note For information about Galois field arrays and how to create them, see "Representing Elements of Galois Fields" on page 8-81 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 = [1 6 4; 0 4 3]; % Message is a Galois array.
obj = comm.RSEncoder(n, k);
cl = step(obj, msg(1,:)');
c2 = step(obj, msg(2,:)');
c = [c1 c2].'
```

The output is
C =

| 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 on page 8-72.

## 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 <br> codeword | Integer between 3 and $2^{\mathrm{m}}-1$ |
| k | Number of symbols per message | Positive integer less than n, <br> such that $\mathrm{n}-\mathrm{k}$ is even |
| t | Error-correction capability of <br> the code | $\mathrm{n}-\mathrm{k}) / 2$ |

## Generator Polynomial

The rsgenpoly function produces generator polynomials for Reed-Solomon codes. 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 $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. 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 $\mathrm{X}^{2}+\left(\mathrm{A}^{2}+\mathrm{A}\right) \mathrm{X}+\left(\mathrm{A}^{3}\right)$, where A is a root of the default primitive polynomial for $\mathrm{GF}(16)$.

## Algebraic Expression for Generator Polynomials

The generator polynomials that rsgenpoly produces have the form ( $\mathrm{X}-\mathrm{A}^{\mathrm{b}}$ ) $\left(\mathrm{X}-\mathrm{A}^{\mathrm{b}+1}\right) \ldots\left(\mathrm{X}-\mathrm{A}^{\mathrm{b}+2 \mathrm{t}-1}\right)$, 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 $[15,13$ ] ReedSolomon code, using b=1.

```
n = 15;
```

$a=g f(2, \log 2(n+1)) ;$ Root of primitive polynomial
f1 = [1 a]; f2 = [1 a^2]; \% Factors that form generator polynomial
$f=\operatorname{conv}(f 1, f 2) \%$ Generator polynomial, same as $r$ above.

## Create and Decode Reed-Solomon Codes

The RS Encoder and RS Decoder System objects create and decode Reed-Solomon codes, using the data described in "Represent Words for Reed-Solomon Codes" on page 8-72 and "Parameters for Reed-Solomon Codes" on page 8-72.

This section illustrates how to use the RS Encoder and RS Decoder System objects. The topics are

- "Reed-Solomon Coding Syntaxes in MATLAB" on page 8-73
- "Detect and Correct Errors in a Reed-Solomon Code Using MATLAB" on page 8-75
- "Excessive Noise in Reed-Solomon Codewords" on page 8-75
- "Create Shortened Reed-Solomon Codes" on page 8-76


## Reed-Solomon Coding Syntaxes in MATLAB

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 g .
- 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 the RS Encoder and RS Decoder System objects 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
msg = randi([0 m-1],4*k,1); % Four random integer messages
% Simplest syntax for encoding
hEnc = comm.RSEncoder(n,k);
hDec = comm.RSDecoder(n,k);
cl = step(hEnc, msg);
d1 = step(hDec, cl);
% Vary the generator polynomial for the code.
release(hEnc), release(hDec)
hEnc.GeneratorPolynomialSource = 'Property';
hDec.GeneratorPolynomialSource = 'Property';
hEnc.GeneratorPolynomial = rsgenpoly(n,k,19,2);
hDec.GeneratorPolynomial = rsgenpoly(n,k,19,2);
c2 = step(hEnc, msg);
d2 = step(hDec, c2);
% Vary the primitive polynomial for GF(16).
release(hEnc), release(hDec)
hEnc.PrimitivePolynomialSource = 'Property';
hDec.PrimitivePolynomialSource = 'Property';
hEnc.GeneratorPolynomialSource = 'Auto';
hDec.GeneratorPolynomialSource = 'Auto';
hEnc.PrimitivePolynomial = [1 1 0 0 1];
hDec.PrimitivePolynomial = [ll 1 0 0 1];
c3 = step(hEnc, msg);
d3 = step(hDec, c3);
% Check that the decoding worked correctly.
chk = isequal(d1,msg) & isequal(d2,msg) & isequal(d3,msg)
% The following code shows how to perform the encoding and decoding
% operations if one chooses to prepend the parity symbols.
% Steps for converting encoded data with appended parity symbols
% to encoded data with prepended parity symbols
c31 = reshape(c3, n, []);
c32 = circshift(c31,n-k);
c3_prepend = c32(:); % RS encoded data with prepended parity symbols
% Steps for converting encoded data with prepended parity symbols
% to encoded data with appended parity symbols prior to decoding
c34 = reshape(c3_prepend, n, []);
c35 = circshift(c34,k);
c3_append = c35(:); % RS encoded data with appended parity symbols
% Check that the prepend-to-append conversion worked correctly.
d3_append = step(hDec,c3_append);
ch\overline{k}= isequal(msg,d3_append)
```

The output is

```
chk =
```


## Detect and Correct Errors in a Reed-Solomon Code Using MATLAB

The example below illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and attempts to decode the noisy code using the RS Decoder System object.

```
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 = randi([0 n],nw*k,1); % Random k-symbol messages
hEnc = comm.RSEncoder(n,k);
hDec = comm.RSDecoder(n,k);
c = step(hEnc, msgw); % Encode the data.
noise = (1+randi([0 n-1],nw,n)).*randerr(nw,n,t); % t errors per codeword
noisy = noise';
noisy = noisy(:);
cnoisy = gf(c,m) + noisy; % Add noise to the code under gf(m) arithmetic.
[dc nerrs] = step(hDec, cnoisy.x); % Decode the noisy code.
% Check that the decoding worked correctly.
isequal(dc,msgw)
nerrs % Find out how many errors hDec corrected.
```

The array of noise values contains integers between 1 and $2^{\wedge} m$, and the addition operation $c+$ noise takes place in the Galois field $G F\left(2^{\wedge} m\right)$ because $c$ is a Galois field array in $G F\left(2^{\wedge} m\right)$.

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, RS Encoder System object corrected all of the errors. However, each ReedSolomon 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 RS Decoder System object 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+randi([0 n-1],nw,n)).*randerr(nw,n,t+1); % t+1 errors/row
```


## Create 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 $2^{\mathrm{m}}-1$. A shortened [ $\mathrm{n}, \mathrm{k}$ ] Reed-Solomon code implicitly uses an [ $\mathrm{n}_{1}, \mathrm{k}_{1}$ ] encoder, where

- $n_{1}=2^{m}-1$, where $m$ is the number of bits per symbol
- $\mathrm{k}_{1}=\mathrm{k}+\left(\mathrm{n}_{1}-\mathrm{n}\right)$

The RS Encoder System object supports shortened codes using the same syntaxes it uses for nonshortened codes. You do not need to indicate explicitly that you want to use a shortened code.

```
hEnc = comm.RSEncoder(7,5);
ordinarycode = step(hEnc,[11 1 1 1 1]');
hEnc = comm.RSEncoder(5,3);
shortenedcode = step(hEnc,[11 1 1 ]');
```


## How the RS Encoder System Object Creates a Shortened Code

When creating a shortened code, the RS Encoder System object 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 following example illustrates this process.

```
n = 12; k = 8; % Lengths for the shortened code
m = ceil(log2(n+1)); % Number of bits per symbol
msg = randi([0 2^m-1],3*k,1); % Random array of 3 k-symbol words
hEnc = comm.RSEncoder(n,k);
code = step(hEnc, msg); % 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); % Messageword length in the actual encoder
h\overline{Enc = comm.\overline{RSEncoder(n_pad,k_pad);}}\mathbf{~}\mathrm{ ;}
mw = reshape(msg,k,[]); % Each column vector represents a messageword
msg_pad = [zeros(n_pad-n,3); mw]; % Prepend zeros to each word.
msg_pad = msg_pad(:);
code_pad = step(hEnc,msg_pad); % Encode padded words.
cw = reshape(code_pad,2^m-1,[]); % Each column vector represents a codeword
code_eqv = cw(n_pad-n+1:n_pad,:); % Remove extra zeros.
code_eqv = code_eqv(:);
ck =- isequal(code_eqv,code); % Returns true (1).
```


## Find 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++\mp@subsup{x}{}{\wedge}4+\mp@subsup{x}{}{\wedge}2+x+1
genpolyRS = rsgenpoly(15,5)
```

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

```
genpolyCyclic =
    \(\begin{array}{lllllllllll}1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1\end{array}\)
genpolyBCH \(=\) GF(2) array.
Array elements =
\begin{tabular}{lllllllllll}
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1
\end{tabular}
genpolyRS \(=\) GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)
Array elements =
\begin{tabular}{lllllllllll}
1 & 4 & 8 & 10 & 12 & 9 & 4 & 2 & 12 & 2 & 7
\end{tabular}
```

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 8-83 and "Polynomials over Galois Fields" on page 8-97.


## 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 ( $\mathrm{X}-\mathrm{A}^{\mathrm{b}}$ ) $\left(\mathrm{X}-\mathrm{A}^{\mathrm{b}}\right.$ $\left.{ }^{+1}\right) \ldots\left(\mathrm{X}-\mathrm{A}^{b+2 t-1}\right)$, 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.

## Reed Solomon Examples with Shortening, Puncturing, and Erasures

In this section, a representative example of Reed Solomon coding with shortening, puncturing, and erasures is built with increasing complexity of error correction.

## Encoder Example with Shortening and Puncturing

The following figure shows a representative example of a $(7,3)$ Reed Solomon encoder with shortening and puncturing.


In this figure, the message source outputs two information symbols, designated by $\mathrm{I}_{1} \mathrm{I}_{2}$. (For a BCH example, the symbols are simply binary bits.) Because the code is a shortened $(7,3)$ code, a zero must be added ahead of the information symbols, yielding a three-symbol message of $0 \mathrm{I}_{1} \mathrm{I}_{2}$. The modified message sequence is then RS encoded, and the added information zero is subsequently removed, which yields a result of $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{2} \mathrm{P}_{3} \mathrm{P}_{4}$. (In this example, the parity bits are at the end of the codeword.)

The puncturing operation is governed by the puncture vector, which, in this case, is 1011 . Within the puncture vector, a 1 means that the symbol is kept, and a 0 means that the symbol is thrown away. In this example, the puncturing operation removes the second parity symbol, yielding a final vector of $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{3} \mathrm{P}_{4}$.

## Decoder Example with Shortening and Puncturing

The following figure shows how the RS encoder operates on a shortened and punctured codeword.


This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a (5,2) codeword, because it has been shortened from a $(7,3)$ codeword by one symbol, and one symbol has also been punctured.

As a first step, the decoder adds an erasure, designated by E, in the second parity position of the codeword. This corresponds to the puncture vector 1011. Adding a zero accounts for shortening, in the same way as shown in the preceding figure. The single erasure does not exceed the erasurecorrecting capability of the code, which can correct four erasures. The decoding operation results in the three-symbol message $\mathrm{DI}_{1} \mathrm{I}_{2}$. The first symbol is truncated, as in the preceding figure, yielding a final output of $\mathrm{I}_{1} \mathrm{I}_{2}$.

## Encoder Example with Shortening, Puncturing, and Erasures

The following figure shows the decoder operating on the punctured, shortened codeword, while also correcting erasures generated by the receiver.


In this figure, demodulator receives the $\mathrm{I}_{1} \mathrm{I}_{2} \mathrm{P}_{1} \mathrm{P}_{3} \mathrm{P}_{4}$ vector that the encoder sent. The demodulator declares that two of the five received symbols are unreliable enough to be erased, such that symbols 2 and 5 are deemed to be erasures. The 01001 vector, provided by an external source, indicates these erasures. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered.

The decoder blocks receive the codeword and the erasure vector, and perform the erasures indicated by the vector 01001 . Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered. The resulting codeword vector is $\mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{P}_{3} \mathrm{E}$, where E is an erasure symbol.

The codeword is then depunctured, according to the puncture vector used in the encoding operation (i.e., 1011). Thus, an erasure symbol is inserted between $P_{1}$ and $P_{3}$, yielding a codeword vector of $\mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{EP}_{3} \mathrm{E}$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0 \mathrm{I}_{1} \mathrm{EP}_{1} \mathrm{EP}_{3} \mathrm{E}$, such that a $(7,3)$ codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of $\mathrm{DI}_{1} \mathrm{I}_{2}$ (where D refers to a dummy symbol). Finally, the removal of the D symbol from the message vector accounts for the shortening and yields the original $\mathrm{I}_{1} \mathrm{I}_{2}$ vector.

For additional information, see the "Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink" on page 11-17 example.

## 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 LDPC Encoder and LDPC Decoder)

Communications Toolbox performs LDPC Coding using Simulink blocks and MATLAB objects.
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.

## Galois Field Computations

A Galois field is an algebraic field that has a finite number of members. Galois fields having $2^{\mathrm{m}}$ members are used in error-control coding and are denoted $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. This chapter describes how to work with fields that have $2^{\mathrm{m}}$ members, where m is an integer between 1 and 16 . The sections in this chapter are as follows.

- "Galois Field Terminology" on page 8-81
- "Representing Elements of Galois Fields" on page 8-81
- "Arithmetic in Galois Fields" on page 8-86
- "Logical Operations in Galois Fields" on page 8-90
- "Matrix Manipulation in Galois Fields" on page 8-92
- "Linear Algebra in Galois Fields" on page 8-93
- "Signal Processing Operations in Galois Fields" on page 8-95
- "Polynomials over Galois Fields" on page 8-97
- "Manipulating Galois Variables" on page 8-100
- "Speed and Nondefault Primitive Polynomials" on page 8-102
- "Selected Bibliography for Galois Fields" on page 8-103

If you need to use Galois fields having an odd number of elements, see "Galois Fields of Odd
Characteristic" on page 8-103.
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.

Note Please note that the Galois field objects do not support the copy method.

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 $\mathrm{GF}\left(2^{\mathrm{m}}\right)$ is a cyclic generator of the group of nonzero elements of $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. 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 $\mathrm{GF}\left(2^{\mathrm{m}}\right)$ is the minimal polynomial of some primitive element of $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. It is the binary-coefficient polynomial of smallest nonzero degree having a certain primitive element as a root in $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. 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" on page 8-81
- "Creating a Galois field array" on page 8-82
- "Example: Creating Galois Field Variables" on page 8-82
- "Example: Representing Elements of GF(8)" on page 8-83
- "How Integers Correspond to Galois Field Elements" on page 8-83
- "Example: Representing a Primitive Element" on page 8-84
- "Primitive Polynomials and Element Representations" on page 8-84


## Section Overview

This section describes how to create a Galois field 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 field array

To begin working with data from a Galois field GF( $\left.2^{\wedge} \mathrm{m}\right)$, 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 field 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 $G F\left(2^{\wedge} m\right)$. 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 $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ you are using in the representations in $x$. If you omit this input argument, $g f$ uses a default primitive polynomial for $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$. For information about this argument, see "Primitive Polynomials and Element Representations" on page 8-84.

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 field 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 field array

Some operations on Galois field arrays require multiple arguments. If you specify one argument that is a Galois field array and another that is an ordinary MATLAB array, MATLAB interprets both as Galois field 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 887.

## 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 field arrays named a and b. Each output section indicates

- The field containing the variable, namely, $\mathrm{GF}\left(2^{\wedge} 2\right)=\mathrm{GF}(4)$.
- 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 field 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 field array mean, the table below lists the elements of the field $\mathrm{GF}(8)$ as integers and as polynomials in a primitive element, A. The table should help you interpret a Galois field array like

```
gf8 = gf([0:7],3); % Galois vector in GF(2^3)
```

| Integer Representation | Binary Representation | Element of GF(8) |
| :--- | :--- | :--- |
| 0 | 000 | 0 |
| 1 | 001 | 1 |
| 2 | 010 | A |
| 3 | 011 | $\mathrm{~A}+1$ |
| 4 | 100 | $\mathrm{~A}^{2}$ |
| 5 | 101 | $\mathrm{~A}^{2}+1$ |
| 6 | 110 | $\mathrm{~A}^{2}+\mathrm{A}$ |
| 7 | 111 | $\mathrm{~A}^{2}+\mathrm{A}+1$ |

## How Integers Correspond to Galois Field Elements

Building on the GF(8) example above on page 8-83, this section explains the interpretation of array elements in a Galois field array in greater generality. The field GF( $2^{\wedge} m$ has $2^{\wedge} \mathrm{m}$ distinct elements, which this toolbox labels as $0,1,2, \ldots, 2^{\wedge} m-1$. 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 $2^{\wedge} m-1$ 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 $\mathrm{m}-1$. Evaluating the binary polynomial at a primitive element of $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ leads to an element of the field.

Conversely, any element of $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ can be expressed as a binary polynomial of order at most $\mathrm{m}-1$, 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 $2^{\wedge} \mathrm{m}$.

Below is a symbolic illustration of the correspondence of an integer $X$, representable in binary form, with a Galois field element. Each $b_{\mathrm{k}}$ is either zero or one, while A is a primitive element.

$$
\begin{aligned}
& X=b_{m-1} \cdot 2^{m-1}+\cdots+b_{2} \cdot 4+b_{1} \cdot 2+b_{0} \\
& \leftrightarrow b_{m-1} \cdot A^{m-1}+\cdots+b_{2} \cdot A^{2}+b_{1} \cdot A+b_{0}
\end{aligned}
$$

## Example: Representing a Primitive Element

The code below defines a variable alph that represents a primitive element of the field $\mathrm{GF}\left(2^{4}\right)$.

```
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 field 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 $0 \mathrm{~A}^{3}+0 \mathrm{~A}^{2}+\mathrm{A}+0$ using the four lowest powers of A)


## Primitive Polynomials and Element Representations

This section builds on the discussion in "Creating a Galois field array" on page $8-82$ by describing how to specify your own primitive polynomial when you create a Galois field array. The topics are

If you perform many computations using a nondefault primitive polynomial, see "Speed and Nondefault Primitive Polynomials" on page 8-102.

## Specifying the Primitive Polynomial

The discussion in "How Integers Correspond to Galois Field Elements" on page 8-83 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 field 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 field array, use a syntax like
$\mathrm{c}=\mathrm{gf}(5,4,25) \% 25$ indicates the primitive polynomial for $\mathrm{GF}(16)$.
instead of

```
cl= 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 $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ in a way similar to the representation described in "How Integers Correspond to Galois Field Elements" on page $8-83$. In this case, the integer 25 corresponds to a binary representation of 11001, which in turn corresponds to the polynomial $\mathrm{D}^{4}+\mathrm{D}^{3}+1$.

Note When you specify the primitive polynomial, the input argument must have a binary representation using exactly $\mathrm{m}+1$ bits, not including unnecessary leading zeros. In other words, a primitive polynomial for $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ 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 
```

Note After you have defined a Galois field 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 $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$ and the isprimitive function to determine whether a polynomial is primitive for $\mathrm{GF}\left(2^{\wedge} \mathrm{m}\right)$. 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)
il = 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) =
```

$D^{\wedge} 4+D^{\wedge} 1+1$
defaultprimpoly =
19
Primitive polynomial(s) =
D^4+D^1+1
$D^{\wedge} 4+D^{\wedge} 3+1$
allprimpolys =
19
25
i1 =
1
i2 =

## 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 field array. For example, exponentiation of a primitive element makes it easy to see how the primitive polynomial affects the representations of field elements.

```
all = 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 + al3.^2 + 1 % 0 because al3 satisfies the equation
nz = al1.^3 + all.^2 + 1 % Nonzero. all does not satisfy equation.
```

The output below shows that when the primitive polynomial has integer representation 13, the Galois field array satisfies a certain equation. By contrast, when the primitive polynomial has integer representation 11, the Galois field 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

- "Section Overview" on page 8-86
- "Example: Addition and Subtraction" on page 8-87
- "Example: Multiplication" on page 8-88
- "Example: Division" on page 8-89
- "Example: Exponentiation" on page 8-89
- "Example: Elementwise Logarithm" on page 8-90


## Section Overview

You can perform arithmetic operations on Galois field arrays by using familiar MATLAB operators, listed in the table below. Whenever you operate on a pair of Galois field arrays, both arrays must be in the same Galois field.

| Operation | Operator |
| :--- | :--- |
| Addition | + |
| Subtraction | - |


| Operation | Operator |
| :--- | :--- |
| Elementwise multiplication | .$^{*}$ |
| Matrix multiplication | $*$ |
| Elementwise left division | .$/$ |
| Elementwise right division | .$\backslash$ |
| Matrix left division | $/$ |
| Matrix right division | \} $\\ {\hline \text { Elementwise exponentiation }} &{.^{\wedge}} \\ {\hline \text { Elementwise logarithm }} &{\log ()} \\ {\hline$ Exponentiation of a square Galois matrix by a  <br>  scalar integer $} &{\wedge} \\ {\hline}$ |

For multiplication and division of polynomials over a Galois field, see "Addition and Subtraction of Polynomials" on page 8-97.

## Example: Addition and Subtraction

The code below adds two Galois field 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 $\mathrm{gf}(6,3)$ plus $g f(3,3)$ equals $g f(5,3)$. 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 field array from an ordinary MATLAB array. The Galois field arrays $h$ and $h 1$ 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 =
$4 \quad 4 \quad 4$
$\begin{array}{lll}4 & 4 & 4\end{array}$

Notice that $1+5$ is reported as 4 in the Galois field. This is true because the 5 represents the polynomial expression $\mathrm{A}^{2}+1$, and $1+\left(\mathrm{A}^{2}+1\right)$ in $\mathrm{GF}(16)$ is $\mathrm{A}^{2}$. Furthermore, the integer that represents the polynomial expression $\mathrm{A}^{2}$ is 4.

## Example: Multiplication

The example below multiplies individual elements in a Galois field 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 field array using the ./ and . $\backslash$ 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 $n z$ before computing; alternatively, you can use as arguments two arrays of the same size.

```
m = 5;
```

$n z=g f\left(\left[1: 2^{\wedge} m-1\right], m\right) ;$ N Nonzero elements of the field
inv1 = 1 ./ nz; \% Divide 1 by each element.
inv2 = nz . $\backslash 1$; \% Obtain same result using . $\backslash$ operator.

## Matrix Division

This example divides the identity array by the square Galois field 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 $\backslash$. 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 field array. To perform matrix exponentiation on a Galois field array, you must use a square Galois field 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 . ${ }^{\wedge}$ 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 =
```


## 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 field array. The output indicates how to express each nonzero element of $\mathrm{GF}(8)$ 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)
exp\overline{format = log(gf8_nonzero) % Logarithm of each element}
```

The output is

```
expformat =
```

$\begin{array}{lllllll}0 & 1 & 3 & 2 & 6 & 4 & 5\end{array}$
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 $\mathrm{gf}(7,3)$ in $\mathrm{GF}(8)$ 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 8-83 for more details.


## Logical Operations in Galois Fields

- "Section Overview" on page 8-90
- "Testing for Equality" on page 8-90
- "Testing for Nonzero Values" on page 8-91


## Section Overview

You can apply logical tests to Galois field arrays and obtain a logical array. Some important types of tests are testing for the equality of two Galois field arrays and testing for nonzero values on page 891 in a Galois field array.

## Testing for Equality

To compare corresponding elements of two Galois field arrays that have the same size, use the operators $==$ and $\sim=$. 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 field 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 field 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 field 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(randi([0 2^m-1],6,1),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 = [0:4 4 0; 3:7 4 5]
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

- "Basic Manipulations of Galois Field Arrays" on page 8-92
- "Basic Information About Galois Field Arrays" on page 8-92


## Basic Manipulations of Galois Field Arrays

Basic array operations on Galois field 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 <br> instead of a vector of explicit indices | a(vector) or a (vector, vector1), where <br> vector and/or vectorl can be ": " instead of a <br> vector |
| Transpose array | $a^{\prime}$ |
| Concatenate matrices | $[\mathrm{a}, \mathrm{b}]$ or $[\mathrm{a} ; \mathrm{b}]$ |
| Create array having specified diagonal elements | $\operatorname{diag}(\mathrm{vector})$ or diag (vector, k$)$ |
| Extract diagonal elements | $\operatorname{diag}(\mathrm{a})$ or diag $(\mathrm{a}, \mathrm{k})$ |
| Extract lower triangular part | $\operatorname{tril}(\mathrm{a})$ or $\operatorname{tril}(\mathrm{a}, \mathrm{k})$ |
| Extract upper triangular part | $\operatorname{triu}(\mathrm{a})$ or $\mathrm{triu}(\mathrm{a}, \mathrm{k})$ |
| Change shape of array | $\operatorname{reshape}(\mathrm{a}, \mathrm{k} 1, \mathrm{k} 2)$ |

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 Field Arrays

You can determine the length of a Galois vector or the size of any Galois field array using the length and size functions. The functionality for Galois field 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 field 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 field array are the positions of nonzero elements. For an ordinary MATLAB array, you might use the find function. However, for a Galois field array, you should use find in conjunction with the $\sim=$ 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" on page 8-93
- "Computing Ranks" on page 8-93
- "Factoring Square Matrices" on page 8-94
- "Solving Linear Equations" on page 8-94

Inverting Matrices and Computing Determinants
To invert a square Galois field array, use the inv function. Related is the det function, which computes the determinant of a Galois field 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 field 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(randi([0 2^m-1],4,4),m);
gfid = gf(eye(4),m);
if det(randommatrix) ~= 0
    invmatrix = inv(randommatrix);
    check1 = invmatrix * randommatrix;
    check2 = randommatrix * invmatrix;
    if (isequal(check1,gfid) & isequal(check2,gfid))
        disp('inv found the correct matrix inverse.');
    end
else
    disp('The matrix is not invertible.');
end
```

The output from this example is either of these two messages, depending on whether the randomly generated matrix is nonsingular or singular.
inv found the correct matrix inverse. The matrix is not invertible.

## Computing Ranks

To compute the rank of a Galois field 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 field 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 $=$
23
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 field array (or a permutation of it) as the product of a lower triangular Galois field array and an upper triangular Galois field 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
cl = 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 field arrays. The table below indicates the equation that each operator addresses, assuming that A and $B$ are previously defined Galois field arrays.

| Operator | Linear Equation | Syntax | Equivalent Syntax Using $\backslash$ |
| :--- | :--- | :--- | :--- |
| Backslash $(\backslash)$ | $A * x=B$ | $x=A \backslash B$ | Not applicable |
| Slash $(/)$ | $x^{*} \mathrm{~A}=\mathrm{B}$ | $\mathrm{x}=\mathrm{B} / \mathrm{A}$ | $\mathrm{x}=\left(\mathrm{A}^{\prime} \backslash \mathrm{B}^{\prime}\right)^{\prime}$ |

The results of the syntax in the table depend on characteristics of the Galois field 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 $\mathrm{A}^{\prime} * \mathrm{~A}$ or $\mathrm{A}^{*} \mathrm{~A}^{\prime}$ is a singular array, or if $A$ is a matrix, where the rows outnumber the columns, 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([12; 0 0],3) \} $\mathrm{gf}([1 ; 0], 3)$ produces an error but the mathematically equivalent $\mathrm{gf}([12], 3) \backslash \mathrm{gf}([1], 3)$ does not. The first syntax fails because $\mathrm{gf}([12 ; 00], 3)$ 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

- "Section Overview" on page 8-95
- "Filtering" on page 8-95
- "Convolution" on page 8-96
- "Discrete Fourier Transform" on page 8-96


## Section Overview

You can perform some signal-processing operations on Galois field arrays, such as filtering on page 895, convolution on page 8-96, and the discrete Fourier transform on page 8-96.

This section describes how to perform these operations.
Other information about the corresponding operations for ordinary real vectors is in the Signal Processing Toolbox ${ }^{\mathrm{TM}}$ 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([ll 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 8-98. 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

Computes the convolution matrix for a vector b in $\mathrm{GF}(4)$. Represent the numerator coefficients for a digital filter, and then illustrate the two equivalent ways to convolve b with x over the Galois field.

```
m = 2; b = gf([lllll
n = 3; x = gf(randi([0 2^m-1],n,1),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 field 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 $G F\left(2^{m}\right)$. The following example illustrates the use of these functions. You can check, using the isequal function, that $y$ equals $y 1, z$ equals $z 1$, and $z$ equals $x$.

```
m = 4;
x = gf(randi([0 2^m-1],2^m-1,1),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" on page 8-97
- "Addition and Subtraction of Polynomials" on page 8-97
- "Multiplication and Division of Polynomials" on page 8-98
- "Evaluating Polynomials" on page 8-98
- "Roots of Polynomials" on page 8-99
- "Roots of Binary Polynomials" on page 8-99
- "Minimal Polynomials" on page 8-100


## 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 $A x^{3}+1 x^{2}+0 x+(A+1)$, where

- A is a primitive element in the field GF $\left(2^{4}\right)$.
- x is the indeterminate quantity in the polynomial.

You can then use such a Galois vector to perform arithmetic with, evaluate on page 8-98, and find roots on page 8-99 of polynomials. You can also find minimal polynomials on page 8-100 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([lll}1\mp@code{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 8-86 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.$^{\wedge}$ 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 $\mathrm{GF}\left(2^{\mathrm{m}}\right)$, the polynomial it represents might have additional roots in some extension field $\mathrm{GF}\left(\left(2^{\mathrm{m}}\right)^{\mathrm{k}}\right)$. However, roots does not find those additional roots or indicate their existence.

The examples below find roots of cubic polynomials in GF(8).

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
    root = rts(ii);
    rootsquared = gfmul(root,root,field);
    rootcubed = gfmul(root,rootsquared,field);
    answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
    % Recall that 1 is really alpha to the zero power.
    % If answer = -Inf, then the variable root represents
    % a root of the polynomial.
end
answer
```


## 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 $\mathrm{GF}(4)$ as well as in $\mathrm{GF}(16)$, so roots 4 and roots 16 are nonempty.

Notice that roots 4 and roots 16 are not equal to each other. They differ in these ways:

- roots 4 is a $\mathrm{GF}(4)$ array, while roots 16 is a $\mathrm{GF}(16)$ array. MATLAB keeps track of the underlying field of a Galois field array.
- The array elements in roots 4 and roots 16 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 roots 4 because the default primitive polynomial for $\mathrm{GF}(4)$ is the same polynomial that gf 4 poly represents. On the other hand, 2 is not an element of roots 16 because the primitive element of $\mathrm{GF}(16)$ is not a root of the polynomial that gf 16 poly represents.


## Minimal Polynomials

The minimal polynomial of an element of $\mathrm{GF}\left(2^{\mathrm{m}}\right)$ is the smallest degree nonzero binary-coefficient polynomial having that element as a root in $\mathrm{GF}\left(2^{\mathrm{m}}\right)$. 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 $g f(6,4)$ is $D^{2}+D+1$ and then checks that $\mathrm{gf}(6,4)$ 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
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

- "Section Overview" on page 8-100
- "Determining Whether a Variable Is a Galois Field Array" on page 8-101
- "Extracting Information from a Galois Field Array" on page 8-101


## Section Overview

This section describes techniques for manipulating Galois variables or for transferring information between Galois field arrays and ordinary MATLAB arrays.

Note These techniques are particularly relevant if you write MATLAB file functions that process Galois field 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 Field Array

To find out whether a variable is a Galois field 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 Field Array

To extract the array elements, field order, or primitive polynomial from a variable that is a Galois field 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 <br> that contains the data values <br> from the Galois field array. |
| Field order | .$m$ | Integer of type double that <br> indicates that the Galois field <br> array is in GF(2^m). |
| Primitive polynomial | . prim_poly | Integer of type uint32 that <br> represents the primitive <br> polynomial. The representation <br> is similar to the description in <br> "How Integers Correspond to <br> Galois Field Elements" on page <br> $8-83$. |

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 field 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);
containst̄wo = roots(primpoly_vect); % Output vector includes 2.
```


## Converting Galois Field 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
$\mathrm{b}=$
10

## Speed and Nondefault Primitive Polynomials

"Primitive Polynomials and Element Representations" on page 8-84 describes 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 field 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.

## Galois Fields of Odd Characteristic

A Galois field is an algebraic field having $\mathrm{p}^{\mathrm{m}}$ 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 on page 8-80. The sections in this chapter are as follows.

- "Galois Field Terminology" on page 8-103
- "Representing Elements of Galois Fields" on page 8-103
- "Default Primitive Polynomials" on page 8-106
- "Converting and Simplifying Element Formats" on page 8-106
- "Arithmetic in Galois Fields" on page 8-109
- "Polynomials over Prime Fields" on page 8-111
- "Other Galois Field Functions" on page 8-113
- "Selected Bibliography for Galois Fields" on page 8-114


## 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 $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ is a cyclic generator of the group of nonzero elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$. 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 $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ is the minimal polynomial of some primitive element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$. As a consequence, it has degree m and is irreducible.


## Representing Elements of Galois Fields

- "Section Overview" on page 8-104
- "Exponential Format" on page 8-104
- "Polynomial Format" on page 8-105
- "List of All Elements of a Galois Field" on page 8-105
- "Nonuniqueness of Representations" on page 8-106


## Section Overview

This section discusses how to represent Galois field elements using this toolbox's exponential on page 8-104 format and polynomial on page 8-105 format. It also describes a way to list all elements on page 8-105 of the Galois field, because some functions use such a list as an input argument. Finally, it discusses the nonuniqueness on page 8-106 of representations of Galois field elements.

The elements of $\mathrm{GF}(\mathrm{p})$ can be represented using the integers from 0 to $\mathrm{p}-1$.
When $m$ is at least $2, \mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ is called an extension field. Integers alone cannot represent the elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ in a straightforward way. MATLAB technical computing software uses two main conventions for representing elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ : 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(pm).

## Exponential Format

This format uses the property that every nonzero element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ can be expressed as $\mathrm{A}^{\mathrm{c}}$ for some integer c between 0 and $\mathrm{p}^{\mathrm{m}}$-2. Higher exponents are not needed, because the theory of Galois fields implies that every nonzero element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ satisfies the equation $\mathrm{x}^{\mathrm{q}-1}=1$ where $\mathrm{q}=\mathrm{p}^{\mathrm{m}}$.

The use of the exponential format is shown in the table below.

| Element of GF $\left(\mathbf{p}^{\mathbf{m}}\right)$ | MATLAB Representation of the Element |
| :--- | :--- |
| 0 | - Inf |
| $\mathrm{A}^{0}=1$ | 0 |
| $\mathrm{~A}^{1}$ | 1 |
| $\cdots$ | $\cdots$ |
| $\mathrm{~A}^{q-2}$ where $\mathrm{q}=\mathrm{p}^{m}$ | $\mathrm{q}-2$ |

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 8-106.

Note The equivalence of all negative integers and - Inf as exponential formats means that, for example, -1 does not represent $\mathrm{A}^{-1}$, 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 $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ can be expressed as a polynomial in A with exponents between 0 and $m-1$, and coefficients in $\mathrm{GF}(\mathrm{p})$. In the polynomial format, the element
$A(1)+A(2) A+A(3) A^{2}+\ldots+A(m) A^{m-1}$
is represented in MATLAB by the vector
$[A(1) A(2) A(3) \ldots A(m)]$

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 $G F\left(p^{m}\right)$. This is again relative to a particular primitive element A of $G F\left(p^{m}\right)$. The proper format for the list of elements is that of a matrix having $\mathrm{p}^{\mathrm{m}}$ 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 8-105 above. The first row contains only zeros because it corresponds to the zero element in $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$. If k is between 2 and $\mathrm{p}^{\mathrm{m}}$, then the kth row specifies the polynomial format of the element $\mathrm{A}^{\mathrm{k}-2}$.

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 $\mathrm{GF}\left(3^{2}\right)$, where A is a root of the primitive polynomial $2+2 x+x^{2}$. This polynomial allows repeated use of the substitution
$\mathrm{A}^{2}=-2-2 \mathrm{~A}=1+\mathrm{A}$
when performing the computations in the middle column of the table.
Elements of GF(9)

| Exponential Format | Polynomial Format | Row of MATLAB Matrix of <br> Elements |
| :--- | :--- | :--- |
| $A^{-\operatorname{lnf}}$ | 0 | $0 \quad 0$ |
| $A^{0}$ | 1 | 1 0 |
| $A^{1}$ | $A$ | $0 \quad 1$ |
| $A^{2}$ | $1+A$ | $1 \quad 1$ |
| $A^{3}$ | $A+A^{2}=A+1+A=1+2 A$ | $1 \quad 2$ |
| $A^{4}$ | $A+2 A^{2}=A+2+2 A=2$ | $2 \quad 0$ |
| $A^{5}$ | $2 A$ | $0 \quad 2$ |
| $A^{6}$ | $2 A^{2}=2+2 A$ | 2 2 |
| $A^{7}$ | $2 A+2 A^{2}=2 A+2+2 A=2+A$ | 2 |

## Example

An automatic way to generate the matrix whose rows are in the third column of the table above is to use the code below.

```
p = 3; m = 2;
% Use the primitive polynomial 2 + 2x + x^2 for GF(9).
prim_poly = [2 2 1];
fiel\overline{d}= gftuple([-1:p^m-2]',prim_poly,p);
```

The gftuple function is discussed in more detail in "Converting and Simplifying Element Formats" on page 8-106.

## 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 $\mathrm{GF}(9)$ is really the same as an exponential format of 0 , because $\mathrm{A}^{8}=1=\mathrm{A}^{0}$ in $\mathrm{GF}(9)$. As another example, the substitution mentioned just before the table Elements of $\mathrm{GF}(9)$ shows that the polynomial format $\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]$ is really the same as the polynomial format [11].

## 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 $=\operatorname{gfprimdf}(m, p) ;$ If $m$ and $p$ are already defined
produces the standard row-vector representation of the default minimal polynomial for $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$.
For example, the command below shows that the default primitive polynomial for $\mathrm{GF}(9)$ is $2+x+x^{2}$, not the polynomial used in "List of All Elements of a Galois Field" on page 8-105.
polyl=gfprimdf(2,3);
poly1 =
$\begin{array}{lll}2 & 1 & 1\end{array}$
To generate a list of elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ using the default primitive polynomial, use the command
field $=$ gftuple([-1:p^m-2]',m,p);

## Converting and Simplifying Element Formats

- "Converting to Simplest Polynomial Format" on page 8-107
- "Example: Generating a List of Galois Field Elements" on page 8-108
- "Converting to Simplest Exponential Format" on page 8-108


## Converting to Simplest Polynomial Format

The gftuple function produces the simplest polynomial representation of an element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$, given either an exponential representation or a polynomial representation of that element. This can be useful for generating the list of elements of $\mathrm{GF}\left(\mathrm{p}^{m}\right)$ that other functions require.

Using gftuple requires three arguments: one representing an element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$, 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; $\mathrm{c}=$ any integer | Integer m > 1 | Polynomial format of $\mathrm{A}^{\mathrm{c}}$, where A is a root of the default primitive polynomial for $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ |
| Example: tp = gftuple(6,2,3); \% c = 6 here |  |  |
| Exponential format; $\mathrm{c}=$ any integer | Vector of coefficients of primitive polynomial | Polynomial format of $\mathrm{A}^{\mathrm{c}}$, where A is a root of the given primitive polynomial |
| Example: polynomial = gfprimdf(2,3); tp = gftuple(6,polynomial,3); \% c=6 here |  |  |
| Polynomial format of any degree | Integer m > 1 | Polynomial format of degree < m , using default primitive polynomial for $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ to simplify |
| Example: tp = gftuple([0 000001$], 2,3)$; |  |  |
| Polynomial format of any degree | Vector of coefficients of primitive polynomial | Polynomial format of degree < m , using the given primitive polynomial for GF(pm) to simplify |
| Example: polynomial $=\operatorname{gfprimdf(2,3);~tp=gftuple([0~0~0~0~} 00$ 1], polynomial,3); |  |  |

The four examples that appear in the table above all produce the same vector $\mathrm{tp}=[2,1]$, but their different inputs to gftuple correspond to the lines of the table. Each example expresses the fact that $\mathrm{A}^{6}=2+\mathrm{A}$, where A is a root of the (default) primitive polynomial $2+\mathrm{x}+\mathrm{x}^{2}$ for $\mathrm{GF}\left(3^{2}\right)$.

## Example

This example shows how gfconv and gftuple combine to multiply two polynomial-format elements of $\mathrm{GF}\left(3^{4}\right)$. Initially, gfconv multiplies the two polynomials, treating the primitive element as if it were a variable. This produces a high-order polynomial, which gftuple simplifies using the polynomial equation that the primitive element satisfies. The final result is the simplest polynomial format of the product.

```
p = 3; m = 4;
a = [1 2 0 1]; b = [2 2 1 2];
notsimple = gfconv(a,b,p) % a times b, using high powers of alpha
simple = gftuple(notsimple,m,p) %Highest exponent of alpha is m-1
```

The output is below.

```
notsimple =
    2 0
simple =
    2 1 0 1
```


## Example: Generating a List of Galois Field Elements

This example applies the conversion functionality to the task of generating a matrix that lists all elements of a Galois field. A matrix that lists all field elements is an input argument in functions such as gfadd and gfmul. The variables field1 and field2 below have the format that such functions expect.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field1 = gftuple([-1:p^m-2]',m,p);
prim_poly = gfprimdf(m,p); % Or any primitive polynomial
% for GF(p^m)
field2 = gftuple([-1:p^m-2]',prim_poly,p);
```


## Converting to Simplest Exponential Format

The same function gftuple also produces the simplest exponential representation of an element of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$, 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. 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 $\mathrm{p}^{\mathrm{m}}-2$.

## Example

To recover the exponential format of the element $2+$ A that the previous section considered, use the commands below. In this case, polyformat contains redundant information, while expformat contains the desired result.

```
[polyformat, expformat] = gftuple([2 1],2,3)
polyformat =
    2 1
expformat =
```

This output appears at first to contradict the information in the table Elements of GF(9) , but in fact it does not. The table uses a different primitive element; two plus that primitive element has the polynomial and exponential formats shown below.

```
prim_poly = [2 2 1];
[polyformat2, expformat2] = gftuple([2 1],prim_poly,3)
```

The output below reflects the information in the bottom line of the table.

```
polyformat2 =
    2 1
expformat2 =
    7
```


## Arithmetic in Galois Fields

- "Section Overview" on page 8-109
- "Arithmetic in Prime Fields" on page 8-109
- "Arithmetic in Extension Fields" on page 8-110


## 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 on page 8-109 and a mode for extension fields on page 8-110.

## Arithmetic in Prime Fields

Arithmetic in $\mathrm{GF}(\mathrm{p})$ is the same as arithmetic modulo p . The functions gfadd , $\mathrm{gfmul}, \mathrm{gfsub}$, and gfdiv accept two arguments that represent elements of $G F(p)$ 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 $g f p \_a d d(a+1, b+1)$ represents the sum $a+b$ in $G F(5)$. For example, $g f p \_a d d(3,5)=1$ because $2+4$ is 1 modulo 5 .

```
p = 5;
row = 0:p-1;
table = ones(p,1)*row;
gfp_add = gfadd(table,table',p)
```

The output for this example follows.
gfp_add =

| 0 | 1 | 2 | 3 | 4 |
| :--- | :--- | :--- | :--- | :--- |
| 1 | 2 | 3 | 4 | 0 |
| 2 | 3 | 4 | 0 | 1 |

```
3 4 4 0 1 1 2
4 0
```

Other values of $p$ produce tables for different prime fields $G F(p)$. Replacing gfadd by gfmul, gfsub, or gfdiv produces a table for the corresponding arithmetic operation in $\operatorname{GF}(\mathrm{p})$.

## Arithmetic in Extension Fields

The same arithmetic functions can add elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ when $\mathrm{m}>1$, 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 8-114 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 $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ in exponential format. The third argument, which is required, lists all elements of $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$ as described in "List of All Elements of a Galois Field" on page $8-105$. The result is in exponential format.

## Example: Addition Table for GF(9)

The code below constructs an addition table for $\mathrm{GF}\left(3^{2}\right)$, using exponential formats relative to a root of the default primitive polynomial for $\mathrm{GF}(9)$. If $a$ and $b$ are between -1 and 7 , then the element $g f p m \_a d d(a+2, b+2)$ represents the sum of $A^{a}$ and $A^{b}$ in $G F(9)$. For example, $\operatorname{gfpm} \_a d d(4,6)=5$ because
$A^{2}+A^{4}=A^{5}$
Using the fourth and sixth rows of the matrix field, you can verify that

```
A}\mp@subsup{A}{}{2}+\mp@subsup{A}{}{4}=(1+2A)+(2+0A)=3+2A=0+2A=\mp@subsup{A}{}{5}\mathrm{ modulo 3 
p = 3; m = 2; % Work in GF(3^2).
field = gftuple([-1:p^m-2]',m,p); % Construct list of elements.
row = -1:p^m-2;
table = ones(p^m,1)*row;
gfpm_add = gfadd(table,table',field)
```

The output is below.

| gfpm_add $=$ |  |  |  |  |  |  |  |  |
| ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
|  |  |  |  |  |  |  |  |  |
| - Inf | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 4 | 7 | 3 | 5 | $-\operatorname{Inf}$ | 2 | 1 | 6 |
| 1 | 7 | 5 | 0 | 4 | 6 | $-\operatorname{Inf}$ | 3 | 2 |
| 2 | 3 | 0 | 6 | 1 | 5 | 7 | $-\operatorname{Inf}$ | 4 |
| 3 | 5 | 4 | 1 | 7 | 2 | 6 | 0 | $-\operatorname{Inf}$ |
| 4 | $-\operatorname{Inf}$ | 6 | 5 | 2 | 0 | 3 | 7 | 1 |
| 5 | 2 | $-\operatorname{Inf}$ | 7 | 6 | 3 | 1 | 4 | 0 |
| 6 | 1 | 3 | $-\operatorname{Inf}$ | 0 | 7 | 4 | 2 | 5 |
| 7 | 6 | 2 | 4 | $-\operatorname{Inf}$ | 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 $G F\left(p^{\wedge} m\right)$. Replacing gfadd by $\mathrm{gfmul}, \mathrm{gfsub}$, or gfdiv produces a table for the corresponding arithmetic operation in $\mathrm{GF}\left(\mathrm{p}^{\wedge} \mathrm{m}\right)$.

## Polynomials over Prime Fields

- "Section Overview" on page 8-111
- "Cosmetic Changes of Polynomials" on page 8-111
- "Polynomial Arithmetic" on page 8-111
- "Characterization of Polynomials" on page 8-112
- "Roots of Polynomials" on page 8-112


## 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 on page 8-111 ways
- Performing polynomial arithmetic on page 8-111
- Characterizing polynomials as primitive or irreducible on page 8-112
- Finding roots on page 8-112 of polynomials in a Galois field

Note The Galois field functions in this toolbox represent a polynomial over GF(p) for odd values of p as a vector that lists the coefficients in order of ascending powers of the variable. This is the opposite of the order that other MATLAB functions use.

## Cosmetic Changes of Polynomials

To display the traditionally formatted polynomial that corresponds to a row vector containing coefficients, use gfpretty. To truncate a polynomial by removing all zero-coefficient terms that have exponents higher than the degree of the polynomial, use gftrunc. For example,

```
polynom = gftrunc([1 20 394 10 0 0 29 3 0 0])
gfpretty(polynom)
```

The output is below.

```
polynom =
    1
        1+20x+394 x}+2040\mp@subsup{x}{}{3}+29\mp@subsup{x}{}{6}+3\mp@subsup{x}{}{7
```

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 $G F(3)$ is $2+2 x^{2}+x^{3}$.

```
a = gfconv([2 1 1],[1 1],3)
[quot, remd] = gfdeconv(a,[2 1 1],3)
```

The output is below.
$\mathrm{a}=$
$\begin{array}{llll}2 & 0 & 2 & 1\end{array}$
quot $=$
$1 \quad 1$
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 $G F\left(p^{m}\right)$, the gfminpol function computes its minimal polynomial over $G F(p)$.

## Example

For example, the code below reflects the irreducibility of all minimal polynomials. However, the minimal polynomial of a nonprimitive element is not a primitive polynomial.

```
p = 3; m = 4;
% Use default primitive polynomial here.
prim_poly = gfminpol(1,m,p);
ckprim = gfprimck(prim_poly,p);
% ckprim = 1, since prim_poly represents a primitive polynomial.
notprimpoly = gfminpol(5,m,p);
cknotprim = gfprimck(notprimpoly,p);
% cknotprim = 0 (irreducible but not primitive)
% since alpha^5 is not a primitive element when p = 3.
ckreducible = gfprimck([0 1 1],p);
% ckreducible = -1 since the polynomial is reducible.
```


## Roots of Polynomials

Given a polynomial over $G F(p)$, the gfroots function finds the roots of the polynomial in a suitable extension field $G F\left(p^{m}\right)$. There are two ways to tell MATLAB the degree $m$ of the extension field $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$, as shown in the following table.

## Formats for Second Argument of gfroots

| Second Argument | Represents |
| :--- | :--- |
| A positive integer | m as in $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$. MATLAB uses the default <br> primitive polynomial in its computations. |
| A row vector | A primitive polynomial for $\mathrm{GF}\left(\mathrm{p}^{\mathrm{m}}\right)$. Here m is the <br> 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 $G F(9)$ and then checks that they are indeed roots. The exponential format of elements of GF(9) is used throughout.

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
    root = rts(ii);
    rootsquared = gfmul(root,root,field);
    rootcubed = gfmul(root,rootsquared,field);
    answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
    % Recall that l is really alpha to the zero power.
    % If answer = -Inf, then the variable root represents
    % a root of the polynomial.
end
answer
```

The output shows that $\mathrm{A}^{0}$ (which equals 1 ), $\mathrm{A}^{5}$, and $\mathrm{A}^{7}$ 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.

## Interleaving

## In this section...

"Block Interleaving" on page 8-115
"Convolutional Interleaving" on page 8-119
"Selected Bibliography for Interleaving" on page 8-127

## Block Interleaving

- "Block Interleaving Features" on page 8-115
- "Improve Error Rate Using Block Interleaving in MATLAB" on page 8-116
- "Improve Error Rate Using Block Interleaving in Simulink" on page 8-117


## Block Interleaving Features

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. The interleaver's operation on a set of symbols is independent of its operation on all other sets of symbols.

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.

Each interleaver function 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.

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.

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 <br> explicitly as an input argument. |
| Algebraic interleaver | algintrlv | Derives a permutation table <br> algebraically, using the Takeshita- <br> Costello or Welch-Costas method. <br> These methods are described in [4]. |
| Helical scan interleaver | helscanintrlv | Fills a matrix with data row by row <br> and then sends the matrix contents to <br> the output in a helical fashion. |


| Type of Interleaver | Interleaver Function | Description |
| :--- | :--- | :--- |
| Matrix interleaver | matintrlv | Fills a matrix with data elements row <br> by row and then sends the matrix <br> contents to the output column by <br> column. |
| Random interleaver | randintrlv | Chooses a permutation table randomly <br> using the initial state input that you <br> provide. |

## Types of Block Interleavers

The set of block interleavers in this library includes a general interleaver/deinterleaver pair as well as several special cases. Each special-case block uses the same computational code that its more general counterpart uses, but provides an interface that is more suitable for the special case.

The Matrix Interleaver block accomplishes block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port column by column. For example, if the interleaver uses a 2-by-3 matrix to do its internal computations, then for an input of $\left[\begin{array}{llllll}1 & 2 & 3 & 4 & 5 & 6\end{array}\right]$, the block produces an output of $\left[\begin{array}{lllll}1 & 4 & 2 & 5 & 3\end{array}\right]$.

The Random Interleaver block chooses a permutation table randomly using the Initial seed parameter that you provide in the block mask. By using the same Initial seed value in the corresponding Random Deinterleaver block, you can restore the permuted symbols to their original ordering.

The Algebraic Interleaver block uses a permutation table that is algebraically derived. It supports Takeshita-Costello interleavers and Welch-Costas interleavers. These interleavers are described in [4].

## Improve Error Rate Using Block Interleaving in MATLAB

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 = randi([0 1],k*500,1); % 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 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

## Improve Error Rate Using Block Interleaving in Simulink

The following example shows how to use an interleaver to improve the error rate when the channel produces bursts of errors.


Before running the model, you must create a binary vector that simulates bursts of errors, as described in "Improve Error Rate Using Block Interleaving in Simulink" on page 8-117. The Signal From Workspace block imports this vector from the MATLAB workspace into the model, where the Logical Operator block performs an XOR of the vector with the signal.

To open the completed model, type doc_interleaver at the MATLAB command line. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Random Data Sources sublibrary of the Comm Sources library
- Check the box next to Frame-based outputs.
- Set Samples per frame to 4.
- Hamming Encoder, in the Block sublibrary of the Error Detection and Correction library. Use default parameters
- Buffer, in the Buffers sublibrary of the Signal Management library in DSP System Toolbox
- Set Output buffer size (per channel) to 84.
- Random Interleaver, in the Block sublibrary of the Interleaving library in Communications Toolbox
- Set Number of elements to 84.
- Logical Operator, in the Simulink Math Operations library
- Set Operator to XOR.
- Signal From Workspace, in the Sources library of the DSP System Toolbox product
- Set Signal to errors.
- Set Sample time to $4 / 7$.
- Set Samples per frame to 84.
- Random Deinterleaver, in the Block sublibrary of the Interleaving library in Communications Toolbox
- Set Number of elements to 84.
- Buffer, in the Buffers sublibrary of the Signal Management library in DSP System Toolbox
- Set Output buffer size (per channel) to 7 .
- Hamming Decoder, in the Block sublibrary of the Error Detection and Correction library. Use default parameters.
- Error Rate Calculation, in the Comm Sinks library
- Set Receive delay to $(4 / 7) * 84$.
- Set Computation delay to 100.
- Set Output data to Port.
- Display, in the Simulink Sinks library. Use default parameters.

On the Simulation tab, in the Simulate section, set Stop time to length (errors). The Simulate section appears on multiple tabs.

## Creating the Vector of Errors

Before running the model, use the following code to create a binary vector in the MATLAB workspace. The model uses this vector to simulate bursts of errors. The vector contains blocks of three 1 s , representing bursts of errors, at random intervals. The distance between two consecutive blocks of 1 s is a random integer between 1 and 80 .

```
errors=zeros(1,10^4);
n=1;
while n<10^4-80;
n=n+floor(79*rand(1))+3;
```

```
errors(n:n+2)=[ll 1 1];
```

end
To determine the ratio of the number of 1 s to the total number of symbols in the vector errors enter

```
sum(errors)/length(errors)
```

Your answer should be approximately $3 / 43$, or .0698 , since after each sequence of three 1 s , the expected distance to the next sequence of 1 s is 40 . Consequently, you expect to see three 1 s in 43 terms of the sequence. If there were no error correction in the model, the bit error rate would be approximately . 0698.

When you run a simulation with the model, the error rate is approximately .019 , which shows the improvement due to error correction and interleaving. You can see the effect of interleaving by deleting the Random Interleaver and Random Deinterleaver blocks from the model, connecting the lines, and running another simulation. The bit error rate is higher without interleaving because the Hamming code can only correct one error in each codeword.

## Convolutional Interleaving

- "Convolutional Interleaving Features" on page 8-119
- "Delays of Convolutional Interleavers" on page 8-120
- "Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB" on page 8-124
- "Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in Simulink" on page 8-126


## Convolutional Interleaving Features

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 $D(1), D(2), \ldots, D(N)$. The kth shift register holds $D(k)$ symbols, where $\mathrm{k}=1,2, \ldots, \mathrm{~N}$. The convolutional interleaving functions in this toolbox have input arguments that indicate the number of shift registers and the delay for each shift register.


Communications Toolbox implements convolutional interleaving functionality using Simulink blocks, System objects, and MATLAB functions.

The set of convolutional interleavers in this product 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 <br> interleaver | muxintrlv | lllows unrestricted delay values for <br> the set of shift registers. |
| Convolutional interleaver | convintrlv | The delay values for the set of shift <br> registers are nonnegative integer <br> multiples of a fixed integer that you <br> specify. |
| Helical interleaver | helintrlv | Fills an array with input symbols in a <br> helical fashion and empties the array <br> 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.


## Types of Convolutional Interleavers

The set of convolutional interleavers in this library includes a general interleaver/deinterleaver pair as well as several special cases. Each special-case block uses the same computational code that its more general counterpart uses, but provides an interface that is more suitable for the special case.

The most general block in this library is the General Multiplexed Interleaver block, which allows arbitrary delay values for the set of shift registers. To implement the preceding schematic using this block, use an Interleaver delay parameter of $[D(1) ; D(2) ; \ldots ; D(N)]$.

More specific is the Convolutional Interleaver block, in which the delay value for the kth shift register is ( k -1) times the block's Register length step parameter. The number of shift registers in this block is the value of the Rows of shift registers parameter.

Finally, the Helical Interleaver block supports a special case of convolutional interleaving that fills an array with symbols in a helical fashion and empties the array row by row. To configure this interleaver, use the Number of columns of helical array parameter to set the width of the array, and use the Group size and Helical array step size parameters to determine how symbols are placed in the array. See the reference page for the Helical Interleaver block for more details and an example.

## 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) |

## 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

Number of shift registers $\times$ Maximum delay among all shift registers
for the most general multiplexed interleaver. If your model incurs an additional delay between the interleaver output and the deinterleaver input, the restored sequence lags behind the original sequence by the sum of the additional delay and the amount in the preceding formula.

Note For proper synchronization, the delay in your model between the interleaver output and the deinterleaver input must be an integer multiple of the number of shift registers. You can use the DSP System Toolbox Delay block to adjust delays manually, if necessary.

## Convolutional Interleaver block

In the special case implemented by the Convolutional Interleaver/Convolutional Deinterleaver pair, the number of shift registers is the Rows of shift registers parameter, while the maximum delay among all shift registers is

B
$x$
where $B$ is the Register length step parameter and $N$ is the Rows of shift registers parameter.

## Helical Interleaver block

In the special case implemented by the Helical Interleaver/Helical Deinterleaver pair, the delay between the restored sequence and the original sequence is

$$
C N\left\lceil\frac{s(C-1)}{N}\right\rceil
$$

where $C$ is the Number of columns in helical array parameter, $N$ is the Group size parameter, and $s$ is the Helical array step size parameter.

## Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB

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 = randi([0 63],20,1); % Original data
nrows = 3; slope = 2; % Interleaver parameters
D = nrows*(nrows-1)*slope; % Delay of interleaver/deinterleaver pair
hInt = comm.ConvolutionalInterleaver('NumRegisters', nrows, ...
    'RegisterLengthStep', slope);
hDeint = comm.ConvolutionalDeinterleaver('NumRegisters', nrows, ...
            'RegisterLengthStep', slope);
% First approach.
x_padded = [x; zeros(D,1)]; % Pad x at the end before interleaving.
a1 = step(hInt, x_padded); % Interleave padded data.
b1 = step(hDeint, a1)
% Omit input padding and the first D symbols of the recovered data and
% compare
servecl = step(comm.ErrorRate('ReceiveDelay',D),x_padded,b1);
ser1 = servec1(1)
% Second approach.
release(hInt); release(hDeint)
a2 = step(hInt,x); % Interleave original data.
b2 = step(hDeint,a2)
% Omit the last D symbols of the original data and the first D symbols of
% the recovered data and compare.
servec2 = step(comm.ErrorRate('ReceiveDelay',D),x,b2);
ser2 = servec2(1)
```

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
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
59
42
1
28
52
54
43
8
ser2 =
0

8-123

## 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 on page 8-121. 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.

## Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB

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.
hInt = comm.MultiplexedInterleaver('Delay', delay);
hDeint = comm.MultiplexedDeinterleaver('Delay', delay);
y = step(hInt,x) % Interleave.
z = step(hDeint,y) % 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 | $\begin{aligned} & {[]} \\ & {[0]} \\ & {[00]} \end{aligned}$ |
| 2 | 2 | 0 | $\begin{aligned} & {[]} \\ & {[2]} \\ & {[0]} \end{aligned}$ |
| 3 | 3 | 0 | [] <br> [2] <br> [0 3] |
| 4 | 1 | 4 | $\left.\begin{array}{l} {[]} \\ {[2]} \\ {[0} \end{array}\right]$ |
| 5 | 2 | 2 | $\begin{aligned} & {\left[\begin{array}{l} {[]} \\ {[5]} \\ {[0} \end{array}\right]} \\ & \hline \end{aligned}$ |
| 6 | 3 | 0 | $\begin{aligned} & {[]} \\ & {[5]} \\ & {[36]} \end{aligned}$ |
| 7 | 1 | 7 | $\begin{aligned} & {\left[\begin{array}{l} {[]} \\ {[5]} \\ {[3} \end{array}\right]} \\ & \hline \end{aligned}$ |


| Current Input | Current Shift Register | Current Output | Contents of Shift <br> Registers |
| :--- | :--- | :--- | :--- |
| 8 | 2 | 5 | [] <br> $[8]$ <br> $[36]$ |
| 9 | 3 | 3 | [] <br> $[8]$ <br> $[69]$ |
| 10 | 1 | 10 | [] <br> $[8]$ <br> $[69]$ |

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 8-120.

## Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in Simulink

The example below illustrates convolutional interleaving and deinterleaving using a sequence of consecutive integers. It also illustrates the inherent delay and the effect of the interleaving blocks' initial conditions.


Open the model by typing doc_convinterleaver at the MATLAB command line. To build the model, gather and configure these blocks:

- Ramp, in the Simulink Sources library. Use default parameters.
- Zero-Order Hold, in the Simulink Discrete library. Use default parameters.
- Convolutional Interleaver
- Set Rows of shift registers to 3 .
- Set Initial conditions to [ $\left.\begin{array}{lll}-1 & -2 & -3\end{array}\right]$ '.
- Convolutional Deinterleaver
- Set Rows of shift registers to 3.
- Set Initial conditions to [ $\left.\begin{array}{lll}-1 & -2 & -3\end{array}\right]$ '.
- Two copies of To Workspace, in the Simulink Sinks library
- Set Variable name to interleaved and restored, respectively, in the two copies of this block.
- Set Save format to Array in each of the two copies of this block.

Connect the blocks as shown in the preceding figure. On the Simulation tab, in the Simulate section, set Stop time to 20. The Simulate section appears on multiple tabs. Run the simulation and execute the following command:

```
comparison = [[0:20]', interleaved, restored]
comparison =
\begin{tabular}{rrr}
0 & 0 & -1 \\
1 & -2 & -2 \\
2 & -3 & -3 \\
3 & 3 & -1 \\
4 & -2 & -2
\end{tabular}
```

| 5 | -3 | -3 |
| ---: | ---: | ---: |
| 6 | 6 | -1 |
| 7 | 1 | -2 |
| 8 | -3 | -3 |
| 9 | 9 | -1 |
| 10 | 4 | -2 |
| 11 | -3 | -3 |
| 12 | 12 | 0 |
| 13 | 7 | 1 |
| 14 | 2 | 2 |
| 15 | 15 | 3 |
| 16 | 10 | 4 |
| 17 | 5 | 5 |
| 18 | 18 | 6 |
| 19 | 13 | 7 |
| 20 | 8 | 8 |

In this output, the first column contains the original symbol sequence. The second column contains the interleaved sequence, while the third column contains the restored sequence.

The negative numbers in the interleaved and restored sequences come from the interleaving blocks' initial conditions, not from the original data. The first of the original symbols appears in the restored sequence only after a delay of 12 symbols. The delay of the interleaver-deinterleaver combination is the product of the number of shift registers (3) and the maximum delay among all shift registers (4).

For a similar example that also indicates the contents of the shift registers at each step of the process, see "Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB" on page 8-124.

## 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.

## Digital 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.

## In this section...

"Digital Modulation Features" on page 8-128
"Signals and Delays" on page 8-131
"PM Modulation" on page 8-137
"AM Modulation" on page 8-138
"CPM Modulation" on page 8-142
"Exact LLR Algorithm" on page 8-144
"Approximate LLR Algorithm" on page 8-145
"Delays in Digital Modulation" on page 8-145
"Selected Bibliography for Digital Modulation" on page 8-147

## Digital Modulation Features

- "Modulation Techniques" on page 8-128
- "Baseband and Passband Simulation" on page 8-130
- "Modulation Terminology" on page 8-131
- "Representing Digital Signals" on page 8-131


## Modulation Techniques

The Communications Toolbox supports these modulation techniques for digital data. All the methods at the far right are implemented in library blocks.


Like analog modulation, digital modulation alters a transmittable signal according to the information in a message signal. However, for digital modulation the message signal is restricted to a finite set. Modulation functions output the complex envelope of the modulated signal. Using the Communications Toolbox, you can modulate or demodulate signals using various digital modulation techniques, and plot signal constellations.

Note Unless otherwise indicated, the modulation and demodulation functions do not perform pulse shaping or filtering. See Combining Pulse Shaping and Filtering with Modulation on page 8-141 for more information about filtering.

The available methods of modulation depend on whether the input signal is analog or digital. For a list of the modulation techniques the Communications Toolbox supports see Digital Baseband Modulation, Analog Baseband Modulation, and Analog Passband Modulation.

## Accessing Digital Modulation Blocks

Open the Modulation library by double-clicking the icon in the main block library. Then open the Digital Baseband sublibrary by double-clicking its icon in the Modulation library.

The Digital Baseband library has sublibraries of its own. Open each of these sublibraries by doubleclicking the icon listed in the table below.

| Kind of Modulation | Icon in Digital Baseband Library |
| :--- | :--- |
| Amplitude modulation | AM |
| Phase modulation | PM |
| Frequency modulation | FM |
| Continuous phase modulation | CPM |
| Trellis-coded modulation | TCM |

Some digital modulation sublibraries contain blocks that implement specific modulation techniques. These specific-case modulation blocks use the same computational code that their general counterparts use, but provide an interface that is either simpler or more suitable for the specific case. This table lists general modulators along with the conditions under which is general modulator is equivalent to a specific modulator. The situation is analogous for demodulators.

## General and Specific Blocks

| General Modulator | General Modulator Conditions | Specific Modulator |
| :--- | :--- | :--- |
| General QAM Modulator <br> Baseband | Predefined constellation containing <br> $2^{K}$ points on a rectangular lattice. <br> K is the modulation order. | Rectangular QAM Modulator <br> Baseband |
| M-PSK Modulator <br> Baseband | M-ary number parameter is 2. | BPSK Modulator Baseband |
|  | M-ary number parameter is 4. | QPSK Modulator Baseband |
| M-DPSK Modulator <br> Baseband | M-ary number parameter is 2. | DBPSK Modulator Baseband |
| CPM Modulator Baseband | M-ary number parameter is 4. <br> M-ary number parameter is 2, <br> Frequency pulse shape parameter <br> is Gaussian. | DQPSK Modulator Baseband |
| M-ary number parameter is 2, <br> Frequency pulse shape parameter <br> is Rectangular, Pulse length <br> parameter is 1. | MSK Modulator Baseband |  |
| Frequency pulse shape parameter <br> is Rectangular, Pulse length <br> parameter is 1. | CPFSK Modulator Baseband |  |
| General TCM Encoder | Predefined signal constellation <br> containing 2K points on a <br> rectangular lattice. | Rectangular QAM TCM Encoder |
| Predefined signal constellation <br> containing 2 $2^{K}$ points on a circle. | M-PSK TCM Encoder |  |

Furthermore, the CPFSK Modulator Baseband block is similar to the M-FSK Modulator Baseband block, when the M-FSK block uses continuous phase transitions. However, the M-FSK features of this product differ from the CPFSK features in their mask interfaces and in the demodulator implementations.

## Baseband and 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. The Communications Toolbox supports baseband simulation for digital modulation and passband simulation for analog modulation.

## 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

$$
Y_{1}(t) \cos \left(2 \pi f_{c} t+\theta\right)-Y_{2}(t) \sin \left(2 \pi f_{c} t+\theta\right),
$$

where $f_{\mathrm{c}}$ is the carrier frequency and $\theta$ is the carrier signal's initial phase, then a baseband simulation recognizes that this equals the real part of

$$
\left[\left(Y_{1}(t)+j Y_{2}(t)\right) e^{j \theta}\right] \exp \left(j 2 \pi f_{c} t\right)
$$

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

$$
\left(Y_{1}(t)+j Y_{2}(t)\right) e^{j \theta}
$$

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.

## Modulation Terminology

Modulation is a process by which a carrier signal is altered according to information in a message signal. The carrier frequency, $F_{\mathrm{c}}$, 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, $F_{\mathrm{s}}$, 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.

## 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 $\mathrm{M}-1$. 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 $526]$ ' 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 .

## Signals and Delays

All digital modulation blocks process only discrete-time signals and use the baseband representation. The data types of inputs and outputs are depicted in the following figure.


Note If you want to separate the in-phase and quadrature components of the complex modulated signal, use the Complex to Real-Imag block in the Simulink Math Operations library.

## Integer-Valued Signals and Binary-Valued Signals

Some digital modulation blocks can accept either integer-valued or binary-valued signals. The corresponding demodulation blocks can output either integers or groups of individual bits that represent integers. This section describes how modulation blocks process integer or binary inputs; the case for demodulation blocks is the reverse. You should note that modulation blocks have an Input type parameter and that demodulation blocks have an Output type parameter.

When you set the Input type parameter to Integer, the block accepts integer values from 0 to M-1. $M$ represents the $\mathbf{M}$-ary number block parameter.

When you set the Input type parameter to Bit, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K=\log _{2}(M)$ bits
where
$K$ represents the number of bits per symbol. Since $M=2^{K}, K$ is commonly referred to as the modulation order.

The input vector length must be an integer multiple of $K$. In this configuration, the block accepts a group of $K$ bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of $K$ bits.

## Constellation Ordering (or Symbol Set Ordering)

Depending on the modulation scheme, the Constellation ordering or Symbol set ordering parameter indicates how the block maps a group of $K$ input bits to a corresponding symbol. When you set the parameter to Binary, the block maps $[u(1) u(2) \ldots u(\mathrm{~K})]$ to the integer

$$
\sum_{i=1}^{K} u(i) 2^{K-i}
$$

and assumes that this integer is the input value. $u(1)$ is the most significant bit.
If you set $M=8$, Constellation ordering (or Symbol set ordering) to Binary, and the binary input word is [1 110 ], the block converts [11 10 ] to the integer 6 . The block produces the same output when the input is 6 and the Input type parameter is Integer.

When you set Constellation ordering (or Symbol set ordering or Symbol mapping) to Gray, the block uses a Gray-coded arrangement and assigns binary inputs to points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
to the \(P^{\text {th }}\) integer.
```

The following tables show the typical Binary to Gray mapping for $M=8$.

## Binary to Gray Mapping for Bits

| Binary Code | Gray Code |
| :--- | :--- |
| 000 | 000 |
| 001 | 001 |
| 010 | 011 |
| 011 | 010 |
| 100 | 110 |
| 101 | 111 |
| 110 | 101 |
| 111 | 100 |

## Gray to Binary Mapping for Integers

| Binary Code | Gray Code |
| :--- | :--- |
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |
| 3 | 2 |
| 4 | 6 |
| 5 | 7 |
| 6 | 5 |
| 7 | 4 |

## Gray Encoding a Modulated Signal

For the PSK, DPSK, FSK, QAM, and PAM modulation types, Gray constellations are obtained by setting the symbol mapping to Gray-encoding in the corresponding modulation function or System object®.

For modulation functions, you can specify 'gray' for the symbol order input argument to obtain Grayencoded modulation.

The following example uses the qammod function with Gray-encoded symbol mapping.

```
y = [0:15];
y = de2bi(y);
M = 16;
symorder = 'gray';
xmap = qammod(y,M,symorder,'InputType','bit','PlotConstellation',true);
```



Checking the constellation plot, you can see the modulated symbols are Gray-encoded because all adjacent elements differ by only one bit.

## Upsample Signals and Rate Changes

Some digital modulation blocks can output an upsampled version of the modulated signal, while their corresponding digital demodulation blocks can accept an upsampled version of the modulated signal as input. In both cases, the Rate options parameter represents the upsampling factor, which must be a positive integer. Depending on whether the input signal is single-rate mode or multirate mode, the block either changes the signal's vector size or its sample time, as the following table indicates. Only the OQPSK blocks deviate from the information in the table, in that S is replaced by 2 S in the scaling factors.

Process Upsampled Modulated Data (Except OQPSK Method)
\(\left.\left.$$
\begin{array}{|l|l|l|}\hline \text { Computation Type } & \text { Input Status } & \text { Result } \\
\hline \text { Modulation } & \text { Single-rate processing } & \begin{array}{l}\text { Output vector length is S times } \\
\text { the number of integers or binary } \\
\text { words in the input vector. } \\
\text { Output sample time equals the } \\
\text { input sample time. }\end{array} \\
\hline \text { Modulation } & \text { Multirate processing } & \begin{array}{l}\text { Output vector is a scalar. Output } \\
\text { sample time is 1/S times the } \\
\text { input sample time. }\end{array} \\
\hline \text { Demodulation } & \text { Single-rate processing } & \begin{array}{l}\text { Number of integers or binary } \\
\text { words in the output vector is 1/S } \\
\text { times the number of samples in } \\
\text { the input vector. Output sample }\end{array} \\
\text { time equals the input sample } \\
\text { time. }\end{array}
$$ \right\rvert\, \begin{array}{l}Output signal contains one <br>
integer or one binary word. <br>
Output sample time is S times <br>

the input sample time.\end{array}\right\}\)| Furthermore, if S > 1 and the |
| :--- |
| demodulator is from the AM, |
| PM, or FM sublibrary, the |
| demodulated signal is delayed |
| by one output sample period. |
| There is no delay if S = or if |
| the demodulator is from the |
| CPM sublibrary. |$|$| Multirate processing |
| :--- |

## Illustrations of Size or Rate Changes

The following schematics illustrate how a modulator (other than OQPSK) upsamples a triplet of frame-based and sample-based integers. In both cases, the Samples per symbol parameter is 2.

Upsample Output: Single-Rate Processing


Upsample Output: Multirate Processing


The following schematics illustrate how a demodulator (other than OQPSK or one from the CPM sublibrary) processes three doubly sampled symbols using both frame-based and sample-based inputs. In both cases, the Samples per symbol parameter is 2. The sample-based schematic includes an output delay of one sample period.

Upsampled Input: Single Rate Processing


Upsampled Input: Multirate Processing


For more information on delays, see Delays in Digital Modulation on page 8-145.

## PM Modulation

## DQPSK Signal Constellation Points and Transitions

The model below plots the output of the DQPSK Modulator Baseband block. The image shows the possible transitions from each symbol in the DQPSK signal constellation to the next symbol.


To open this model enter doc_dqpsk_plot at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
- Set M-ary number to 4.
- Set Initial seed to any positive integer scalar, preferably the output of the randn function.
- Set Sample time to . 01.
- DQPSK Modulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- Complex to Real-Imag, in the Simulink Math Operations library
- XY Graph, in the Simulink Sinks library

Use the blocks' default parameters unless otherwise instructed. Connect the blocks as in the figure above. Running the model produces the following plot. The plot reflects the transitions among the eight DQPSK constellation points.


This plot illustrates $\pi / 4$-DQPSK modulation, because the default Phase offset parameter in the DQPSK Modulator Baseband block is $\mathrm{pi} / 4$. To see how the phase offset influences the signal constellation, change the Phase offset parameter in the DQPSK Modulator Baseband block to pi/8 or another value. Run the model again and observe how the plot changes.

## AM Modulation

## Rectangular QAM Modulation and Scatter Diagram

The model below uses the M-QAM Modulator Baseband block to modulate random data. After passing the symbols through a noisy channel, the model produces a scatter diagram of the noisy data. The
diagram suggests what the underlying signal constellation looks like and shows that the noise distorts the modulated signal from the constellation.


To open this model, enter doc_qam_scatter at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
- Set M-ary number to 16.
- Set Initial seed to any positive integer scalar, preferably the output of the randn function.
- Set Sample time to . 1 .
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
- Set Normalization method to Peak Power.
- AWGN Channel, in the Channels library
- Set Es/No to 20.
- Set Symbol period to .1.
- Constellation Diagram, in the Comm Sinks library
- Set Symbols to display to 160.

Connect the blocks as shown in the preceding figure. On the Simulation tab, in the Simulate section, set Stop time to 250. The Simulate section appears on multiple tabs.

Running the model produces a scatter diagram like the following one. Your plot might look somewhat different, depending on your Initial seed value in the Random Integer Generator block. Because the modulation technique is 16 -QAM, the plot shows 16 clusters of points. If there were no noise, the plot would show the 16 exact constellation points instead of clusters around the constellation points.


## Compute Symbol Error Rate

The example generates a random digital signal, modulates it, adds noise, demodulates the noisy signal, and computes the symbol error rate. The noisy, modulated data is plotted in a constellation diagram. Numerical results and plot may vary due to the random input data.

Create a random digital message and a constellation diagram System object.

```
M = 16; % Alphabet size, 16-QAM
x = randi([0 M-1],5000,1);
cpts = qammod(0:M-1,M);
constDiag = comm.ConstellationDiagram('ReferenceConstellation',cpts, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Apply 16-QAM modulation and transmit signal through an AWGN channel.

```
y = qammod(x,M);
ynoisy = awgn(y,15,'measured');
```

Demodulate ynoisy to recover the message and check the symbol error rate.
z = qamdemod(ynoisy, M);
[num, rt] $=\operatorname{symer}(x, z)$
num $=79$
$r t=0.0158$
Create constellation diagram from noisy data. The signal reference constellation has 16 precisely located points but the transmitted symbols with the noise added causes the scatter plot to have a small cluster of points scattered around each reference constellation point.

```
constDiag(ynoisy)
```



## Combine 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 16-6.

## Rectangular Pulse Shaping

Rectangular pulse shaping repeats each output from the modulator a fixed number of times to create an upsampled signal. Although it is less realistic than other kinds of pulse shaping, rectangular pulse shaping can be a first step or an exploratory step in algorithm development. If the transmitter upsamples the modulated signal, then the receiver should downsample the received signal before demodulating. The code below uses the rectpulse function for rectangular pulse shaping at the transmitter and the intdump function for downsampling at the receiver. The "integrate and dump" operation is one way to downsample the received signal.

```
% Create a random digital message and a constellation diagram System
% object.
M = 16; % Alphabet size, 16-QAM
x = randi([0 M-1],5000,1); % Message signal
Nsamp = 4; % Oversampling rate
% Apply 16-QAM modulation and rectangular pulse shaping. Transmit signal
% through an AWGN channel.
y = qammod(x,M);
ypulse = rectpulse(y,Nsamp);
ynoisy = awgn(ypulse,15,'measured');
% Downsample at the receiver.
ydownsamp = intdump(ynoisy,Nsamp);
% Demodulate to recover the message.
z = qamdemod(ydownsamp,M);
```


## CPM Modulation

## Phase Tree for Continuous Phase Modulation

This example plots a phase tree associated with a continuous phase modulation scheme. A phase tree is a diagram that superimposes many curves, each of which plots the phase of a modulated signal over time. The distinct curves result from different inputs to the modulator.

This example uses the CPM Modulator Baseband block for its numerical computations. The block is configured using a raised cosine filter pulse shape. The example also illustrates how you can use Simulink and MATLAB together. The example uses MATLAB commands to run a series of simulations with different input signals, to collect the simulation results, and to plot the full data set.

Note In contrast to this example's approach using both MATLAB and Simulink, the commcpmphasetree example produces a phase tree using a Simulink model without additional lines of MATLAB code.


Open the model by typing doc_phasetree at the MATLAB command line. To build the model, gather and configure these blocks:

- Constant, in the Simulink Commonly Used Blocks library
- Set Constant value to $s$ (which will appear in the MATLAB workspace).
- Set Sampling mode to Frame-based.
- Set Frame period to 1 .
- CPM Modulator Baseband
- Set M-ary number to 2.
- Set Modulation index to $2 / 3$.
- Set Frequency pulse shape to Raised Cosine.
- Set Pulse length to 2.
- To Workspace, in the Simulink Sinks library
- Set Variable name to $x$.
- Set Save format to Array.

Do not run the model, because the variable s is not yet defined in the MATLAB workspace. Instead, save the model to a folder on your MATLAB path, using the filename doc_phasetree.

The second step of this example is to execute the following MATLAB code:

```
% Parameters from the CPM Modulator Baseband block
M_ary_number = 2;
mōdulàtion_index = 2/3;
pulse_length = 2;
samples_per_symbol = 8;
L = 5; % Symbols to display
pmat = [];
for ip_sig = 0:(M_ary_number^L)-1
    s = de2bi(ip_sig,_L,M_ary_number,'left-msb');
    % Apply the mapping of the input symbol to the CPM
    % symbol 0 -> -(M-1), 1 -> -(M-2), etc.
    s = 2*s'+1-M_ary_number;
    sim('doc_phasetrēe', .9); % Run model to generate x.
    % Next column of pmat
    pmat(:,ip_sig+1) = unwrap(angle(x(:)));
end;
pmat = pmat/(pi*modulation_index);
t = (0:L*samples_per_symbol-1)'/samples_per_symbol;
plot(t,pmat); figure(gcf); % Plot phase tree.
```

This code defines the parameters for the CPM Modulator, applies symbol mapping, and plots the results. Each curve represents a different instance of simulating the CPM Modulator Baseband block with a distinct (constant) input signal.


## 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:

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

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

$$
L(b)=\log \left(\frac{\sum_{s \in S_{0}} e^{-\frac{1}{\sigma^{2}}\left(\left(x-s_{X}\right)^{2}+\left(y-s_{y}\right)^{2}\right)}}{\sum_{s \in S_{1}} e^{\left.-\frac{1}{\sigma^{2}}\left(x-s_{X}\right)^{2}+\left(y-s_{y}\right)^{2}\right)}}\right)
$$

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 <br> are equally probable. |
| $S_{0}$ | 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. |
| $s_{\chi}$ | In-phase coordinate of ideal symbol or constellation point. |
| $s_{y}$ | Quadrature coordinate of ideal symbol or constellation point. |


| Variable | What the Variable Represents |
| :---: | :--- |
| $\sigma^{2}$ | Noise variance of baseband signal. |
| $\sigma_{x}^{2}$ | Noise variance along in-phase axis. |
| $\sigma_{y}^{2}$ | 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 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 [8]:

$$
L(b)=-\frac{1}{\sigma^{2}}\left(\min _{s \in S_{0}}\left(\left(x-s_{x}\right)^{2}+\left(y-s_{y}\right)^{2}\right)-\min _{s \in S_{1}}\left(\left(x-s_{x}\right)^{2}+\left(y-s_{y}\right)^{2}\right)\right)
$$

## Delays in Digital Modulation

Digital modulation and demodulation blocks sometimes incur delays between their inputs and outputs, depending on their configuration and on properties of their signals. The following table lists sources of delay and the situations in which they occur.

Delays Resulting from Digital Modulation or Demodulation

| Modulation or <br> Demodulation Type | Situation in Which Delay Occurs | Amount of Delay |
| :--- | :--- | :--- |
| FM demodulator | Sample-based processing | One output period |
| All demodulators in CPM <br> sublibrary | Multirate processing, and the model uses a <br> variable-step solver or a fixed-step solver with the <br> Tasking Mode parameter set to <br> SingleTasking <br> D = Traceback length parameter | D+1 output periods |
|  | Single-rate processing, D = Traceback depth <br> parameter | D output periods |
| OQPSK demodulator | Single-rate processing | For more information, <br> see OQPSK |
| Multirate processing, and the model uses a fixed- <br> sutep solver with Tasking Mode parameter set to <br> Auto or MultiTasking. | Demodulator <br> Baseband. |  |
|  | Multirate processing processing, and the model <br> uses a variable-step solver or the Tasking Mode <br> parameter is set to SingleTasking. |  |
| All decoders in TCM <br> sublibrary | Operation mode set to Continuous, Tr $=$ <br> Traceback depth parameter, and code rate k/n | Tr*k output bits |

As a result of delays, data that enters a modulation or demodulation block at time T appears in the output at time T+delay. In particular, if your simulation computes error statistics or compares transmitted with received data, it must take the delay into account when performing such computations or comparisons.

## First Output Sample in DPSK Demodulation

In addition to the delays mentioned above, the M-DPSK, DQPSK, and DBPSK demodulators produce output whose first sample is unrelated to the input. This is related to the differential modulation technique, not the particular implementation of it.

## Example: Delays from Demodulation

Demodulation in the model below causes the demodulated signal to lag, compared to the unmodulated signal. When computing error statistics, the model accounts for the delay by setting the Error Rate Calculation block's Receive delay parameter to 0. If the Receive delay parameter had a different value, then the error rate showing at the top of the Display block would be close to $1 / 2$.


To open this model , enter doc_oqpsk_modulation_delay at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
- Set M-ary number to 4.
- Set Initial seed to any positive integer scalar.
- OQPSK Modulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- AWGN Channel, in the Channels library
- Set Es/No to 6.
- OQPSK Demodulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- Error Rate Calculation, in the Comm Sinks library
- Set Receive delay to 1 .
- Set Computation delay to 0 .
- Set Output data to Port.
- Display, in the Simulink Sinks library
- Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as shown in the preceding figure. On the Simulation tab, in the Simulate section, set Stop time to 1000. The Simulate section appears on multiple tabs. Then run the model and observe the error rate at the top of the Display block's icon. Your error rate will vary depending on your Initial seed value in the Random Integer Generator block.

## Selected Bibliography for Digital 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, PrenticeHall, 1988.
[4] Anderson, J. B., T. Aulin, and C.-E. Sundberg, Digital Phase Modulation, New York, Plenum Press, 1986.
[5] Biglieri, E., D. Divsalar, P.J. McLane, and M.K. Simon, Introduction to Trellis-Coded Modulation with Applications, New York, Macmillan, 1991.
[6] Pawula, R.F., "On M-ary DPSK Transmission Over Terrestrial and Satellite Channels," IEEE Transactions on Communications, Vol. COM-32, July 1984, pp. 752-761.
[7] Smith, J. G., "Odd-Bit Quadrature Amplitude-Shift Keying," IEEE Transactions on Communications, Vol. COM-23, March 1975, pp. 385-389.
[8] 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

## Analog Passband Modulation

In this section...<br>"Analog Modulation Features" on page 8-148<br>"Represent Signals for Analog Modulation" on page 8-148<br>"Sampling Issues in Analog Modulation" on page 8-150<br>"Filter Design Issues" on page 8-151

## Analog Modulation Features

In most communication medium, 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. This section describes how to modulate and demodulate analog signals using blocks.

Open the Modulation library by double-clicking its icon in the main Communications Toolbox block library. Then, open the Analog Passband sublibrary by double-clicking its icon in the Modulation library.

The following figure shows the modulation techniques that Communications Toolbox supports for analog signals. As the figure suggests, some categories of techniques include named special cases.


For a given modulation technique, two ways to simulate modulation techniques are called baseband and passband. This product supports passband simulation for analog modulation.

The modulation and demodulation blocks also let you control such features as the initial phase of the modulated signal and post-demodulation filtering.

## Represent Signals for Analog Modulation

Analog modulation blocks in this product process only sample-based scalar signals. The input and output of the analog modulator and demodulator are all real signals.

All analog demodulators in this product produce discrete-time, not continuous-time, output.

## Representing Analog Signals Using MATLAB

To modulate an analog signal using MATLAB, 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 $1 / F s$. 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;
$\mathrm{t}=$ [0:.1*Fs]'/Fs;
$x=[\sin (20 * p i * t), \sin (20 * p i * t+p i / 8)] ;$

## Analog Modulation with Additive White Gaussian Noise (AWGN) Using MATLAB

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+l]'/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.

## Sampling Issues in Analog Modulation

The proper simulation of analog modulation requires that the Nyquist criterion be satisfied, taking into account the signal bandwidth.

Specifically, the sample rate of the system must be greater than twice the sum of the carrier frequency and the signal bandwidth.

## Filter Design Issues

After demodulating, you might want to filter out the carrier signal. The particular filter used, such as butter, cheby1, cheby2, and ellip, can be selected on the mask of the demodulator block. Different filtering methods have different properties, and you might need to test your application with several filters before deciding which is most suitable.

## Varying Filter's Cutoff Frequency Using Simulink

In many situations, a suitable cutoff frequency is half the carrier frequency. Because the carrier frequency must be higher than the bandwidth of the message signal, a cutoff frequency chosen in this way properly filters out unwanted frequency components. If the cutoff frequency is too high, the unwanted components may not be filtered out. If the cutoff frequency is too low, it might narrow the bandwidth of the message signal.

The following example modulates a sawtooth message signal, demodulates the resulting signal using a Butterworth filter, and plots the original and recovered signals. The Butterworth filter is implemented within the SSB AM Demodulator Passband block.


To open this model , enter doc_filtercutoffs at the MATLAB command line.
This example generates the following output:


There is invariably a delay between a demodulated signal and the original received signal. Both the filter order and the filter parameters directly affect the length of this delay.

## Other Filter Cutoffs

To see the effect of a lowpass filter with a higher cutoff frequency, set the Cutoff frequency of the SSB AM Demodulator Passband block to 49, and run the simulation again. The new result is shown below. The higher cutoff frequency allows the carrier signal to interfere with the demodulated signal.


To see the effect of a lowpass filter with a lower cutoff frequency, set the Cutoff frequency of the SSB AM Demodulator Passband block to 4, and run the simulation again. The new result is shown in the following figure. The lower cutoff frequency narrows the bandwidth of the demodulated signal.


## Phase-Locked Loops

## Phase-Locked Loop Features

The Components sublibrary contains voltage-controlled oscillator (VCO) models as well as phaselocked loop (PLL) models.

For details about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" on page 8-154.

## Voltage-Controlled Oscillator Blocks

A voltage-controlled oscillator is one part of a phase-locked loop. The Continuous-Time VCO and Discrete-Time VCO blocks implement voltage-controlled oscillators. These blocks produce continuoustime and discrete-time output signals, respectively. Each block's output signal is sinusoidal, and changes its frequency in response to the amplitude variations of the input signal.

## Overview of PLL Simulation

A phase-locked loop (PLL), when used in conjunction with other components, helps synchronize the receiver. A PLL is an automatic control system that adjusts the phase of a local signal to match the phase of the received signal. The PLL design works best for narrowband signals.

A simple PLL consists of a phase detector, a loop filter, and a voltage-controlled oscillator (VCO). For example, the following figure shows how these components are arranged for an analog passband PLL. In this case, the phase detector is just a multiplier. The signal $\mathrm{e}(\mathrm{t})$ is often called the error signal.


The following table indicates the supported types of PLLs and the blocks that implement them.

## Supported PLLs in Components Library

| Type of PLL | Block |
| :--- | :--- |
| Analog passband PLL | Phase-Locked Loop |
| Analog baseband PLL | Baseband PLL |
| Linearized analog baseband PLL | Linearized Baseband PLL |
| Digital PLL using a charge pump | Charge Pump PLL |

Different PLLs use different phase detectors, filters, and VCO characteristics. Some of these attributes are built into the PLL blocks in this product, while others depend on parameters that you set in the block mask:

- You specify the filter's transfer function in the block mask using the Lowpass filter numerator and Lowpass filter denominator parameters. Each of these parameters is a vector that lists the coefficients of the respective polynomial in order of descending exponents of the variable $s$. To design a filter, you can use functions such as butter, cheby1, and cheby2 in Signal Processing Toolbox.
- You specify the key VCO characteristics in the block mask. All four PLL blocks use a VCO input sensitivity parameter. Some blocks also use VCO quiescent frequency, VCO initial phase, and VCO output amplitude parameters.
- The phase detector for each of the PLL blocks is a feature that you cannot change from the block mask.


## Implementing an Analog Baseband PLL

Unlike passband models for a phase-locked loop, a baseband model does not depend on a carrier frequency. This allows you to use a lower sampling rate in the simulation. Two blocks implement analog baseband PLLs:

- Baseband PLL
- Linearized Baseband PLL

The linearized model and the nonlinearized model differ in that the linearized model uses the approximation

$$
\sin (\Delta \theta(t)) \cong \Delta \theta(t)
$$

to simplify the computations. This approximation is close when $\Delta \theta(\mathrm{t})$ is near zero. Thus, instead of using the input signal and the VCO output signal directly, the linearized PLL model uses only their phases.

## Implementing a Digital PLL

The charge pump PLL is a classical digital PLL. Unlike the analog PLLs mentioned above, the charge pump PLL uses a sequential logic phase detector, which is also known as a digital phase detector or a phase/frequency detector.

## Selected Bibliography for Synchronization

[1] Gardner, F.M., "Charge-pump Phase-lock Loops," IEEE Trans. on Communications, Vol. 28, November 1980, pp. 1849-1858.
[2] Gardner, F.M., "Phase Accuracy of Charge Pump PLLs," IEEE Trans. on Communications, Vol. 30, October 1982, pp. 2362-2363.
[3] Gupta, S.C., "Phase Locked Loops," Proceedings of the IEEE, Vol. 63, February 1975, pp. 291-306.
[4] Lindsay, W.C. and C.M. Chie, "A Survey on Digital Phase-Locked Loops," Proceedings of the IEEE, Vol. 69, April 1981, pp. 410-431.
[5] Mengali, Umberto, and Aldo N. D'Andrea, Synchronization Techniques for Digital Receivers, New York, Plenum Press, 1997.
[6] Meyr, Heinrich, and Gerd Ascheid, Synchronization in Digital Communications, Vol. 1, New York, John Wiley \& Sons, 1990.
[7] Moeneclaey, Marc, and Geert de Jonghe, "ML-Oriented NDA Carrier Synchronization for General Rotationally Symmetric Signal Constellations," IEEE Transactions on Communications, Vol. 42, No. 8, Aug. 1994, pp. 2531-2533.
[8] Rice, Michael. Digital Communications: A Discrete-Time Approach. Upper Saddle River, NJ: Prentice Hall, 2009.

## Multiple-Input Multiple-Output (MIMO)

## In this section...

"Orthogonal Space-Time Block Codes (OSTBC)" on page 8-156
"MIMO Fading Channel" on page 8-157
"Spherical Decoding" on page 8-157
"Selected Bibliography for MIMO systems" on page 8-157
The use of Multiple-Input Multiple-Output (MIMO) techniques has revolutionized wireless communications systems with potential gains in capacity when using multiple antennas at both transmitter and receiver ends of a communications system. New techniques, which account for the extra spatial dimension, have been adopted to realize these gains in new and previously existing systems.

MIMO technology has been adopted in multiple wireless systems, including Wi-Fi, WiMAX, LTE, and LTE-Advanced.

The Communications Toolbox product offers components to model:

- OSTBC (orthogonal space-time block coding technique)
- MIMO Fading Channels
- Spherical Decoding
and demos highlighting the use of these components in applications.
For background material on the subject of MIMO systems, see the works listed in Selected Bibliography for MIMO systems on page 8-157.


## Orthogonal Space-Time Block Codes (OSTBC)

With Communications Toolbox you can model Orthogonal Space Time Block Coding (OSTBC) - a MIMO technique which offers full spatial diversity gain with extremely simple single-symbol maximum likelihood decoding $[4,6,8]$.

In Simulink, the OSTBC Encoder and OSTBC Combiner blocks, residing in the MIMO block library, implement the orthogonal space time block coding technique. These two blocks offer a variety of specific codes (with different rates) for up to 4 transmit and 8 receive antenna systems. The encoder block is used at the transmitter to map symbols to multiple antennas while the combiner block is used at the receiver to extract the soft information per symbol using the received signal and the channel state information. You access the MIMO library by double clicking the icon in the main Communications Toolbox block library. Alternatively, you can type commmimo at the MATLAB command line.

The OSTBC technique is an attractive scheme because it can achieve the full (maximum) spatial diversity order and have symbol-wise maximum-likelihood (ML) decoding. For more information pertaining to the algorithmic details and the specific codes implemented, see OSTBC Combining Algorithms on the OSTBC Combiner block help page and OSTBC Encoding Algorithms on the OSTBC Encoder block help page. Similar functionality is available in MATLAB by using the comm. OSTBCCombiner and comm. OSTBCEncoder System objects.

## MIMO Fading Channel

With Communications Toolbox you can model a MIMO fading channel using the comm.MIMOChannel System object in MATLABor the MIMO Fading Channel block in Simulink. Using them you model the fading channel characteristics of MIMO links with Rayleigh and Rician fading, and uses the Kronecker model for the spatial correlation between the links [1].

For more information, see the .

## Spherical Decoding

With Communications Toolbox you can model a sphere decoder using the comm. SphereDecoder System object in MATLABor the Sphere Decoder block in Simulink. You can use them to find the maximum-likelihood solution for a set of received symbols over a MIMO channel with any number transmit antennas and receive antennas.

## Selected Bibliography for MIMO systems

[1] C. Oestges and B. Clerckx, MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design, Academic Press, 2007.
[2] George Tsoulos, Ed., "MIMO System Technology for Wireless Communications", CRC Press, Boca Raton, FL, 2006.
[3] L. M. Correira, Ed., Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G, Academic Press, 2006.
[4] M. Jankiraman, "Space-time codes and MIMO systems", Artech House, Boston, 2004.
[5] G. J. Foschini, M. J. Gans, "On the limits of wireless communications in a fading environment when using multiple antennas", IEEE Wireless Personal Communications, Vol. 6, Mar. 1998, pp. 311-335.
[6] S. M. Alamouti, "A simple transmit diversity technique for wireless communications," IEEE Journal on Selected Areas in Communications, vol. 16, no. 8, pp. 1451-1458, Oct. 1998.
[7] V. Tarokh, N. Seshadri, and A. R. Calderbank, "Space-time codes for high data rate wireless communication: Performance analysis and code construction," IEEE Transactions on Information Theory, vol. 44, no. 2, pp. 744-765, Mar. 1998.
[8] V. Tarokh, H. Jafarkhani, and A. R. Calderbank, "Space-time block codes from orthogonal designs," IEEE Transactions on Information Theory, vol. 45, no. 5, pp. 1456-1467, Jul. 1999.
[9] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), Base Station (BS) radio transmission and reception, Release 10, 3GPP TS 36.104, v10.0.0, 2010-09.
[10] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), User Equipment (UE) radio transmission and reception, Release 10, 3GPP TS 36.101, v10.0.0, 2010-10.

## Huffman Coding

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

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

Note For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding. To learn how to use arithmetic coding, see Arithmetic Coding on page 8-9.

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 1 s with probability $0.1,2 \mathrm{~s}$ with probability 0.1 , and $3 s$ 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 example below creates such a dictionary and then show the codeword vector associated with a particular value from the data source.

## Create a Huffman Code Dictionary

This example shows how to create a Huffman code dictionary using the huffmandict function.
Create a vector of data symbols and assign a probability to each.

```
symbols = [ll 2 3];
prob = [0.1 0.1 0.8];
```

Create a Huffman code dictionary. The most probable data symbol, 3 , is associated with a one-digit codeword, while less probable data symbols are associated with two-digit codewords.

```
dict = huffmandict(symbols,prob)
dict=3\times2 cell array
    {[1]}
    {[3]} {[ 0]}
```

Display the second row of the dictionary. The output also shows that a Huffman encoder receiving the data symbol 2 substitutes the sequence 10 .

```
dict{2,:}
ans = 2
ans = 1\times2
```

10

## Create and Decode a Huffman Code

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

Generate a data sequence to encode.

```
sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
```

Define the set of data symbols and the probability associated with each element.
symbols = [llll 123$] ;$
p = [0.1 0.1 0.8];
Create the Huffman code dictionary.
dict $=$ huffmandict(symbols,p);
Encode and decode the data. Verify that the original data, sig, and the decoded data, dhsig, are identical.

```
hcode = huffmanenco(sig,dict);
dhsig = huffmandeco(hcode,dict);
isequal(sig,dhsig)
ans = logical
    1
```


## Differential Pulse Code Modulation

## In this section...

> "Section Overview" on page 8-160
"DPCM Terminology" on page 8-160
"Represent Predictors" on page 8-160
"Example: DPCM Encoding and Decoding" on page 8-161
"Optimize DPCM Parameters" on page 8-161

## Section Overview

The quantization in the section Quantizing a Signal on page 8-10 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 "Represent Partitions" on page 8-2 and "Represent Codebooks" on page 8-2, but also a predictor. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form
$y(k)=p(1) x(k-1)+p(2) \times(k-2)+\ldots+p(m-1) \times(k-m+1)+p(m) \times(k-m)$
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.

## Represent 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) +\ldots+ 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),\ldots., 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 $\mathrm{y}(\mathrm{k})=\mathrm{x}(\mathrm{k}-1)$. The code below implements this scheme. It encodes a sawtooth signal, decodes it, and plots both the original and decoded signals. The solid line is the original signal, while the dashed line is the recovered signals. The example also computes the mean square error between the original and decoded signals.

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

The output is
distor =
0.0327

| Figure 1 |
| :---: |
| File Edit |
| 78近 |



## Optimize DPCM Parameters

- "Section Overview" on page 8-162
- "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 8-162


## Section Overview

The section "Optimize Quantization Parameters" on page 8-3 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
    
## Compand a Signal

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

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

## Quantize and Compand an Exponential Signal

Quantize an exponential signal with and without companding and compare the mean square distortions.

Set the $\mu$-law parameter Mu.

```
Mu = 255;
```

Create an exponential signal and find its maximum value.

```
sig = exp(-4:0.1:4);
V = max(sig);
```

Quantize the signal using equal-length intervals. Set the partition and codebook arguments assuming six bit quantization.

```
partition = 0:2^6-1;
codebook = 0:2^6;
[~,~,distor] = quantiz(sig,partition,codebook);
```

Compress the signal using the compand function. Apply quantization and expand the quantized signal. Calculate the mean square distortion.

```
compsig = compand(sig,Mu,V,'mu/compressor');
[~,quants] = quantiz(compsig,partition,codebook);
newsig = compand(quants,Mu,max(quants),'mu/expander');
distor2 = sum((newsig-sig).^2)/length(sig);
```

Compare the mean square distortions. The output shows that the distortion is smaller when companding is used. This is because equal-length intervals are well suited to the logarithm of sig but not well suited to sig itself.

```
[distor, distor2]
ans = 1\times2
    0.5348 0.0397
```

Plot the signal and its companded version.

```
plot([sig' compsig'])
legend('Original','Companded','location','nw')
```



## Arithmetic Coding

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

The arithenco and arithdeco functions support arithmetic coding and decoding.

## Represent 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 $=[101080]$;
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];
```


## Create and Decode an Arithmetic Code

Encode and decode a sequence from a source having three symbols.
Create a sequence vector containing symbols from the set of $\{1,2,3\}$.
seq = $\begin{array}{llllllll}3 & 3 & 1 & 3 & 3 & 3 & 3 & 3\end{array} 2$ 3];
Set the counts vector to define an encoder that produces 10 ones, 20 twos, and 70 threes from a typical 100-symbol set of test data.
counts $=\left[\begin{array}{lll}10 & 20 & 70\end{array}\right] ;$
Apply the arithmetic encoder and decoder functions.

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

Verify that the decoder output matches the original input sequence.

```
isequal(seq,dseq)
ans = logical
    1
```


## See Also

## Functions

arithdeco|arithenco

## Quantization

## In this section...

"Represent Partitions" on page 8-167
"Represent Codebooks" on page 8-167
"Determine Which Interval Each Input Is In" on page 8-167
"Optimize Quantization Parameters" on page 8-168
"Quantize a Signal" on page 8-169

## Represent Partitions

Scalar quantization is a process that maps all inputs within a specified range to a common value. This process 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 on page 8-2 and a codebook on page 8-2.

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.

## Represent 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 $[0,1,3]$.

## Determine 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);

## Optimize Quantization Parameters

- "Section Overview" on page 8-168
- "Example: Optimizing Quantization Parameters" on page 8-168


## Section Overview

Quantization distorts a signal. You can reduce 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 =
\[
0.0148 \quad 0.0024
\]
```


## Quantize a Signal

- "Scalar Quantization Example 1" on page 8-169
- "Scalar Quantization Example 2" on page 8-169


## 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([-. $27-1.2$ 1.2])


## See Also

## Functions

quantiz

## OFDM Modulation

- "OFDM with User-Specified Pilot Indices" on page 9-2
- "SER Simulation for OFDM Link" on page 9-6
- "OFDM with MIMO Simulation" on page 9-8
- "Gray Coded 8-PSK" on page 9-11
- "CPM Phase Tree" on page 9-17
- "Filtered QPSK vs. MSK" on page 9-21
- "GMSK vs. MSK" on page 9-25
- "GMSK vs. MSK" on page 9-29
- "Gray Coded 8-PSK" on page 9-33
- "Soft Decision GMSK Demodulator" on page 9-38
- "16-PSK with Custom Symbol Mapping" on page 9-45
- "General QAM Modulation in AWGN Channel" on page 9-48
- "FM Modulate and Demodulate a Sinusoidal Signal" on page 9-50


## OFDM with User-Specified Pilot Indices

This example shows how to construct an orthogonal frequency division modulation (OFDM) modulator/demodulator pair and to specify their pilot indices. The OFDM modulator System object enables you to specify pilot subcarrier indices consistent with the constraints described in comm. OFDMModulator.info. In this example, for OFDM transmission over a 3 x 2 channel, pilot indices are created for each of the three transmit antennas. Additionally, the pilot indices differ between odd and even symbols.

Create an OFDM modulator object having five symbols, three transmit antennas, and length six windowing.

```
ofdmMod = comm.OFDMModulator('FFTLength',256, ...
    'NumGuardBandCarriers',[12; 11], ...
    'NumSymbols', 5, ...
    'NumTransmitAntennas', 3, ...
    'PilotInputPort',true, ...
    'Windowing', true, ...
    'WindowLength', 6);
```

Specify pilot indices for even and odd symbols for the first transmit antenna.

```
pilotIndOdd = [20; 58; 96; 145; 182; 210];
pilotIndEven = [35; 73; 111; 159; 197; 225];
pilotIndicesAnt1 = cat(2, pilotIndOdd, pilotIndEven, pilotIndOdd, ...
    pilotIndEven, pilotIndOdd);
```

Generate pilot indices for the second and third antennas based on the indices specified for the first antenna. Concatenate the indices for the three antennas and assign them to the PilotCarrierIndices property.

```
pilotIndicesAnt2 = pilotIndicesAnt1 + 5;
pilotIndicesAnt3 = pilotIndicesAnt1 - 5;
ofdmMod.PilotCarrierIndices = cat(3, pilotIndicesAnt1, pilotIndicesAnt2, pilotIndicesAnt3);
```

Create on OFDM demodulator with two receive antennas based on the existing OFDM modulator System object. Determine the data and pilot dimensions using the info function.

```
ofdmDemod = comm.OFDMDemodulator(ofdmMod);
ofdmDemod.NumReceiveAntennas = 2;
dims = info(ofdmMod)
dims = struct with fields:
    DataInputSize: [215 5 3]
    PilotInputSize: [6 5 3]
        OutputSize: [1360 3]
```

Generate data and pilot symbols for the OFDM modulator given the array sizes specified in modDim.

```
dataIn = complex(randn(dims.DataInputSize), randn(dims.DataInputSize));
pilotIn = complex(randn(dims.PilotInputSize), randn(dims.PilotInputSize));
```

Apply OFDM modulation to the data and pilots.
modOut $=$ ofdmMod(dataIn, pilotIn);
Pass the modulated data through a $3 \times 2$ random channel.

```
chanGain = complex(randn(3,2), randn(3,2));
```

chanOut $=$ modOut $*$ chanGain;

Demodulate the received data using the OFDM demodulator object.
[data0ut,pilotOut] = ofdmDemod(chan0ut);
Show the resource mapping for the three transmit antennas. The gray lines in the figure show the placement of custom nulls to avoid interference among antennas.
showResourceMapping (ofdmMod)




For the first transmit and first receive antenna pair, demonstrate that the input pilot signal matches the input pilot signal.

```
pilotCompare = abs(pilotIn(:,:,1)*chanGain(1,1)) - abs(pilotOut(:,:,1,1));
max(pilotCompare(:) < 1e-10)
ans = logical
    1
```


## SER Simulation for OFDM Link

This example shows how to perform a symbol error rate (SER) simulation of an over-the-air OFDM communication link.

A basic communications link using OFDM modulation with QPSK symbols is simulated. There is a single transmit and a single receive antenna.

Create QPSK modulator and demodulator objects.

```
qpskMod = comm.QPSKModulator;
qpskDemod = comm.QPSKDemodulator;
```

Create a default OFDM modulator and demodulator pair.

```
ofdmMod = comm.OFDMModulator;
ofdmDemod = comm.OFDMDemodulator;
```

Use the info function to determine the required input dimensions for the OFDM modulator.

```
modDim = info(ofdmMod)
modDim = struct with fields:
    DataInputSize: [53 1]
        OutputSize: [80 1]
```

Set the number of frames. Determine the number of OFDM symbols per frame from the modDim. DataInputSize array.
nFrames = 10000;
nSymbolsPerFrame = modDim.DataInputSize(1);
Create an error rate counter with a reset input port. Initialize the symbol error rate vector, SER.

```
errRate = comm.ErrorRate('ResetInputPort',true);
SER = zeros(nFrames,1);
```

Run the simulation over 10000 OFDM frames ( 530000 symbols). During loop execution, generate a random data vector with length equal to the required number of symbols per frame, Apply QPSK modulation and then apply OFDM modulation. Pass the OFDM modulated data through the AWGN channel and then apply OFDM demodulation. Demodulate the resultant QPSK data and compare it with the original data to determine the symbol error rate.

```
snr = [8 9 10 11];
meanSER = zeros(size(snr));
for j = 1:length(snr)
    for k = 1:nFrames
    % Generate random data for each OFDM frame
    data = randi([0 3],nSymbolsPerFrame,1);
    % Apply QPSK modulation
    txQPSK = qpskMod(data);
    % Apply OFDM modulation
    txSig = ofdmMod(txQPSK);
```

```
    % Pass OFDM signal through AWGN channel
    rxSig = awgn(txSig,snr(j),'measured');
    % Demodulate OFDM data
    rxQPSK = ofdmDemod(rxSig);
    % Demodulate QPSK data
    rxData = qpskDemod(rxQPSK);
    % Compute BER
    errors = errRate(data,rxData,1);
    SER(k) = errors(1);
    end
    meanSER(j) = mean(SER);
end
Display the symbol error rate (SER) curve.
semilogy(snr,meanSER)
xlabel('SNR(dB)')
ylabel('SER')
```



## OFDM with MIMO Simulation

This example shows how to use an OFDM modulator and demodulator in a simple, 2 x 2 MIMO error rate simulation. The OFDM parameters are based on the 802.11 n standard.

Create a QPSK modulator and demodulator pair.
qpskMod = comm.QPSKModulator;
qpskDemod = comm.QPSKDemodulator;
Create an OFDM modulator and demodulator pair with user-specified pilot indices, an inserted DC null, two transmit antennas, and two receive antennas. Specify pilot indices that vary across antennas.

```
ofdmMod = comm.OFDMModulator('FFTLength',128,'PilotInputPort',true,...
    'PilotCarrierIndices',cat(3,[12; 40; 54; 76; 90; 118],...
    [13; 39; 55; 75; 91; 117]),'InsertDCNull',true,...
    'NumTransmitAntennas',2);
ofdmDemod = comm.OFDMDemodulator(ofdmMod);
ofdmDemod.NumReceiveAntennas = 2;
```

Show the resource mapping of pilot subcarriers for each transmit antenna. The gray lines in the figure denote the insertion of null subcarriers to minimize pilot signal interference.
showResourceMapping(ofdmMod)



OFDM Symbols

Determine the dimensions of the OFDM modulator by using the info method.

```
ofdmModDim = info(ofdmMod);
numData = ofdmModDim.DataInputSize(1); % Number of data subcarriers
numSym = ofdmModDim.DataInputSize(2); % Number of OFDM symbols
numTxAnt = ofdmModDim.DataInputSize(3); % Number of transmit antennas
```

Generate data symbols to fill 100 OFDM frames.
nframes = 100;
data $=$ randi([0 3],nframes*numData,numSym,numTxAnt);
Apply QPSK modulation to the random symbols and reshape the resulting column vector to match the OFDM modulator requirements.

```
modData = qpskMod(data(:));
modData = reshape(modData,nframes*numData,numSym,numTxAnt);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate;
```

Simulate the OFDM system over 100 frames assuming a flat, 2x2, Rayleigh fading channel. Remove the effects of multipath fading using a simple, least squares solution, and demodulate the OFDM waveform and QPSK data. Generate error statistics by comparing the original data with the demodulated data.

```
for k = 1:nframes
    % Find row indices for kth OFDM frame
    indData = (k-1)*ofdmModDim.DataInputSize(1)+1:k*numData;
    % Generate random OFDM pilot symbols
    pilotData = complex(rand(ofdmModDim.PilotInputSize), ...
        rand(ofdmModDim.PilotInputSize));
    % Modulate QPSK symbols using OFDM
    dataOFDM = ofdmMod(modData(indData,:,:),pilotData);
    % Create flat, i.i.d., Rayleigh fading channel
    chGain = complex(randn(2,2),randn(2,2))/sqrt(2); % Random 2x2 channel
    % Pass OFDM signal through Rayleigh and AWGN channels
    receivedSignal = awgn(data0FDM*chGain,30);
    % Apply least squares solution to remove effects of fading channel
    rxSigMF = chGain.' \ receivedSignal.';
    % Demodulate OFDM data
    receivedOFDMData = ofdmDemod(rxSigMF.');
    % Demodulate QPSK data
    receivedData = qpskDemod(receivedOFDMData(:));
    % Compute error statistics
    dataTmp = data(indData,:,:);
    errors = errorRate(dataTmp(:),receivedData);
end
```

Display the error statistics.

```
fprintf('\nSymbol error rate = %d from %d errors in %d symbols\n',errors)
Symbol error rate = 9.471154e-02 from 1970 errors in 20800 symbols
```


## Gray Coded 8-PSK

This example shows a communications system with Gray-coded 8 -ary phase shift keying (8-PSK) modulation using communications related System objects. Gray coding is a technique that multilevel modulation schemes often use to minimize the bit error rate. It consists of ordering modulation symbols so that the binary representations of adjacent symbols differ by only one bit.

## In this section...

"Introduction" on page 9-11
"Initialization" on page 9-12
"Stream Processing Loop" on page 9-13
"Cleanup" on page 9-14
"Conclusions" on page 9-14

## Introduction

This example modulates data using the 8-PSK method. The data passes through an AWGN channel, and is demodulated using an 8-PSK demodulator. An error rate calculator System object measures the symbol and bit error rates.

In this communications system, the PSK Modulator System object:

- Accepts binary-valued inputs that represent integers between 0 and $M-1 . M$ is the modulation order and is equal to 8 for 8-PSK modulation.
- Maps binary representations to constellation points using Gray-coded ordering.
- Produces unit-magnitude complex phasor outputs, with evenly spaced phases between 0 and $2 \pi(M$ -1)/M.

The following table indicates the relationship between binary representations in the input and phasors in the output. The second column of the table is an intermediate representation that the System object uses in its computations.

| Modulator Input | Gray-Coded <br> Ordering | Modulator Output |
| :---: | :--- | :--- |
| 000 | 0 | $\exp (0)$ |
| 001 | 1 | $\exp (j \Pi / 4)$ |
| 010 | 3 | $\exp (j 3 \pi / 4)$ |
| 011 | 2 | $\exp (j \Pi / 2)=\exp (j 2 \pi / 4)$ |
| 100 | 7 | $\exp (j 7 \pi / 4)$ |
| 101 | 6 | $\exp (j 3 \pi / 2)=\exp (j 6 \Pi / 4)$ |
| 110 | 4 | $\exp (j \Pi)=\exp (j 4 \pi / 4)$ |
| 111 | 5 | $\exp (j 5 \pi / 4)$ |

The table below sorts the first two columns from the previous table, according to the output values. This sorting makes it clearer that there is only a 1 bit difference between neighboring symbols. In the following figure, notice that the numbers in the second column of the table appear in counterclockwise order.

| Modulator Output | Modulator Input |
| :--- | :--- |
| $\exp (0)$ | 000 |
| $\exp (j \Pi / 4)$ | 001 |
| $\exp (j \Pi / 2)=\exp (j 2 \Pi / 4)$ | 011 |
| $\exp (j 3 \Pi / 4)$ | 010 |
| $\exp (j \Pi)=\exp (j 4 \Pi / 4)$ | 110 |
| $\exp (j 5 \Pi / 4)$ | 111 |
| $\exp (j 3 \Pi / 2)=\exp (j 6 \Pi / 4)$ | 101 |
| $\exp (j 7 \Pi / 4)$ | 100 |



## Initialization

This section of the code initializes the system variables. It also creates and configures the System objects used in this example.

Set the modulation order to 8 for 8 -PSK modulation. Run the simulation until either the specified maximum number of bit errors (maxNumErrs) or the maximum number of bits (maxNumBits) is reached. This simulation iterates over a number of bit energy to noise power spectral density $\mathrm{Eb} / \mathrm{No}$ values.

```
M = 8; % Modulation order
SamplesPerFrame = 10000; % Symbols processed for each iteration of the
    % stream processing loop
% Initialize variables used to determine when to stop processing bits
maxNumErrs=100;
maxNumBits=1e8;
% Since the AWGN Channel as well as the RANDI function uses the default
% random stream, the following commands are executed so that the results
% will be repeatable, i.e. same results will be obtained for every run of
```

```
% the example. The default stream will be restored at the end of the
% example.
prevState = rng;
rng(529558);
```

Create an integer to bit converter (hInt2Bit) and a bit to integer converter (hBit2Int) System object to convert the randomly generated integer data to bits and the demodulated data bits back to integers

```
hInt2Bit = comm.IntegerToBit('BitsPerInteger',log2(M), ...
    'OutputDataType','uint8');
hBit2Int = comm.BitToInteger('BitsPerInteger',log2(M), ...
    'OutputDataType','uint8');
```

Create and configure a PSK modulator (hMod) System object to map the binary input data to an 8PSK gray coded constellation as well as a matching PSK demodulator (hDemod) System object

```
hMod = comm.PSKModulator('Modulation0rder',M, ...
    'SymbolMapping','gray', ...
    'Phase0ffset',0, ...
    'BitInput',true);
hDemod = comm.PSKDemodulator('ModulationOrder',M, ...
    'SymbolMapping','gray', ...
    'Phase0ffset',0, ...
    'BitOutput',true, ...
    'OutputDataType','uint8', ...
    'DecisionMethod','Hard decision');
```

Create an AWGN channel System object to add additive white Gaussian noise to the modulated signal. The noise method is appropriately selected so it specifies the bit energy to noise power spectral density in the stream processing loop. Because the PSK modulator generates symbols with 1 Watt of power, the signal power property of the AWGN channel is also set to 1.

```
hChan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'BitsPerSymbol',log2(M), ...
    'SignalPower',1);
```

Create a symbol error rate calculator (hSymError) and a bit error rate calculator (hBitError) System object to compare the demodulated integer and bit data with the original source data. This comparison yields symbol error and bit error statistics. The output of the error rate calculator System object is a three-element vector containing the calculated error rate, the number of errors observed, and the amount of data processed. The simulation uses the three-element vector generated by hBitError to determine when to stop the simulation.
hSymError = comm.ErrorRate;
hBitError = comm.ErrorRate;

## Stream Processing Loop

This section of the code calls the processing loop where data is gray coded, modulated, and demodulated using 8-PSK modulation. The loop simulates the communications system for Eb/No values in the range 0 dB to 12 dB in steps of 2 dB .

```
% For each Eb/No value, simulation stops when either the maximum number of
% errors (maxNumErrs) or the maximum number of bits (maxNumBits) processed
% by the bit error rate calculator System object is reached.
EbNoVec = 0:2:12; % Eb/No values to simulate
```

```
SERVec = zeros(size(EbNoVec)); % Initialize SER history
BERVec = zeros(size(EbNoVec)); % Initialize BER history
for p = 1:length(EbNoVec)
    % Reset System objects
    reset(hSymError);
    reset(hBitError);
    hChan.EbNo = EbNoVec(p);
    % Reset SER / BER for the current Eb/No value
    SER = zeros(3,1); % Symbol Error Rate
    BER = zeros(3,1); % Bit Error Rate
    while (BER(2)<maxNumErrs) && (BER(3)<maxNumBits)
        % Generate random data
        txSym = randi([0 M-1], SamplesPerFrame, 1, 'uint8');
        txBits = step(hInt2Bit, txSym); % Convert symbols to bits
        tx = step(hMod, txBits); %Modulate
        rx = step(hChan, tx); % Add white Gaussian noise
        rxBits = step(hDemod, rx); % Demodulate
        rxSym = step(hBit2Int, rxBits); %Convert bits back to symbols
            % Calculate error rate
            SER = step(hSymError, txSym, rxSym); % Symbol Error Rate
            BER = step(hBitError, txBits, rxBits); % Bit Error Rate
    end
    % Save history of SER and BER values
    SERVec(p) = SER(1);
    BERVec(p) = BER(1);
end
```


## Cleanup

Restore the default stream.

```
rng(prevState)
```


## Conclusions

Analyze the data that the example produces and compare theoretical performance with simulation performance. The theoretical symbol error probability of MPSK is

$$
P_{E}(M)=\operatorname{erfc}\left(\sqrt{\frac{E_{S}}{N_{0}}} \sin \left(\frac{\pi}{M}\right)\right)
$$

where erfc is the complementary error function, $\mathrm{E}_{s} / \mathrm{N}_{0}$ is the ratio of energy in a symbol to noise power spectral density, and $M$ is the number of symbols.

To determine the bit error probability, convert the symbol error probability, $\mathrm{P}_{\mathrm{E}}$, to its bit error equivalent. There is no general formula for the symbol to bit error conversion. Nevertheless, upper and lower limits are easy to establish. The actual bit error probability, $\mathrm{P}_{\mathrm{b}}$, can be shown to be bounded by

$$
\frac{P_{E}(M)}{\log _{2} M} \leq P_{b} \leq \frac{M / 2}{M-1} P_{E}(M)
$$

The lower limit corresponds to the case where the symbols have undergone Gray coding. The upper limit corresponds to the case of pure binary coding.

The following script plots the simulated symbol error rates (SERVec) and bit error rates (BERVec) together with the theoretical symbol error and bit error probabilities.

Calculate theoretical error probabilities.
[theorBER, theorSER] = berawgn(EbNoVec, 'psk', M, 'nondiff');
Plot the results.

```
figure;
semilogy(EbNoVec,SERVec,'o', EbNoVec,BERVec,'*', ...
    EbNoVec,theorSER,'-', EbNoVec,theorBER,'-');
legend ( 'Symbol error rate', 'Bit error rate', ...
            'Theoretical Symbol error rate', 'Theoretical Bit error rate', ...
            'Location','SouthWest');
xlabel ( 'Eb/No (dB)' ); ylabel( 'Error Probability' );
title ( 'Symbol and Bit Error Probability' ); grid on;
```



As a further exercise, you can compare Gray coding with pure binary coding by modifying the PSK modulator and PSK demodulator System objects so that their constellation ordering parameters are 'Binary' instead of 'Gray'. Setting this property and re-running the simulation should generate results similar to the following:


## CPM Phase Tree

## In this section...

"Structure of the Example" on page 9-17
"Results and Displays" on page 9-18
"Exploring the Example" on page 9-20
This model shows how to use the Eye Diagram block to view the phase trajectory, phase tree, and instantaneous frequency of a CPM modulated signal.

## Structure of the Example

This example, doc_cpm_phase_tree, uses various Communications Toolbox, DSP System Toolbox, and Simulink blocks to model a baseband CPM signal.

In particular, the example model includes these blocks:

- Random Integer Generator block
- Integer to Bit Converter block
- CPM Modulator Baseband block
- Complex to Magnitude-Angle block
- Phase Unwrap block
- Zero-Order Hold block
- Discrete Transfer Fcn block
- Gain block
- Multiple copies of the Eye Diagram block



## Results and Displays

When you run the example, several Eye Diagram blocks show how the CPM signal changes over time:

- The Modulated Signal block displays the in-phase and quadrature signals. Double-click the block to open the scope. The modulated signal is easy to see in the eye diagram only when the Modulation index parameter in the CPM Modulator Baseband block is set to 0.5 . If you set the Modulation index to another value, for example 2/3, the features of the modulated signal are difficult to decipher for this more complex modulation. Unwrapping the phase and plotting it is another way to illustrate these more complex CPM modulated signals.
- The Phase Trajectory block displays the CPM phase. Double-click the block to open the scope. The Phase Trajectory block reveals that the signal phase is also difficult to view because it drifts with the data input to the modulator.
- The Phase Tree block displays the phase tree of the signal. The CPM phase is processed by a few simple blocks to make the CPM pulse shaping easier to view. This processing holds the phase at the beginning of the symbol interval and subtracts it from the signal. This resets the phase to zero every three symbols. The resulting plot shows the many phase trajectories that can be taken by the signal from any given symbol epoch.

- The Instantaneous Frequency block displays the instantaneous frequency of the signal. The CPM phase is differentiated to produce the frequency deviation of the signal. Viewing the CPM frequency signal enables you to observe the frequency deviation qualitatively, as well as make quantitative observations, such as measuring peak frequency deviation.



## Exploring the Example

To learn more about the example, try changing the following parameters in the CPM Modulator Baseband block:

- Change Pulse length to a value between 1 and 6.
- Change Frequency pulse shape to one of the other settings, such as Rectangular or Gaussian.

You can observe the effect of changing these parameters on the phase tree and instantaneous frequency of the modulated signal.

## Filtered QPSK vs. MSK

## In this section... <br> "Structure of the Example" on page 9-21 <br> "Results and Displays" on page 9-22

This model shows filtered quadrature phase shift keying (QPSK) and minimum shift keying (MSK) modulation schemes and visually compare them.

## Structure of the Example

This example, doc_qpsk_vs_msk, uses Communications Toolbox blocks to model filtered QPSK and MSK modulation.

The example model includes these blocks:

- Random Integer Generator block (for QPSK)
- Bernoulli Binary Generator block (for MSK)
- QPSK Modulator Baseband block
- MSK Modulator Baseband block
- Raised Cosine Transmit Filter block
- AWGN Channel block
- Eye Diagram block



## Results and Displays

The example uses eye diagram blocks to show the eye diagrams of filtered QPSK and MSK signals plus noise. You can observe that:

1 In filtered QPSK, the values of both the in-phase and quadrature components of the signal are permitted to change at any symbol interval.


2 However, for MSK, the symbol interval is half that for QPSK, but the in-phase and quadrature components change values in alternate symbol epochs. Therefore, the ideal sampling time for QPSK is $0.5,1.5,2.5, \ldots$, while the ideal sampling period for MSK is $0.5,1.5,2.5, \ldots$ for the inphase signal and 1, 2, 3,... for the quadrature signal.


## GMSK vs. MSK

```
In this section...
"Structure of the Example" on page 9-25
"Results and Displays" on page 9-26
```

This model shows how to visually compare Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes.

## Structure of the Example

The example model, doc_gmsk_vs_msk, includes these blocks:

- Random Integer Generator block, which provides a source of uniformly distributed random integers in the range [0, M-1], where M is the constellation size of the GMSK or MSK signal
- Unipolar to Bipolar Converter block
- GMSK Modulator Baseband block
- MSK Modulator Baseband block
- AWGN Channel block
- Eye Diagram block



## Results and Displays

The example illustrates the difference between the two modulation schemes. The Eye Diagram blocks show the eye diagrams of the noisy GMSK and MSK signals.



The eye diagrams show the similarity between the GMSK and MSK signals when you set the Pulse length of the GMSK Modulator Baseband block to 1. Setting the Pulse length to 3 or 5 enables you to view the difference that a partial response modulation can have on the eye diagram. The number of paths increases, showing that the CPM waveform depends on values of the previous symbols as well as the present symbol. You can change the pulse length to 2 or 4, but you should change the Phase offset to pi/4 for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use scopes that enable you to view the phase of the signal, as described in the "CPM Phase Tree" on page 9-17 example.

## GMSK vs. MSK

Compare, using eye diagrams, Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes.

Set the samples per symbol variable.
sps = 8;
Generate random binary data.

```
data = randi([0 1],1000,1);
```

Create GMSK and MSK modulators that accept binary inputs. Set the PulseLength property of the GMSK modulator to 1.

```
gmskMod = comm.GMSKModulator('BitInput',true,'PulseLength',1, ...
    'SamplesPerSymbol',sps);
mskMod = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',sps);
```

Modulate the data using the GMSK and MSK modulators.

```
modSigGMSK = gmskMod(data);
modSigMSK = mskMod(data);
```

Pass the modulated signals through an AWGN channel having an SNR of 30 dB .
rxSigGMSK = awgn(modSigGMSK,30);
rxSigMSK = awgn(modSigMSK,30);
Plot the eye diagrams of the noisy signals. With the GMSK pulse length set to 1 , the eye diagrams are nearly identical.

```
eyediagram(rxSigGMSK,sps,1,sps/2)
```


eyediagram(rxSigMSK,sps,1,sps/2)


Set the PulseLength property of hGMSK to 3 . Because the property is nontunable, the object must be released first.

```
release(gmskMod)
gmskMod.PulseLength = 3;
```

Generate a modulated signal using the updated GMSK modulator object and pass it through the AWGN channel.

```
modSigGMSK = gmskMod(data);
rxSigGMSK = awgn(modSigGMSK,30);
```

Plot the eye diagram of the GMSK signal. The increased pulse length results in an increase in the number of paths showing that the CPM waveform depends on values of the previous symbols as well as the present symbol.
eyediagram(rxSigGMSK,sps,1,sps/2)


Experiment by changing the PulseLength parameter of hGMSK to other values. If you set the property to an even number, you should set hGMSK. InitialPhaseOffset to pi/4 and the offset argument of the eyediagram function from sps/2 to 0 for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use scopes that display the phase of the signal, as described in the "CPM Phase Tree" on page 9-17 example.

## Gray Coded 8-PSK

```
In this section...
"Structure of the Example" on page 9-33
"Gray-Coded M-PSK Modulation" on page 9-33
"Exploring the Example" on page 9-35
"Simulation Results" on page 9-36
"Comparison with Pure Binary Coding and Theory" on page 9-37
```

This model, doc_gray_code, shows a communications link using Gray-coded 8-PSK modulation. Gray coding is a technique often used in multilevel modulation schemes to minimize the bit error rate by ordering modulation symbols so that the binary representations of adjacent symbols differ by only one bit.

## Structure of the Example

The example model includes these blocks:

- The Random Integer Generator block serves as the source, producing a sequence of integers.
- The Integer to Bit Converter block converts each integer into a corresponding binary representation.
- The AWGN Channel block adds white Gaussian noise to the modulated data.
- The M-PSK Demodulator Baseband block demodulates the corrupted data.
- The Bit to Integer Converter block converts each binary representation to a corresponding integer.
- One copy of the Error Rate Calculation block (labeled Error Rate Calculation1 in this model) compares the demodulated integer data with the original source data, yielding symbol error statistics. The output of the Error Rate Calculation block is a three-element vector containing the calculated error rate, the number of errors observed, and the amount of data processed.
- Another copy of the Error Rate Calculation library block (labeled Error Rate Calculation2 in this model) compares the demodulated binary data with the binary representations of the source data, yielding bit error statistics.


## Gray-Coded M-PSK Modulation

In this model, the M-PSK Modulator Baseband block:

- Accepts binary-valued inputs that represent integers between 0 and $M-1$, where $M$ is the alphabet size
- Maps binary representations to constellation points using a Gray-coded ordering
- Produces unit-magnitude complex phasor outputs, with evenly spaced phases between 0 and $2 \pi(M$ - 1)/M


The table indicates which binary representations in the input correspond to which phasors in the output. The second column of the table is an intermediate representation that the block uses in its computations.

| Modulator Input | Gray-Coded Ordering | Modulator Output |
| :--- | :--- | :--- |
| 000 | 0 | $\exp (0)=1$ |
| 001 | 1 | $\exp (j \Pi / 4)$ |
| 010 | 3 | $\exp (j 3 \pi / 4)$ |
| 011 | 2 | $\exp (j 2 \pi / 4)=\exp (j \pi / 2)$ |
| 100 | 7 | $\exp (j 7 \Pi / 4)$ |
| 101 | 6 | $\exp (j 6 \pi / 4)=\exp (j 3 \pi / 2)$ |
| 110 | 4 | $\exp (j 4 \pi / 4)=\exp (j \pi)$ |


| Modulator Input | Gray-Coded Ordering | Modulator Output |
| :--- | :--- | :--- |
| 111 | 5 | $\exp (j 5 \Pi / 4)$ |

The table below sorts the first two columns of the table above, according to the output values. This sorting makes it clearer that the overall effect of this subsystem is a Gray code mapping, as shown in the figure below. Notice that the numbers in the second column of the table below appear in counterclockwise order in the figure.

| Modulator Output | Modulator Input |
| :--- | :--- |
| $\exp (0)$ | 000 |
| $\exp (j \Pi / 4)$ | 001 |
| $\exp (j 2 \Pi / 4)=\exp (j \Pi / 2)$ | 011 |
| $\exp (j 3 \Pi / 4)$ | 010 |
| $\exp (j 4 \Pi / 4)=\exp (j п)$ | 110 |
| $\exp (j 5 \Pi / 4)$ | 111 |
| $\exp (j 6 \pi / 4)=\exp (j 3 \Pi / 2)$ | 101 |
| $\exp (j 7 \Pi / 4)$ | 100 |



## Exploring the Example

You can analyze the data that the example produces to compare theoretical performance with simulation performance.

The theoretical symbol error probability of MPSK is

$$
P_{E}(M)=\operatorname{erfc}\left(\sqrt{\frac{E_{S}}{N_{0}}} \sin \left(\frac{\pi}{M}\right)\right)
$$

where erfc is the complementary error function, $E_{\delta} / N_{0}$ is the ratio of energy in a symbol to noise power spectral density, and $M$ is the number of symbols.

To determine the bit error probability, the symbol error probability, $P_{E}$, needs to be converted to its bit error equivalent. There is no general formula for the symbol to bit error conversion. Upper and lower limits are nevertheless easy to establish. The actual bit error probability, $P_{b}$, can be shown to be bounded by

$$
\frac{P_{E}(M)}{\log _{2} M} \leq P_{b} \leq \frac{M / 2}{M-1} P_{E}(M)
$$

The lower limit corresponds to the case where the symbols have undergone Gray coding. The upper limit corresponds to the case of pure binary coding.

## Simulation Results

To test the Gray code modulation scheme in this model, simulate the graycode model for a range of $E_{b} / N_{0}$ values. If you want to study bit error rates but not symbol error rates, then you can use the bertool graphical user interface as described in "BERTool" on page 15-28.

The rest of this section studies both the bit and symbol error rates and hence does not use bertool.
Because increasing the value of $E_{b} / N_{0}$ lowers the number of errors produced, the length of each simulation must be increased to ensure that the statistics of the errors remain stable.

Using the sim command to run a Simulink simulation from the MATLAB command window, the following code generates data for symbol error rate and bit error rate curves. It considers $E_{b} / N_{0}$ values in the range 0 dB to 12 dB , in steps of 2 dB .

```
M = 8;
Tsym = 0.2;
BERVec = [];
SERVec = [];
EbNoVec = [0:2:12];
for n = 1:length(EbNoVec);
    EbNo = EbNoVec(n);
    sim('doc_gray_code') ;
    SERVec(n,:) = graySER;
    BERVec(n,:) = grayBER;
end;
```

After simulating for the full set of $E_{b} / N_{0}$ values, you can plot the results using these commands:

```
semilogy( EbNoVec,SERVec(:,1), 'o', EbNoVec, BERVec(:,1), '*' );
legend ( 'Symbol error rate', 'Bit error rate' );
xlabel ( 'Eb/No (dB)' ); ylabel( 'Error Probability' );
title ( 'Symbol and Bit Error Probability' );
```



## Comparison with Pure Binary Coding and Theory

As a further exercise, using data obtained from berawgn, you can plot the theoretical curves on the same axes with the simulation results. You can also compare Gray coding with pure binary coding, by modifying the M-PSK Modulator Baseband and M-PSK Demodulator Baseband blocks so that their Constellation ordering parameters are Binary instead of Gray.

## Soft Decision GMSK Demodulator

## In this section...

"Structure of the Example" on page 9-38
"The Serial GMSK Receiver" on page 9-39
"Results and Displays" on page 9-39
This model shows a system that includes convolutional coding and GMSK modulation. The receiver in this model includes two parallel paths, one that uses soft decisions and another that uses hard decisions. The model uses the bit error rates for the two paths to illustrate that the soft decision receiver performs better. This is to be expected, because soft decisions enable the system to retain more information from the demodulation operation to use in the decoding operation.

## Structure of the Example

The example model, doc_gmsk_soft_decision, transmits and receives a coded GMSK signal.
The key components are:

- A Bernoulli Binary Generator block, which generates binary numbers.
- A Convolutional Encoder block, which encodes the binary numbers using a rate $1 / 2$ convolutional code.
- A GMSK modulator section, which computes the logical difference between successive bits and modulates the result using the GMSK Modulator Baseband block.
- A GMSK soft demodulator section that implements the detector design proposed in [1], called a serial receiver. This section of the model produces a noisy bipolar signal. The section labeled Soft Decisions uses an eight-region partition in the Quantizing Encoder block to prepare for 3-bit softdecision decoding using the Viterbi Decoder block. The section labeled Hard Decisions uses a two-region partition to prepare for hard-decision Viterbi decoding. Using a two-region partition here is equivalent to having the demodulator make hard decisions. In each decoding section, a Delay block aligns codeword boundaries with frame boundaries so that the Viterbi Decoder block can decode properly. This is necessary because the combined delay of other blocks in the system is not an integer multiple of the length of a codeword.
- A pair of Error Rate Calculation blocks, as well as Display blocks that show the BER for the system with each type of decision.

The large signal-to-noise ratio in this example produces an eye diagram with large eye openings. If you decrease the SNR parameter in the AWGN Channel block, the eyes in the diagram will close more.

## Reduce ISI Using Raised Cosine Filtering

Employ raised cosine filtering to reduce inter-symbol interference (ISI) that results from a nonlinear amplifier.

Initialize a simulation variable for modulation order.
M = 16; \% Modulation order
Create square root raised cosine filter objects.

```
txfilter = comm.RaisedCosineTransmitFilter;
rxfilter = comm.RaisedCosineReceiveFilter;
```

Create a memoryless nonlinearity System object to introduce nonlinear behavior to the modulated signal. Using name-value pairs, set the Method property to Saleh model to emulate a high power amplifier.

```
hpa = comm.MemorylessNonlinearity('Method','Saleh model', ...
    'InputScaling',-10,'OutputScaling',0);
```

Generate random integers and apply 16-QAM modulation.

```
x = randi([0 M-1],1000,1);
modSig = qammod(x,M,'UnitAveragePower',true);
```

Plot the eye diagram of the modulated signal. At time 0 , there are three distinct "eyes" for 16 -QAM modulation.

```
eyediagram(modSig,2)
```



Amplify the modulated signal using hpa.

```
txSigNoFilt = hpa(modSig);
```

Plot the eye diagram of the amplified signal without RRC filtering. At time 0 , there are multiple eyes. This is a result of inter-symbol interference from the nonlinear amplifier.
eyediagram(txSigNoFilt,2)


Filter the modulated signal using the RRC transmit filter.

```
filteredSig = txfilter(modSig);
```

Release hpa and amplify the filtered signal. The release function is needed because the input signal dimensions change due to filter interpolation.

```
release(hpa)
txSig = hpa(filteredSig);
```

Filter txSig using the RRC matched receive filter.
rxSig = rxfilter(txSig);
Plot the eye diagram of the signal after the application of the receive filter. There are once again three distinct eyes as the matched RRC filters mitigate ISI.
eyediagram(rxSig,2)

Eye Diagram for In-Phase Signal


Eye Diagram for Quadrature Signal


## Find Delay for Encoded and Filtered Signal

Determine the delay for a convolutionally encoded and filtered link. Use the delay to accurately determine the number of bit errors.

Create a QPSK modulator and demodulator pair. Specify the objects to operate on bits.
qpskmod = comm.QPSKModulator('BitInput',true);
qpskdemod = comm.QPSKDemodulator('BitOutput',true);
Create a raised cosine transmit and receive filter pair.

```
txfilt = comm.RaisedCosineTransmitFilter;
```

rxfilt = comm.RaisedCosineReceiveFilter;

Create a convolutional encoder and Viterbi decoder pair.
convEnc = comm. ConvolutionalEncoder;
vitDec = comm.ViterbiDecoder('InputFormat','Hard');
Generate random binary data. Convolutionally encode the data.

```
txData = randi([0 1],1000,1);
encData = convEnc(txData);
```

Modulate the encoded data. Pass the modulated data through the raised cosine transmit filter.

```
modSig = qpskmod(encData);
```

txSig = txfilt(modSig);

Pass the filtered signal through an AWGN channel.

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

Filter and then demodulate the received signal.

```
filtSig = rxfilt(rxSig);
demodSig = qpskdemod(filtSig);
```

Decode the demodulated data.

```
rxData = vitDec(demodSig);
```

Find the delay between the transmitted and received binary data by using the finddelay function.

```
td = finddelay(txData,rxData)
td = 44
```

Confirm that the computed delay matches the expected delay, which is equal to the sum of the group delay of the matched filters and the traceback depth of the Viterbi decoder.

```
tdexpected = (txfilt.FilterSpanInSymbols + rxfilt.FilterSpanInSymbols)/2 + ...
    vitDec.TracebackDepth;
isequal(td,tdexpected)
ans = logical
    1
```

Calculate the number of bit errors by discarding the last td bits from the transmitted sequence and discarding the first $t d$ bits from the received sequence.

```
numErrors = biterr(txData(1:end-td),rxData(td+1:end))
numErrors = 0
```


## Visual Analysis

- "Constellation Visualization" on page 17-2
- "Plot Signal Constellations" on page 17-8
- "Eye Diagram Analysis" on page 17-12
- "Scatter Plots and Constellation Diagrams" on page 17-23
- "Channel Visualization" on page 17-30
- "Visualize RF Impairments" on page 17-35


## Constellation Visualization

Some linear modulator blocks provide the capability to visualize a signal constellation right from the block mask. This View Constellation feature allows you to visualize a signal constellation for specific block parameters. These blocks support constellation visualization:

- BPSK Modulator Baseband
- QPSK Modulator Baseband
- M-PSK Modulator Baseband
- M-PAM Modulator Baseband
- Rectangular QAM Modulator Baseband
- General QAM Modulator Baseband

Note To display Fixed-Point settings, you need a Fixed-Point Designer ${ }^{\text {TM }}$ user license.

Clicking View Constellation on a linear modulator block mask, plots the signal constellation using the block's mask parameters. If you set a modulator block to output single or fixed-point data types, clicking View Constellation generates two signal constellations plots overlaid on each other.

- One plot provides a reference constellation using double precision data type
- The other plot provides data whose data type selection is defined in the block mask

The title of the plot indicates the values of significant parameters. You can use the full array of MATLAB plot tools to manipulate plot figures. Selecting Inherit via back propagation for the Output Data Type generates a constellation plot with double as the Output data type.

## Observe Modulator Design Affect Signal Constellation

In this tutorial, you will make changes to the modulator block. Without actually applying the changes to the model, you will observe how these changes effect the signal constellation.

1 Open the constellation visualization tutorial model by typing doc_CVTutorialModel at the MATLAB command line.
2 Double-click the Rectangular QAM Modulator Baseband block.
3 Next, click View Constellation


The constellation plot shows that the constellation:

- Uses a 16-QAM modulation scheme
- Uses Binary constellation mapping
- Has 0 degree phase offset
- Has a minimum distance between two constellation points of 2

The constellation plot also shows that the signal has a double precision data type. Because the Input type is integer, the constellation has integer symbol mapping.
4 From the block mask, select Bit for the Input type parameter.
5 Select Gray for the Constellation ordering parameter.
6 Click View Constellation, and observe the results. Even though you did not click Apply, making these changes part of the model, the constellation plot still updates. The plot indicates gray constellation ordering using a bit representation of symbols.


7 You can overlay and compare the effect that two different data type selections have on a signal constellation. For example, you can compare the effect of changing Output data type from double to Fixed-point on the signal constellation.

To compare settings, perform the following tasks:

- Click the Data Types tab.
- Set the Output data type parameter to Fixed - point.
- Set the Output word length parameter to 16.
- Set the Set Output fraction length to parameter to Best precision.

8 Click Main tab, and then click View Constellation.


The plot overlays the fixed-point constellation on top of the double-precision constellation.
9 You can specify a block parameter value using variables defined in the MATLAB workspace. To define a variable, type $M=32$ in the MATLAB workspace.

Note The model workspace in Simulink has priority over the base workspace in MATLAB.
10 In the block mask, click the Main tab, and then type M for the M-ary number parameter. This parameter allows the block to use the variable value you defined in MATLAB workspace.
11 Click the Data Types tab and then select double for the Output data type parameter.
12 Click the Main tab. Then, click the View Constellation button and observe the results.


13 You can also use the Constellation Visualization feature while a simulation is running. Type $\mathrm{M}=16$ in the MATLAB workspace, select Integer for the Input type and click Apply.
14 Simulate the model by clicking Run in the Simulink model window.


15 While the simulation is running, click View Constellation. Compare the signal constellation to the scatter plot generated in the previous step.


16 End the simulation by clicking the Stop button in the Simulink model window.
The View Constellation feature provides full access to the MATLAB plotting capabilities, including: capturing a figure, saving a figure in multiple file formats, changing display settings, or saving files for archiving purposes. To capture a figure, select Edit > Copy Figure.

Using this tutorial, you have generated numerous constellation plots. If you close the Simulink model or delete the modulator block from the model, all the plots will close.

Tip If you capture a figure you want to archive for future use, save the figure before closing the model.
17 Close the Simulink model, and observe that all of the constellation figures also close.

## Plot Signal Constellations

## In this section...

"Create 16-PSK Constellation Diagram" on page 17-8
"Create 32-QAM Constellation Diagram" on page 17-9
"Create 8-QAM Gray Coded Constellation Diagram" on page 17-9
"Plot a Triangular Constellation for QAM" on page 17-10

## Create 16-PSK Constellation Diagram

This example shows how to plot a PSK constellation having 16 points.
Set the parameters for 16-PSK modulation with no phase offset and binary symbol mapping.

```
M = 16; %Modulation alphabet size
phOffset = 0; % Phase offset
symMap = 'binary'; % Symbol mapping (either 'binary' or 'gray')
```

Construct the modulator object.
pskModulator = comm. PSKModulator (M, phOffset,'SymbolMapping', symMap);
Plot the constellation.

```
constellation(pskModulator)
```



## Create 32-QAM Constellation Diagram

This example shows how to plot a QAM constellation having 32 points.
Use the qammod function to generate the 32-QAM symbols with binary symbol ordering.

```
M = 32;
```

data $=0: \mathrm{M}-1$;
sym = qammod(data, M,'bin');

Plot the constellation. Label the order of the constellation symbols.

```
scatterplot(sym,1,0,'b*');
for k = 1:M
    text(real(sym(k))-0.4,imag(sym(k))+0.4,num2str(data(k)));
end
axis([-6 6 -6 6])
```



## Create 8-QAM Gray Coded Constellation Diagram

Use the qammod function to generate the 8-QAM symbols with Gray symbol ordering. Note that Gray coding is the default symbol mapping for the qammod function.

```
M = 8;
data = 0:M-1;
sym = qammod(data,M);
```

Plot the constellation. Label the order of the constellation symbols.

```
scatterplot(sym,1,0,'r*');
grid on
for k = 1:M
    text(real(sym(k))-0.4,imag(sym(k))+0.4,num2str(data(k)));
end
axis([-4 4 -2 2])
```



## Plot a Triangular Constellation for QAM

This example shows how to plot a customized QAM reference constellation.
Describe the constellation.

```
inphase =[ll/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(:);
refConst = inphase + li*quadr;
```

Construct a constellation diagram System object using name-value pairs to specify the title, the axes limits, the reference marker type, and the reference marker color.
constDiagram = comm. ConstellationDiagram('Title','Customized Constellation for QAM', ...
'XLimits',[-3 3],'YLimits',[-3 3], ...
'ReferenceConstellation',refConst, ...
'ReferenceMarker','*','ReferenceColor', [0 1 0]);
Plot the customized constellation.
constDiagram(refConst)


## See Also

"Constellation Visualization" on page 17-2

## Eye Diagram Analysis

In digital communications, an eye diagram provides a visual indication of how noise might impact system performance.

Use the comm. EyeDiagram System object, eyediagram function, or Eye Diagram Scope block to examine the eye diagram of signals.

For information about constructing an eye diagram object, running a simulation, collecting data, and analyzing the simulated data, refer to the "Eye Diagram Measurements" example. The "Eye and Constellation Diagram" example covers eye diagram analysis applied to a communications system.

You can obtain the following measurements on an eye diagram:

- Amplitude Measurements
- Eye Amplitude
- Eye Crossing Amplitude
- Eye Crossing Percentage
- Eye Height
- Eye Level
- Eye SNR
- Quality Factor
- Vertical Eye Opening
- Time Measurements
- Deterministic Jitter
- Eye Crossing Time
- Eye Delay
- Eye Fall Time
- Eye Rise Time
- Eye Width
- Horizontal Eye Opening
- Peak-to-Peak Jitter
- Random Jitter
- RMS Jitter
- Total Jitter

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

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

The Total Jitter (TJ) PDF is the convolution of these two PDFs, which is composed of two Gaussian curves with variance $\sigma$ and mean values $\mu_{\mathrm{L}}$ and $\mu_{\mathrm{R}}$.


The dual-Dirac model is described in [5] in more detail. The amplitude of the two Dirac functions may not be the same. In such a case, the analyze method estimates these amplitudes, $\rho_{\mathrm{L}}$ and $\rho_{\mathrm{R}}$.

## Amplitude Measurements

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

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

## Eye Amplitude (EyeAmplitude)

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


## Eye Crossing Amplitude (EyeCrossingLevel)

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


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


## Eye Crossing Percentage (EyeOpeningVer)

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

## Eye Height (EyeHeight)

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

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


## Eye Level (EyeLevel)

Eye Level is the amplitude level used to represent data bits, measured in Amplitude Units (AU).
For an ideal NRZ signal, there are two eye levels: +A and -A. The analyze method calculates eye levels by estimating the mean value of the vertical histogram in a window around the EyeDelay, which is also the $50 \%$ point between eye crossing times [3]. The width of this window is determined by the EyeLevelBoundary property of the eye measurement setup object.


The analyze method calculates the mean value of all the vertical histograms within the eye level boundaries. The mean vertical histograms show two distinct PDFs, one for each eye level.


## Eye SNR (EyeSNR)

Eye signal-to-noise ratio is defined as the ratio of the eye amplitude to the sum of the standard deviations of the two eye levels. It can be expressed as:
$\mathrm{SNR}=\frac{L_{1}-L_{0}}{\sigma_{1}+\sigma_{0}}$
where $L_{1}$ and $L_{0}$ represent eye level 1 and 0 , respectively, and $\sigma_{1}$ and $\sigma_{2}$ are the standard deviation of eye level 1 and 0 , respectively.

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

## Quality Factor (QualityFactor)

The analyze method calculates Quality Factor the same way as the eye SNR. However, instead of using the mean and standard deviation values of the vertical histogram for $L_{1}$ and $\sigma_{1}$, the analyze method uses the mean and standard deviation values estimated using the dual-Dirac method. For more detail, see dual-Dirac section in [2].

## Vertical Eye Opening (EyeOpeningVer)

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

## Time Measurements

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

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

## Deterministic Jitter (JitterDeterministic)

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

DJ $=\mu_{L}-\mu_{R}$
where $\mu_{L}$ and $\mu_{R}$ are the mean values returned by the dual-Dirac algorithm.

## Eye Crossing Time (EyeCrossingTime)

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

The region is from $-A_{\text {total }} * B W$ to $+A_{\text {total }} * B W$, where $A_{\text {total }}$ is the total amplitude range of the eye diagram (i.e., $A_{\text {total }}=A_{\max }-A_{\min }$ ) and $B W$ is the crossing band width.


Because this example assumes two symbols per trace, the average PDF in this region indicate there are two crossing points.


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

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

## Eye Delay (EyeDelay)

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


## Eye Fall Time (EyeFallTime)

Eye Fall Time is the mean time between the high and low threshold values defined by the AmplitudeThreshold property of the eye measurement setup object. The fall time is calculated from $10 \%$ to $90 \%$ of the eye amplitude.


## Eye Rise Time (EyeRiseTime)

Eye Rise Time is the mean time between the low and high threshold values defined by the AmplitudeThreshold property of the eye measurement setup object. The rise time is calculated from $10 \%$ to $90 \%$ of the eye amplitude.


## Eye Width (EyeWidth)

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


## Horizontal Eye Opening (EyeOpeningHor)

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

A typical $B E R$ value for the eye opening measurements is $10^{-12}$, which approximately corresponds to the $7 \sigma$ point assuming a Gaussian distribution.

## Peak-to-Peak Jitter (JitterP2P)

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

## Random Jitter (JitterRandom)

Random Jitter is defined as the Gaussian unbounded component of the jitter. The analyze method calculates it using the tail standard deviation estimated using the dual-Dirac method as follows [5]:
$R J=\left(Q_{\mathrm{L}}+Q_{\mathrm{R}}\right) * \sigma$
where

$$
Q_{L}=\sqrt{2} * e r f c^{-1}\left(\frac{2 * B E R}{\rho_{L}}\right)
$$

and

$$
Q_{R}=\sqrt{2} * e r f c^{-1}\left(\frac{2 * B E R}{\rho_{R}}\right)
$$

$B E R$ is the bit error ratio at which the random jitter is calculated. It is defined with the BERThreshold property of the eye measurement setup object.

## RMS Jitter (JitterRMS)

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

## Total Jitter (JitterTotal)

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

## References

[1] Nelson Ou, et al, Models for the Design and Test of Gbps-Speed Serial Interconnects,IEEE Design \& Test of Computers, pp. 302-313, July-August 2004.
[2] HP E4543A Q Factor and Eye Contours Application Software, Operating Manual, http:// agilent.com
[3] Agilent 71501D Eye-Diagram Analysis, User's Guide, http://www.agilent.com
[4] 4] Guy Foster, Measurement Brief: Examining Sampling Scope Jitter Histograms, White Paper, SyntheSys Research, Inc., July 2005.
[5] Jitter Analysis: The dual-Dirac Model, RJ/DJ, and Q-Scale, White Paper, Agilent Technologies, December 2004, http://www.agilent.com

## Scatter Plots and Constellation Diagrams

A scatter plot or constellation diagram is used to visualize the constellation of a digitally modulated signal.

To produce a scatter plot from a signal, use the scatterplot function or use the comm. ConstellationDiagram System object. A scatter plot or constellation diagram can be useful when comparing system performance to a published standard, such as 3GPP or DVB.

You create the comm. ConstellationDiagram object with a default object or by defining namevalue pairs.

## View Signals Using Constellation Diagrams

This example shows how to use constellation diagrams to view QPSK transmitted and received signals which are pulse shaped with a raised cosine filter.

Create a QPSK modulator.

```
qpsk = comm.QPSKModulator;
```

Create a raised cosine transmit filter with samples per symbol, sps, equal to 16 .

```
sps = 16;
txfilter = comm.RaisedCosineTransmitFilter('Shape','Normal', ...
    'RolloffFactor',0.22, ...
    'FilterSpanInSymbols',20, ...
    'OutputSamplesPerSymbol',sps);
```

Generate data symbols, apply QPSK modulation, and pass the modulated data through the raised cosine transmit filter.

```
data = randi([0 3],200,1);
modData = qpsk(data);
txSig = txfilter(modData);
```

You can display the constellation diagram of the transmitted signal using scatterplot. Since the signal is oversampled at the filter output, you need to decimate by the number of samples per symbol so that the scatter plot does not show the transition path between constellation points. If the signal had a timing offset, you could provide that as an input parameter to display the signal constellation with the timing offset corrected.

```
scatterplot(txSig,sps)
```


## Scatter plot



Alternately, you can use comm. ConstellationDiagram, specifying the number of samples per symbol, and if needed the timing offset. Also, using comm. ConstellationDiagram the reference constellation can be shown.

Create a constellation diagram and set the SamplesPerSymbol property to the oversampling factor of the signal. Specify the constellation diagram so that it only displays the last 100 samples. This hides the zero values output by the RRC filter for the first FilterSpanInSymbols samples.

```
constDiagram = comm.ConstellationDiagram('SamplesPerSymbol',sps, ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',100);
```

Display the constellation diagram of the transmitted signal.

```
constDiagram(txSig)
```



To match the signal to its reference constellation, normalize the filter by setting its gain to the square root of the OutputSamplesPerSymbol property. This was previously specified as sps. The filter gain is nontunable so the object must be released prior to changing this value.
release(txfilter)
txfilter.Gain = sqrt(sps);
Pass the modulated signal through the normalized filter.
txSig $=$ txfilter(modData);
Display the constellation diagram of the normalized signal. The data points and reference constellation nearly overlap.
constDiagram(txSig)


To view the transmitted signal more clearly, hide the reference constellation by setting the ShowReferenceConstellation property to false.
constDiagram.ShowReferenceConstellation = false;


Create a noisy signal by passing txSig through an AWGN channel.
rxSig = awgn(txSig, 20,'measured');
Show the reference constellation and plot the received signal constellation. constDiagram.ShowReferenceConstellation = true; constDiagram(rxSig)


You can also use scatterplot to view this noisy signal but there is no built in option to add the reference constellation using scatterplot.
scatterplot(rxSig,sps)


## See Also

"Constellation Visualization" on page 17-2

## Channel Visualization

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 14-8 for a description of fading channels, objects, and blocks.

Select the desired visualization setting to plot the Impulse Response on page 17-30, Frequency Response on page 17-32, or Doppler Spectrum on page 17-33 of the channel.

## Impulse Response

The Impulse Response plot displays the path gains, the channel filter coefficients, and the interpolated path gains of the channel. The path gains shown in magenta occur at time instances that correspond to the specified path delays. These might not be aligned with the input sampling time. The channel filter coefficients shown in yellow are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. When the path gains align with the sampling time, they overlap the filter coefficients. Sinc interpolation is used to generate the blue points that appear between the channel filter coefficients. These points are used solely for display purposes and not used in subsequent channel filtering. For a flat fading channel (one path), the sinc interpolation points are not displayed. For all impulse response plots, the frame and sample numbers appear in the upper left corner of the display. The Impulse Response plot shares the same toolbar and menus as the System object it was based on, dsp. ArrayPlot.

The figure shows the impulse response of a channel in which the path gains align with the sample time. The path gains and channel filter coefficients overlap.


The next plot shows when the specified path gains are not aligned with the sample rate. Observe that the path gains and the channel filter coefficients do not overlap and that the filter coefficients are equally distributed.


The impulse response for a frequency-flat channel is shown next. Because the channel is represented by a single coefficient, no interpolation is done, and the interpolated path gains do not appear.


## Note

- The displayed and specified path gain locations can differ by as much as $5 \%$ of the input sample time.
- For MIMO, when the antenna selection property is set to any value other than Off and the specified transmit-receive pair is not selected for the current frame transmission, nothing will be displayed. Antenna selection is not applicable for SISO.
- The visualization display speed is controlled by the combination of the samples to display property and the Playback > Reduce Updates to Improve Performance menu item. Reducing the percentage of samples to display and enabling reduced updates speeds up the rendering of the impulse response.
- After the Impulse Response plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is set to Off.


## Frequency Response

The Frequency Response plot displays the channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. For the MIMO case, this transform is performed for the specified transmit-receive antenna pair. The Frequency Response plot shares the same toolbar and menus as the System object it was based on, dsp. SpectrumAnalyzer. The default settings use a rectangular window. The window length is set based on the channel model configuration. Use the View > Spectrum Settings menu to change property values from their default settings.

The frequency response plot for a frequency-selective channel is shown.


## Note

- The visualization display speed is controlled by the combination of the samples to display property and the Playback > Reduce Plot Rate to Improve Performance menu item. Reducing the
percentage of samples to display and enabling reduced updates speeds up the rendering of the frequency response.
- After the Frequency Response plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is set to Off.


## Doppler Spectrum

The Doppler Spectrum plot displays both the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a yellow line for nonstatic channels and as a yellow point for static channels. The empirical data is shown in blue. When the internal buffer is completely filled with filtered Gaussian samples, the empirical plot is updated. The empirical plot is the running mean of the spectrum calculated from each full buffer. For nonstatic channels, the number of input samples needed before the next update is displayed in the upper-left corner. The samples needed is a function of the sample rate and the maximum Doppler shift. The Doppler Spectrum plot shares the same toolbar and menus as the System object it was based on, dsp.ArrayPlot.


For static channels, the text Reset fading channel for next update is displayed.


## Note

- After the Doppler Spectrum plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is Off.


## See Also

## Blocks

MIMO Fading Channel | SISO Fading Channel

## Objects

comm.MIMOChannel

## Visualize RF Impairments

Apply various RF impairments to a QAM signal. Observe the effects by using constellation diagrams, time-varying error vector magnitude (EVM) plots, and spectrum plots. Estimate the equivalent signal-to-noise ratio (SNR).

## Initialization

Set the sample rate, modulation order, and SNR. Calculate the reference constellation points.

```
fs = 1000;
M = 16;
snrdB = 30;
refConst = qammod(0:M-1,M,'UnitAveragePower',true);
```

Create constellation diagram and time scope objects to visualize the impairment effects.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
```


## White Noise

Generate a 16-QAM signal, and pass it through an AWGN channel. Plot its constellation.

```
data = randi([0 M-1],1000,1);
modSig = qammod(data,M,'UnitAveragePower',true);
noisySig = awgn(modSig,snrdB);
constDiagram(noisySig)
```



Estimate the EVM of the noisy signal from the reference constellation points.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power');
rmsEVM = evm(noisySig)
rmsEVM =
    3.1768
```

The modulation error rate (MER) closely corresponds to the SNR. Create an MER object, and estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(noisySig)
snrEst =
    30.1071
```

The estimate is quite close to the specified SNR of 30 dB .

## Amplifier Distortion

Create an amplifier using the memoryless nonlinearity object.
amp = comm.MemorylessNonlinearity('IIP3',38,'AMPMConversion',0);
Pass the modulated signal through the nonlinear amplifier, and plot its constellation diagram.

```
txSig = amp(modSig);
constDiagram(txSig)
```



The corner points of the constellation have moved toward the origin due to amplifier gain compression.

Introduce a small AM/PM conversion, and display the received signal constellation.
amp.AMPMConversion = 1;
txSig = amp(modSig);
constDiagram(txSig)


The constellation has rotated due to the AM/PM conversion. To compute the time-varying EVM, release the EVM object and set the AveragingDimensions property to 2 . To estimate the EVM against an input signal, omit the ReferenceSignalSource property definition. This method produces more accurate results.
evm = comm.EVM('AveragingDimensions',2);
evmTime $=$ evm(modSig,txSig);
Plot the time-varying EVM of the distorted signal.
timeScope(evmTime)


Compute the RMS EVM.
evmRMS $=\operatorname{sqrt}($ mean(evmTime.^2))
evmRMS = 35.5919

Compute the MER.
mer $=$ comm.MER;
snrEst $=$ mer(modSig,txSig)
snrEst =
8.1392

The SNR ( $\approx 8 \mathrm{~dB}$ ) is reduced from its initial value $(\infty)$ due to amplifier distortion.

Specify input power levels ranging from 0 to 40 dBm . Convert those levels to their linear equivalent in W. Initialize the output power vector.

```
powerIn = 0:40;
pin = 10.^((powerIn-30)/10);
powerOut = zeros(length(powerIn),1);
```

Measure the amplifier output power for the range of input power levels.

```
for k = 1:length(powerIn)
    data = randi([0 15],1000,1);
    txSig = qammod(data,16,'UnitAveragePower',true)*sqrt(pin(k));
    ampSig = amp(txSig);
    powerOut(k) = 10*log10(var(ampSig))+30;
end
```

Plot the power output versus power input curve.
figure
plot(powerIn, power0ut, powerIn, powerIn, ' - - ')
legend('Amplifier Output','Ideal Output','location','se')
xlabel('Power In (dBm)')
ylabel('Power Out (dBm)')
grid


The output power levels off at 30 dBm . The amplifier exhibits nonlinear behavior for input power levels greater than 25 dBm .

## I/Q Imbalance

Apply an amplitude and phase imbalance to the modulated signal.

```
ampImb = 3;
phImb = 10;
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(modSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(modSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the received constellation.
constDiagram(rxSig)


The magnitude and phase of the constellation has changed as a result of the I/Q imbalance.
Calculate and plot the time-varying EVM.
evmTime $=$ evm(modSig,rxSig);
timeScope(evmTime)


The EVM exhibits a behavior that is similar to that experienced with a nonlinear amplifier though the variance is smaller.

Create a 100 Hz sine wave having a 1000 Hz sample rate.
sinewave = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
'SamplesPerFrame', 1e4, 'Complex0utput', true);
x = sinewave();
Apply the same 3 dB and 10 degree I/Q imbalance.

```
ampImb = 3;
phImb = 10;
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(x)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(x)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
y = imbI + imbQ;
```

Plot the spectrum of the imbalanced signal.

```
spectrum = dsp.SpectrumAnalyzer('SampleRate',1000,'PowerUnits','dBW');
```

spectrum(y)


The I/Q imbalance introduces a second tone at -100 Hz , which is the inverse of the input tone.
Phase Noise
Apply phase noise to the transmitted signal. Plot the resulting constellation diagram.
pnoise = comm. PhaseNoise('Level',-50,'Frequency0ffset', 20,'SampleRate',fs); pnoiseSig = pnoise(modSig);
constDiagram(pnoiseSig)


The phase noise introduces a rotational jitter.
Compute and plot the EVM of the received signal.
evmTime $=$ evm(modSig,pnoiseSig);
timeScope(evmTime)


Determine the RMS EVM.
evmRMS $=\operatorname{sqrt}($ mean(evmTime.^2))
evmRMS =

### 6.1989

## Filter Effects

Specify the samples per symbol parameter. Create a pair of raised cosine matched filters.
sps = 4;
txfilter $=$ comm. RaisedCosineTransmitFilter('RolloffFactor', 0.2,'FilterSpanInSymbols',8, ... 'OutputSamplesPerSymbol', sps, 'Gain', sqrt(sps));
rxfilter $=$ comm. RaisedCosineReceiveFilter('RolloffFactor',0.2,'FilterSpanInSymbols',8, ... 'InputSamplesPerSymbol', sps,'Gain', 1/sqrt(sps), ...
'DecimationFactor', sps);

Determine the delay through the matched filters.

```
fltDelay = 0.5*(txfilter.FilterSpanInSymbols + rxfilter.FilterSpanInSymbols);
```

Pass the modulated signal through the matched filters.

```
filtSig = txfilter(modSig);
rxSig = rxfilter(filtSig);
```

To account for the delay through the filters, discard the first fltDelay samples.

```
rxSig = rxSig(fltDelay+1:end);
```

To accommodate the change in the number of received signal samples, create new constellation diagram and time scope objects.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
```

Estimate EVM. Plot the received signal constellation diagram and the time-varying EVM.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
evmTime = evm(rxSig);
constDiagram(rxSig)
timeScope(evmTime)
```




Determine the RMS EVM.
evmRMS $=\operatorname{sqrt}($ mean (evmTime.^2) $)$
evmRMS =
2.7199

Determine the equivalent SNR.
mer $=$ comm.MER;
snrEst $=$ mer(modSig(1:end-fltDelay),rxSig)
snrEst $=$
31.4603

## Combined Effects

Combine the effects of the filters, nonlinear amplifier, AWGN, and phase noise. Display the constellation and EVM diagrams.

Create EVM, time scope and constellation diagram objects.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
timeScope = dsp.TimeScope('YLimits',[0 40],'SampleRate',fs,'TimeSpan',1, ...
    'ShowGrid',true,'YLabel','EVM (%)');
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
```

Specify the nonlinear amplifier and phase noise objects.

```
amp = comm.MemorylessNonlinearity('IIP3',45,'AMPMConversion',0);
pnoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',fs);
```

Filter and then amplify the modulated signal.

```
txfiltOut = txfilter(modSig);
txSig = amp(txfiltOut);
```

Add phase noise. Pass the impaired signal through the AWGN channel. Plot the constellation diagram.

```
rxSig = awgn(txSig,snrdB);
pnoiseSig = pnoise(rxSig);
rxfiltOut = rxfilter(pnoiseSig);
constDiagram(rxfiltOut)
```



Calculate the time-varying EVM. Plot the result.
evmTime $=$ evm(rxfiltOut);
timeScope (evmTime)


Determine the RMS EVM.
evmRMS $=\operatorname{sqrt}($ mean(evmTime.^2))
evmRMS =
6.6444

Estimate the SNR.
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
'ReferenceConstellation', refConst);
snrEst $=$ mer(rxfiltOut)
snrEst =
23.6978

This value is approximately 6 dB worse than the specified value of 30 dB , which means that the effects of the other impairments are significant and will degrade the bit error rate performance.

## See Also

Fading Channels on page 14-8 | RF Impairments on page 14-40

## C Code Generation

- "Generate C Code from MATLAB Code" on page 18-2
- "Generate C Code from Simulink Model" on page 18-9


## Generate C Code from MATLAB Code

MATLAB Coder generates highly optimized ANSI C and C++ code from functions and System objects in Communications Toolbox. You can deploy this code in a wide variety of applications. The workflow described in this topic uses DSP System Toolbox features but the same workflow applies for Communications Toolbox.

This example generates C code from the "Construct a Sinusoidal Signal Using High Energy FFT Coefficients" (DSP System Toolbox) example and builds an executable from the generated code.

Here is the MATLAB code for this example:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,\ldots.
'PhaseOffset',10,'SampleRate',44100,'Frequency',1000);
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewavel = Sineobject();
    Input = Sinewavel + 0.01*randn(size(Sinewavel));
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;
    EnergyFreqDomain = (1/L)*sum(FFTCoeffMagSq);
    [FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');
    CumFFTCoeffs = cumsum(FFTCoeffSorted);
    EnergyPercent = (CumFFTCoeffs/EnergyFreqDomain)*100;
    Vec = find(EnergyPercent > 99.99);
    FFTCoeffsModified = zeros(L,1);
    FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
    ReconstrSignal = ift(FFTCoeffsModified);
end
max(abs(Input-ReconstrSignal))
plot(Input,'*');
hold on;
plot(ReconstrSignal,'o');
hold off;
```

You can run the generated executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have MATLAB installed. You can generate code using the MATLAB Coder app or the codegen function. This example shows you the workflow using the codegen function. For more information on the app workflow, see "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder).

## Set Up the Compiler

The first step is to set up a supported C compiler. MATLAB Coder automatically locates and uses a supported installed compiler. You can change the default compiler using mex - setup. For more details, see "Change Default Compiler" (MATLAB). For a current list of supported compilers, see Supported and Compatible Compilers.

## Break Out the Computational Part of the Algorithm into a MATLAB Function

To generate C code, the entry point must be a function. You do not have to generate code for the entire MATLAB application. If you have specific portions that are computationally intensive, generate code from these portions in order to speed up your algorithm. The harness or the driver that calls this MATLAB function does not need to generate code. The harness runs in MATLAB and can contain visualization and other verification tools that are not actually part of the system under test. For
example, in the "Construct a Sinusoidal Signal Using High Energy FFT Coefficients" (DSP System Toolbox) example, the plot functions plot the input signal and the reconstructed signal. plot is not supported for code generation and must stay in the harness. To generate code from the harness that contains the visualization tools, rewrite the harness as a function and declare the visualization functions as extrinsic functions using coder.extrinsic. To run the generated code that contains the extrinsic functions, you must have MATLAB installed on your machine.

The MATLAB code in the for loop that reconstructs the original signal using high-energy FFT coefficients is the computationally intensive portion of this algorithm. Speed up the for loop by moving this computational part into a function of its own, GenerateSignalWithHighEnergyFFTCoeffs.m.

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    SampleRate',44100,'Frequency', 1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewavel + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
end
max(abs(Input-ReconstrSignal))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignal,'*')
hold off
function [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input)
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);
FFTCoeff = ft(Input);
FFTCoeffMagSq = abs(FFTCoeff).^2;
L = size(Input,1);
EnergyF = (1/L)*sum(FFTCoeffMagSq);
[FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');
CumFFTCoeffs = cumsum(FFTCoeffSorted);
EnergyPercent = (CumFFTCoeffs/EnergyF)*100;
Vec = find(EnergyPercent > 99.99);
FFTCoeffsModified = zeros(L,1);
FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
numCoeff = Vec(1);
ReconstrSignal = ift(FFTCoeffsModified);
end
```


## Make Code Suitable for Code Generation

Before you generate code, you must prepare your MATLAB code for code generation.

## Check Issues at Design Time

The first step is to eliminate unsupported constructs and check for any code generation issues. For a list of Communications Toolbox features supported by MATLAB Coder, see Functions and System Objects Supported for C Code Generation. For a list of supported language constructs, see "MATLAB Language Features Supported for C/C++ Code Generation" (MATLAB Coder).

The code analyzer detects coding issues at design time as you enter the code. To enable the code analyzer, you must add the \%\#codegen pragma to your MATLAB file.

```
Editor - H:\Documents\MATLAB\C_Code_Gen_Revamp\GenerateSignalWithHighEnergyFFTCoeffs.m
GenerateSignalWithHighEnergyFFTCoeffs.m < +
    function [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input) %#codegen
    ft = dsp.FFT('FFTImplementation','FFTW');
    ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;
    L = size(Input,1);
    EnergyF = (1/L)*sum(FFTCoeffMagSq);
```

The code generation readiness tool screens MATLAB code for features that are not supported for code generation. One of the ways to access this tool is by right-clicking on the MATLAB file in its current folder. Running the code generation tool on GenerateSignalWithHighEnergyFFTCoeffs.m finds no issues.
A Code Generation Readiness - GenerateSignalWithHighEnergyFFTCoeffs.m
Code Generation Readiness Score:
Suitable now or with minimal changes
This code is ready to use with code generation tools. A preliminary check found no issues. Some
minor problems might be found if code generation is attempted, but if so, they should be easy to fix.

## Check Issues at Code Generation Time

Before you generate $C$ code, ensure that the MATLAB code successfully generates a MEX function. The codegen command used to generate the MEX function detects any errors that prevent the code for being suitable for code generation.

Run codegen on GenerateSignalWithHighEnergyFFTCoeffs.m function.
codegen -args \{Input\} GenerateSignalWithHighEnergyFFTCoeffs
The following message appears in the MATLAB command prompt:
??? The left-hand side has been constrained to be non-complex, but the right-hand side is complex. To correct this problem, make the right-hand side real using the function
REAL, or change the initial assignment to the left-hand side variable to be a complex
value using the COMPLEX function.
Error in ==> GenerateSignalWithHighEnergy Line: 24 Column: 1
Code generation failed: View Error Report
Error using codegen

This message is referring to the variable FFTCoeffsModified. The coder is expecting this variable to be initialized as a complex variable. To resolve this issue, initialize the FFTCoeffsModified variable as complex.

```
FFTCoeffsModified = zeros(L,1)+0i;
```

Rerun the codegen function and you can see that a MEX file is generated successfully in the current folder with a .mex extension.

```
codegen -args {Input} GenerateSignalWithHighEnergyFFTCoeffs
```


## Check Issues at Run Time

Run the generated MEX function to see if there are any run-time issues reported. To do so, replace
[ReconstrSignal, numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
with
[ReconstrSignalMex, numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input);
inside the harness.
The harness now looks like:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    SampleRate',44100,'Frequency', 1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewavel + 0.01*randn(size(Sinewavel));
    [ReconstrSignalMex,numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input,L);
end
max(abs(Input-ReconstrSignalMex))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignalMex,'*')
hold off
```

The code runs successfully, indicating that there are no run-time errors.

## Compare the MEX Function with the Simulation

Notice that the harness runs much faster with the MEX function compared to the regular function. The reason for generating the MEX function is not only to detect code generation and run-time issues, but also to speed up specific parts of your algorithm. For an example, see "Signal Processing Algorithm Acceleration in MATLAB" (DSP System Toolbox).

You must also check that the numeric output results from the MEX and the regular function match. Compare the reconstructed signal generated by the GenerateSignalWithHighEnergyFFTCoeffs.m function and its MEX counterpart GenerateSignalWithHighEnergyFFTCoeffs_mex.
max(abs(ReconstrSignal-ReconstrSignalMex))
ans $=$
2.2204e-16

The results match very closely, confirming that the code generation is successful.

## Generate a Standalone Executable

If your goal is to run the generated code inside the MATLAB environment, your build target can just be a MEX function. If deployment of code to another application is the goal, then generate a standalone executable from the entire application. To do so, the harness must be a function that calls the subfunction GenerateSignalWithHighEnergyFFTCoeffs. Rewrite the harness as a function.
function reconstructSignalTestbench()
L = 1020;

```
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    SampleRate',44100,'Frequency', 1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewavel = Sineobject();
    Input = Sinewavel + 0.01*randn(size(Sinewave1));
    [ReconstrSignal, numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input,L);
end
```

Log all 1000 frames of the input and reconstructed signal and the number of FFT coefficients used to reconstruct each frame of the signal. Write all this data to a binary file named data.bin using the dsp.BinaryFileWriter System object. This example logs the number of coefficients, which are scalar values, as the first element of each frame of the input signal and the reconstructed signal. The data to be written has a frame size of $M=L+1$ and has a format that looks like this figure.
$N$ is the number of FFT coefficients that represent $99.99 \%$ of the signal energy of the current input frame. The meta data of the binary file specifies this information. Release the binary file writer and close the binary file at the end.

The updated harness function, reconstructSignalTestbench, is shown here:

```
function reconstructSignalTestbench()
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
header = struct('FirstElemInBothCols','Number of Coefficients',...
    'FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfw = dsp.BinaryFileWriter('data.bin','HeaderStructure',header);
numIter = 1000;
M = L+1;
ReSignalAll = zeros(M*numIter,1);
InputAll = zeros(M*numIter,1);
rng(1);
for Iter = 1 : numIter
    Sinewave1 = Sineobject();
    Input = Sinewavel + 0.01*randn(size(Sinewavel));
    [ReconstrSignal,numCoeffs] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
    InputAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;Input];
    ReSignalAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;ReconstrSignal];
end
bfw([InputAll ReSignalAll]);
release(bfw);
```

The next step in generating a C executable is to create a coder. config object for an executable and provide a main. c function to this object.

```
cfg = coder.config('exe');
```

cfg.CustomSource = 'reconstructSignalTestbench_Main.c';

Here is how the reconstructSignalTestbench_Main.c function looks for this example.

```
** reconstructSignalTestbench_main.c
*
* Copyright 2017 The MathWorks, Inc.
*/
#include <stdio.h>
#include <stdlib.h>
#include "reconstructSignalTestbench_initialize.h"
#include "reconstructSignalTestbench.h"
#include "reconstructSignalTestbench_terminate.h"
int main()
{
    reconstructSignalTestbench_initialize();
    reconstructSignalTestbench();
    reconstructSignalTestbench_terminate();
    return 0;
}
```

For additional details on creating the main function, see "Generating Standalone C/C++ Executables from MATLAB Code" (MATLAB Coder).

Set the CustomInclude property of the configuration object to specify the location of the main file. In this example, the location is the current folder.

```
cfg.CustomInclude = ['"',pwd,'"'];
```

Generate the $C$ executable by running the following command in the MATLAB command prompt:

```
codegen -config cfg -report reconstructSignalTestbench
```

MATLAB Coder compiles and links the main function with the C code that it generates from the reconstructSignalTestbench.m.

If you are using Windows, you can see that reconstructSignalTestbench.exe is generated in the current folder. If you are using Linux, the generated executable does not have the .exe extension.

## Read and Verify the Binary File Data

Running the executable creates a binary file, data.bin, in the current directory and writes the input, reconstructed signal, and the number of FFT coefficients used to reconstruct the signal.

```
!reconstructSignalTestbench
```

You can read this data from the binary file using the dsp.BinaryFileReader object. To verify that the data is written correctly, read data from the binary file in MATLAB and compare the output with variables InputAll and ReSignalAll.

The header prototype must have a structure similar to the header structure written to the file. Read the data as two channels.

```
M = 1021;
numIter = 1000;
headerPro = struct('FirstElemInBothCols','Number of Coefficients',...
    FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfr = dsp.BinaryFileReader('data.bin','HeaderStructure',...
```

```
headerPro,'SamplesPerFrame',M*numIter,'NumChannels',2);
Data = bfr();
```

Compare the first channel with InputAll and the second channel with ReSignalAll.

```
isequal(InputAll,Data(:,1))
ans =
    logical
    1
isequal(ReSignalAll,Data(:,2))
ans =
    logical
    1
```

The results match exactly, indicating a successful write operation.

## Relocate Code to Another Development Environment

Once you generate code from your MATLAB algorithm, you can relocate the code to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB. You can package the files into a compressed file using the packNGo function at the command line or the Package option in the MATLAB Coder app. For an example that illustrates both the workflows, see "Package Code for Other Development Environments" (MATLAB Coder). For more information on the packNGo option, see packNGo in "Build Information Methods" (MATLAB Coder). You can relocate and unpack the compressed zip file using a standard zip utility. For an example on how to package the executable generated in this example, see "Relocate Code Generated from MATLAB Code to Another Development Environment" (DSP System Toolbox).

## See Also

## Functions

codegen

## More About

- "Relocate Code Generated from MATLAB Code to Another Development Environment" (DSP System Toolbox)
- "Generate C Code from Simulink Model" (DSP System Toolbox)
- "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder)
- "Generate C Code at the Command Line" (MATLAB Coder)
- "Code Generation Workflow" (MATLAB Coder)


## External Websites

- Supported and Compatible Compilers


## Generate C Code from Simulink Model

Simulink Coder generates standalone C and C++ code from Simulink models for deployment in a wide variety of applications. The workflow described in this topic uses DSP System Toolbox features but the same workflow applies for Communications Toolbox. For a list of Communications Toolbox features supported by Simulink Coder, see Blocks Supported for C Code Generation.

This example generates $C$ code from the ex_codegen_dsp model and builds an executable from the generated code. You can run the executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have the MATLAB and Simulink products installed.

## Open the Model

The ex_codegen_dsp model implements a simple adaptive filter to remove noise from a signal while simultaneously identifying a filter that characterizes the noise frequency content. To open this model, enter the following command in MATLAB command prompt:

```
open_system('ex_codegen_dsp')
```



You can alternatively create the model using the DSP System template. For more information, see "Configure the Simulink Environment for Signal Processing Models" (DSP System Toolbox).

## Configure Model for Code Generation

Prepare the model for code generation by specifying code generation settings in the Configuration Parameters dialog box. Choose the appropriate solver and code generation target, and check the model configuration for execution efficiency. For more details on each of these steps, see "Generate C Code for a Model" (Simulink Coder).

## Simulate the Model

Simulate the model. The Time Scope shows the input and filtered signal characteristics.


The Array Plot shows the last 32 filter weights for which the LMS filter has effectively adapted and filtered out the noise from the signal.


These coefficients can also be accessed using the following command:
filter_wts(:,:,1201)

## Generate Code from the Model

Before you generate code from the model, you must first ensure that you have write permission in your current folder.

To generate code, you must make the following changes:
1 In the Modeling tab of the model toolstrip, click Model Settings. The Configuration Parameters dialog opens. Navigate to the Code Generation tab, select the Generate code only parameter, and click Apply.
2 In the Apps gallery, click Simulink Coder. The C Code tab appears. Click the Generate Code


After the model finishes generating code, the Code Generation Report appears, allowing you to inspect the generated code. Note that the build process creates a new subfolder called ex_codegen_dsp_grt_rtw in your current MATLAB working folder. This subfolder contains all the files created by the code generation process, including those that contain the generated C source
code. For more information on viewing the generated code, see "Generate C Code for a Model" (Simulink Coder).

## Build and Run the Generated Code

## Set Up the C/C++ Compiler

To build an executable, you must set up a supported C compiler. For a list of compilers supported in the current release, see Supported and Compatible Compilers.

To set up your compiler, run the following command in the MATLAB command prompt:
mex -setup

## Build the Generated Code

After your compiler is setup, you can build and run the compiled code. The ex_codegen_dsp model is currently configured to generate code only. To build the generated code, you must first make the following changes:

1 In the Modeling tab of the model toolstrip, click Model Settings. The Configuration Parameters dialog opens. Navigate to the Code Generation tab, clear the Generate code only parameter, and click Apply.

2
In the C Code tab of the model toolstrip, click the Build icon (
The code generator builds the executable and generates the Code Generation Report. The code generator places the executable in the working folder. On Windows, the executable is ex_codegen_dsp.exe. On Linux, the executable is ex_codegen_dsp.

## Run the Generated Code

To run the generated code, enter the following command in the MATLAB command prompt:

```
!ex_codegen_dsp
```

Running the generated code creates a MAT-file that contains the same variables as those generated by simulating the model. The variables in the MAT-file are named with a prefix of rt_. After you run the generated code, you can load the variables from the MAT-file by typing the following command at the MATLAB prompt:
load ex_codegen_dsp.mat
You can now compare the variables from the generated code with the variables from the model simulation. To access the last set of coefficients from the generated code, enter the following in the MATLAB prompt:

```
rt_filter_wts(:,:,1201)
```

Note that the coefficients in filter_wts(:,:,1201) and rt_filter_wts(:,:,1201) match.
For more details on building and running the executable, see "Generate C Code for a Model" (Simulink Coder).

## Relocate Code to Another Development Environment

Once you generate code from your Simulink model, you can relocate the code to another development environment using the pack-and-go utility. Use this utility when the development environment does not have the MATLAB and Simulink products.

The pack-and-go utility uses the tools for customizing the build process after code generation and a packNGo function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

You can package the code by either using the user interface or by using the command-line interface. The command-line interface provides more control over the details of code packaging. For more information on each of these methods, see "Relocate Code to Another Development Environment" (Simulink Coder).

For an example on how to package the C code and executable generated from this example, see "Relocate Code Generated from a Simulink Model to Another Development Environment" (DSP System Toolbox).

## See Also

## More About

- "Generate C Code for a Model" (Simulink Coder)
- "Relocate Code Generated from a Simulink Model to Another Development Environment" (DSP System Toolbox)
- "Relocate Code to Another Development Environment" (Simulink Coder)
- "Generate C Code from MATLAB Code" (DSP System Toolbox)
- "How To Run a Generated Executable Outside MATLAB" (DSP System Toolbox)


## External Websites

- Supported and Compatible Compilers


## HDL Code Generation

- "HDL Code Generation Support for Communications Toolbox" on page 19-2
- "Find Blocks and System Objects Supporting HDL Code Generation" on page 19-4
- "Wireless Communications Design for FPGAs and ASICs" on page 19-6


## HDL Code Generation Support for Communications Toolbox

```
In this section...
"Blocks" on page 19-2
"System Objects" on page 19-3
```


## Blocks

You can find Communications Toolbox blocks that support HDL code generation, in the 'Communications Toolbox HDL Support' library, in the Simulink library browser. Alternately, you can type commhdllib in the MATLAB command prompt to open this library. The blocks in commhdllib have their parameters set for HDL code generation.

## Error Correction

- Convolutional Encoder
- General CRC Generator HDL Optimized
- General CRC Syndrome Detector HDL Optimized
- Integer-Input RS Encoder HDL Optimized
- Integer-Output RS Decoder HDL Optimized
- Viterbi Decoder


## Filtering

- DC Blocker
- Raised Cosine Receive Filter
- Raised Cosine Transmit Filter


## Interleaving

- Convolutional Deinterleaver
- Convolutional Interleaver
- General Multiplexed Deinterleaver
- General Multiplexed Interleaver


## Modulation

- BPSK Demodulator Baseband
- BPSK Modulator Baseband
- M-PSK Demodulator Baseband
- M-PSK Modulator Baseband
- QPSK Demodulator Baseband
- QPSK Modulator Baseband
- Rectangular QAM Demodulator Baseband
- Rectangular QAM Modulator Baseband


## Sequence Generation

- PN Sequence Generator


## Sinks and Visualization

These blocks can be used for simulation visibility in models that generate HDL code, but are not included in the hardware implementation.

- Constellation Diagram
- Eye Diagram
- Error Rate Calculation


## System Objects

HDL Coder supports the following Communications Toolbox System objects for HDL code generation:

## Error Correction

- comm. HDLCRCDetector
- comm.HDLCRCGenerator
- comm.HDLRSDecoder
- comm.HDLRSEncoder
- comm.ViterbiDecoder


## Interleaving

- comm.ConvolutionalDeinterleaver
- comm.ConvolutionalInterleaver


## Modulation

- comm.BPSKDemodulator
- comm.BPSKModulator
- comm. PSKDemodulator
- comm.PSKModulator
- comm.QPSKDemodulator
- comm.QPSKModulator


## Find Blocks and System Objects Supporting HDL Code Generation

## Blocks

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the HDL Coder, Communications Toolbox HDL Support, and DSP System Toolbox HDL Support block libraries.

To create a library of HDL-supported blocks from all your installed products, enter hdllib at the MATLAB command line. This command requires an HDL Coder license.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click Blocks in the blue bar at the top of the Help window, then select the HDL code generation check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.


## System Objects

To find System objects supported for HDL code generation, see Predefined System Objects (HDL Coder).

# Wireless Communications Design for FPGAs and ASICs 

```
In this section...
"From Mathematical Algorithm to Hardware Implementation" on page 19-6
"HDL-Optimized Blocks" on page 19-8
"Reference Applications" on page 19-8
"Generate HDL Code and Prototype on FPGA" on page 19-9
```

Deploying algorithmic models to FPGA hardware makes it possible to do over-the-air testing and verification. However, designing wireless communications systems for hardware requires design tradeoffs between hardware resources and throughput. You can speed up hardware design and deployment by using HDL-optimized blocks that have hardware-suitable interfaces and architectures, reference applications that implement portions of the LTE and 5G NR physical layer, and automatic HDL code generation. You can also use hardware support packages to assist with deploying and verifying your design on real hardware.

MathWorks HDL products, such as Wireless HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from LTE Toolbox ${ }^{\mathrm{TM}}$ or 5G Toolbox ${ }^{\mathrm{TM}}$, and design a hardware implementation of that algorithm that is suitable for FPGAs and ASICs.

## From Mathematical Algorithm to Hardware Implementation

Wireless communications design often starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

Hardware engineers typically receive a mathematical specification from an algorithm team, and reimplement the algorithm for hardware. Hardware designs require tradeoffs of resource usage for clock speed and overall throughput. Usually this tradeoff means operating on streaming data, and using some logic to control the storage and flow of data. Hardware engineers usually work in hardware description languages (HDLs), like VHDL and Verilog, that provide cycle-based modeling and parallelism.

To bridge this gap between mathematical algorithm and hardware implementation, use the MATLAB algorithm model as a starting point for hardware implementation. Make incremental changes to the design to make it suitable for hardware, and progress towards a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardwarecompatible implementation in Simulink, and then the generated VHDL code.


While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing of the data stream. To aid in fixed-point type choices, it clearly visualizes data type propagation in the design. It also allows for easy pipelining of mathematical operations to improve maximum clock frequency in hardware.

While you create your hardware-ready design, use the MATLAB algorithm as a "golden reference" to verify that each version of the design still meets the mathematical requirements. The workflow shown in the diagram uses MATLAB and Simulink as collaboration and communication tools between the algorithm and hardware design teams.


For instance, when designing for LTE or 5G wireless standards, you can use LTE Toolbox and 5G Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from Wireless HDL Toolbox and blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation. You can reuse test and data generation infrastructure from MATLAB by importing data from MATLAB to your Simulink model and returning the output of the model to MATLAB to verify it against the "golden reference".


## HDL-Optimized Blocks

Library blocks from Wireless HDL Toolbox implement encoders, decoders, modulators, demodulators, and sequence generators for use in an LTE, 5G, or general wireless communications system. These blocks use a standard streaming data interface for hardware. This interface makes it easy to connect parts of the algorithm together, and includes control signals that manage the flow of data and mark frame boundaries. These blocks support automatic HDL code generation with HDL Coder. You can also use blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation.

The blocks provide hardware-suitable architectures that optimize resource use, such as including adder and multiplier pipelining to fit well into FPGA DSP slices. They also support automatic and configurable fixed-point data types. Using predefined blocks also allows you to try different parameter configurations without changing the rest of the design.

For lists of blocks that support HDL code generation, see Wireless HDL Toolbox Block List (HDL Code Generation), Communications Toolbox Block List (HDL Code Generation), and DSP System Toolbox Block List (HDL Code Generation).

## Reference Applications

Wireless HDL Toolbox provides reference applications that contain hardware-ready implementations of large parts of the LTE and 5G NR physical layer. These designs are verified against the "golden reference" functions provided by LTE Toolbox and 5G Toolbox. They have also been tested on FPGA boards to confirm that they encode and decode over-the-air waveforms and use a reasonable amount of hardware resources. They are designed to be modular, scalable, and extensible so you can insert additional physical channels. The receiver design was tested using waveforms captured off-the-air.

The suite of reference applications includes:

- LTE and 5G NR primary and secondary synchronization signal (PSS/SSS) generation and detection
- LTE downlink shared control channel detector and master information block (MIB) generation and recovery
- LTE first system information block (SIB1) decoder
- Hardware-software interface models for MIB and SIB1 bit parsing and channel estimation data indexing
- LTE waveform generation for multiple-antenna transmission
- Support for FDD and TDD for LTE transmitter and receiver applications

These reference applications can be used as-is to deliver packet information to your unique application and to generate synthesizable VHDL or Verilog with HDL Coder. They also serve as examples to illustrate recommended practices for implementing communications algorithms on FPGA or ASIC hardware.

## Generate HDL Code and Prototype on FPGA

Wireless HDL Toolbox provides blocks that support HDL code generation. To generate HDL code from designs that use these blocks, you must have an HDL Coder license. HDL Coder produces deviceindependent code with signal names that correspond to the Simulink model. HDL Coder also provides a tool to drive the FPGA synthesis and targeting process, and enables you to generate scripts and test benches for use with third-party HDL simulators.

To assist with the setup and targeting of programmable logic on a prototype board, and to verify your wireless communications system design on hardware, download a hardware support package such as Communications Toolbox Support Package for Xilinx Zynq-Based Radio.

## See Also

## External Websites

- Wireless HDL Toolbox
- HDL Coder

Simulation Acceleration

## Simulation Acceleration Using GPUs

```
In this section...
"GPU-Based System objects" on page 20-2
"General Guidelines for Using GPUs" on page 20-2
"Transmit and decode using BPSK modulation and turbo coding" on page 20-3
"Process Multiple Data Frames Using a GPU" on page 20-4
"Process Multiple Data Frames Using NumFrames Property" on page 20-4
"gpuArray and Regular MATLAB Numerical Arrays" on page 20-5
"Pass gpuArray as an Input" on page 20-5
"System Block Support for GPU System Objects" on page 20-5
```


## GPU-Based System objects

GPU-based System objects look and behave much like the other System objects in the Communications Toolbox product. The important difference is that the algorithm is executed on a Graphics Processing Unit (GPU) rather than on a CPU. Using the GPU can accelerate your simulation.

System objects for the Communications Toolbox product are located in the comm package and are constructed as:

H = comm.<object name>
For example, a Viterbi Decoder System object is constructed as:
H = comm.ViterbiDecoder
In cases where a corresponding GPU-based implementation of a System object exists, they are located in the comm. gpu package and constructed as:

H = comm.gpu.<object name>
For example, a GPU-based Viterbi Decoder System object is constructed as:
H = comm.gpu.ViterbiDecoder
To see a list of available GPU-based implementations enter help comm at the MATLAB command line and click GPU Implementations.

## General Guidelines for Using GPUs

Graphics Processing Units (GPUs) excel at processing large quantities of data and performing computations with high compute intensity. Processing large quantities of data is one way to maximize the throughput of your GPU in a simulation. The amount of the data that the GPU processes at any one time depends on the size of the data passed to the input of a GPU System object. Therefore, one way to maximize this data size is by processing multiple frames of data.

You can use a single GPU System object to process multiple data frames simultaneously or in parallel. This differs from the way many of the standard, or non-GPU, System objects are implemented. For

GPU System objects, the number of frames the objects process in a single call to the object function is either implied by one of the object properties or explicitly stated using the NumFrames property on the objects.

## Transmit and decode using BPSK modulation and turbo coding

This example shows how to transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, it shows how to decode using an iterative turbo decoder and display errors.

Define a noise variable, establish a frame length of 256 , and use the random stream property so that the results are repeatable.

```
noiseVar = 4; frmLen = 256;
s = RandStream('mt19937ar', 'Seed', 11);
intrlvrIndices = randperm(s, frmLen);
```

Create a Turbo Encoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The InterleaverIndices property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
turboEnc = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
    [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

Create a BPSK Modulator System object.

```
bpsk = comm.BPSKModulator;
```

Create an AWGN Channel System object.

```
channel = comm.AWGNChannel('NoiseMethod', 'Variance', 'Variance', ...
    noiseVar);
```

Create a GPU-Based Turbo Decoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The InterleaverIndicies property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
turboDec = comm.gpu.TurboDecoder('TrellisStructure', poly2trellis(4, ...
    [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
    'NumIterations', 4);
```

Create an Error Rate System object.

```
errorRate = comm.ErrorRate;
```

Run the simulation.

```
for frmIdx = 1:8
    data = randi(s, [0 1], frmLen, 1);
    encodedData = turboEnc(data);
    modSignal = bpsk(encodedData);
    receivedSignal = channel(modSignal);
```

Convert the received signal to log-likelihood ratios for decoding.

```
receivedBits = turboDec(-2/(noiseVar/2))*real(receivedSignal));
```

Compare original the data to the received data and then calculate the error rate results.

```
errorStats = errorRate(data,receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\nTotal bits = %d\n', ...
errorStats(1), errorStats(2), errorStats(3))
```


## Process Multiple Data Frames Using a GPU

This example shows how to simultaneously process two data frames using an LDPC Decoder System object. The ParityCheckMatrix property determines the frame size. The number of frames that the object processes is determined by the frame size and the input data vector length.

```
numframes = 2;
ldpcEnc = comm.LDPCEncoder;
ldpcGPUDec = comm.gpu.LDPCDecoder;
ldpcDec = comm.LDPCDecoder;
msg = randi([0 1], 32400,2);
for ii=1:numframes,
    encout(:,ii) = ldpcEnc(msg(:,ii));
end
%single ended to bipolar (for LLRs)
encout = 1-2*encout;
%Decode on the CPU
for ii=1:numframes;
    cout(:,ii) = ldpcDec(encout(:,ii));
end
%Multiframe decode on the GPU
gout = ldpcGPUDec(encout(:));
%check equality
isequal(gout,cout(:))
```


## Process Multiple Data Frames Using NumFrames Property

This example shows how to process multiple data frames using the NumFrames property of the GPUbased Viterbi Decoder System object. For a Viterbi Decoder, the frame size of your system cannot be inferred from an object property. Therefore, the NumF rames property defines the number of frames present in the input data.

```
numframes = 10;
convEncoder = comm.ConvolutionalEncoder('TerminationMethod', 'Terminated');
vitDecoder = comm.ViterbiDecoder('TerminationMethod', 'Terminated');
%Create a GPU Viterbi Decoder, using NumFrames property.
vitGPUDecoder = comm.gpu.ViterbiDecoder('TerminationMethod', 'Terminated', ...
    'NumFrames', numframes );
msg = randi([0 1], 200, numframes);
for ii=1:numframes,
    convEncOut(:,ii) = 1-2*convEncoder(msg(:,ii));
end
```

```
%Decode on the CPU
for ii=1:numframes;
    cVitOut(:,ii) = vitDecoder(convEncOut(:,ii));
end
%Decode on the GPU
gVitOut = vitGPUDecoder(convEncOut(:));
isequal(gVit0ut,cVit0ut(:))
```


## gpuArray and Regular MATLAB Numerical Arrays

A GPU-based System object accepts typical MATLAB arrays or objects created using the gpuArray class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a gpuArray, the data remains on the GPU. The output signal is a gpuArray. When the object is given a gpuArray, calculations take place entirely on the GPU, and no data transfer occurs. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

Passing MATLAB arrays to a GPU System object requires transferring the initial data from a CPU to the GPU. Then, the GPU System object performs calculations and transfers the output data back to the CPU. This process introduces latency. When data in the form of a gpuArray is passed to a GPU System object, the object does not incur the latency from data transfer. Therefore, a GPU System object runs faster when you supply a gpuArray as the input.

In general, you should try to minimize the amount of data transfer between the CPU and the GPU in your simulation.

## Pass gpuArray as an Input

This example shows how to pass a gpuArray to the input of the PSK modulator, reducing latency.

```
pskGPUModulator = comm.gpu.PSKModulator;
x = randi([0 7], 1000, 1, 'single');
gx = gpuArray(x);
o = pskGPUModulator(x);
class(o)
release(pskGPUModulator); %allow input types to change
go = pskGPUModulator(gx);
class(go)
```


## System Block Support for GPU System Objects

- "GPU System Objects Supported in System Block" on page 20-6
- "System Block Limitations for GPU System Objects" on page 20-6


## GPU System Objects Supported in System Block

- comm.gpu.AWGNChannel
- comm.gpu.BlockDeinterleaver
- comm.gpu.BlockInterleaver
- comm.gpu.ConvolutionalDeinterleaver
- comm.gpu.ConvolutionalEncoder
- comm.gpu.ConvolutionalInterleaver
- comm.gpu.PSKDemodulator
- comm.gpu.PSKModulator
- comm.gpu.TurboDecoder
- comm.gpu.ViterbiDecoder


## System Block Limitations for GPU System Objects

The GPU System objects must be simulated using Interpreted Execution. You must select this option explicitly on the block mask; the default value is Code generation.


## See Also

## More About

- "GPU Capabilities and Performance" (Parallel Computing Toolbox)


## Wireless Waveform Generator App

## Using Wireless Waveform Generator App

The Wireless Waveform Generator app is an interactive tool for creating, impairing, visualizing, and exporting modulated waveforms.

Start the app. On the Apps tab in the MATLAB toolstrip, under Signal Processing and Communications, click the app icon. You can also start the app by entering wirelessWaveformGenerator at the MATLAB command prompt.


These are the typical workflows when using the Wireless Waveform Generator app.

- Generate a waveform.
- Select the desired waveform type from the options available in the Waveform Type section. Adjust the configuration parameters in the Waveform pane. For more information, see "Waveform Type" on page 21-3.
- Select Impairments in the Generation section to open the Impairments pane. Adjust the configuration parameters in the Impairments pane. For more information, see "Add Impairments" on page 21-3.
- To perform signal generation, click Generate. After generation, the waveform is displayed. You can adjust the waveform, filtering, and impairment configurations and then regenerate the waveform.
- The default visualization plots vary based on the waveform type selected. Additional "Visualization Options" on page 21-4 can be opened by selecting them from Visualize in the Generation section.
- After generating a waveform you can export it by selecting the desired options from Export in the Export section. For more information, see "Export Waveform" on page 21-4.
- You can save the current session, open a previously saved session, or open a new session by selecting the desired option in the File section. For more information, see "Waveform Generator Session" on page 21-5.


## Waveform Type

To generate the various available waveforms, the Wireless Waveform Generator app uses Communications Toolboxfeatures. The supported waveform types include:

- OFDM - The app uses the comm. OFDMModulator System object to generate this type of waveform.
- QAM - The app uses the qammod function to generate this type of waveform.
- PSK - The app uses the pskmod function to generate this type of waveform.
- Sinewave - The app uses the dsp. SineWave System object to generate this type of waveform.
- 5G - If you have the 5G Toolbox, you can also generate NR test models (NR-TM) and NR uplink and downlink fixed reference channel (FRC) waveforms.
- LTE - If you have the LTE Toolbox you can also generate LTE modulated waveforms using features in the "LTE Toolbox".
- WLAN - If you have the WLAN Toolbox ${ }^{\mathrm{TM}}$ you can also generate $802.11^{\mathrm{TM}}$ modulated waveforms using features in the "WLAN Toolbox".
- Bluetooth - You can download the Communications Toolbox Library for the Bluetooth Protocol Add-On and generate Bluetooth modulated waveforms using features described in "Bluetooth".

By default, generated waveforms have no filtering applied. To apply filtering to the waveform, select the desired filter option from the Filtering parameter on the Waveform pane. The available filter options vary based on the waveform type you select.

## Add Impairments

You can add these impairments to the waveform that you generate.

- AWGN - The app uses the awgn function to impair the waveform.
- Phase offset - The app impairs the waveform by applying the specified phase offset as $y=x e^{j \varphi}$, where $\varphi$ is the phase offset in radians.
- Frequency offset - The app uses the comm. PhaseFrequencyOffset System object to impair the waveform.
- Phase noise - The app uses the comm. PhaseNoise System object to impair the waveform.
- DC offset - The app impairs the waveform by applying the specified DC offset as $y=x+d c O f f$, where dcOff is the complex DC offset in Volts.
- IQ imbalance - The app uses the iqimbal function to impair the waveform..
- Memoryless cubic nonlinearity - The app uses the comm.MemorylessNonlinearity System object to impair the waveform.


## Visualization Options

You can use these plot types to visualize waveforms that you generate.

- Spectrum Analyzer - The app plots the waveform in the frequency domain.
- OFDM Grid - For OFDM waveforms, the app plots the resource allocation of data and control channels.
- Time scope - The app plots the inphase and quadrature (IQ) waveform samples in the time domain.
- Constellation diagram - The app plots the constellation points of the modulation symbols.


## Export Waveform

You can export the waveform to a runnable MATLAB script, to your workspace, or to a signal file.

- Waveforms exported to a runnable MATLAB script can be used to reproduce your waveform outside of the Wireless Waveform Generator app by running the script.
- Waveforms exported to the workspace are saved as a structure containing these fields:
- type - This field is a character vector indicating the waveform type.
- config - This field is a structure or object containing fields that specify the configured waveform type.
- Fs - This field is the signal sample rate in Hertz.
- waveform - The field is the complex waveform samples output as an $N_{\mathrm{S}}$-by-1 column vector or an $N_{\mathrm{S}}$-by- $N_{\mathrm{T}}$ matrix. $N_{\mathrm{S}}$ is the number of time-domain samples, and $N_{\mathrm{T}}$ is the number of transmit antennas.
- Waveforms exported to a signal file can be saved as a .mat or.$b b$ file.
- MAT-files are binary MATLAB files that store workspace variables. For more information, see "MAT-File Versions" (MATLAB).
- The app uses the comm. BasebandFileWriter System object to save .bb files.


## Transmit Using Lab Test Instrument

Generate a waveform that you can transmit using a connected lab test instrument. The Wireless Waveform Generator app can transmit using instruments supported by the rfsiggen function.


Use of the transmit feature in the Wireless Waveform Generator app requires "Instrument Control Toolbox".

## Waveform Generator Session

You can save the current session, open a previously saved session, or open a new session by selecting the desired option in the FILE section. When you save a waveform generator session, the session configuration is saved as a .mat file. For more information, see "MAT-File Versions" (MATLAB).

## See Also

Wireless Waveform Generator

## RF Propagation

- "Site Viewer" on page 22-2
- "Troubleshooting Site Viewer" on page 22-8
- "Access Basemaps and Terrain in Site Viewer" on page 22-9
- "Access TIREM Software" on page 22-11


## Site Viewer

```
In this section...
"Introduction" on page 22-2
"Dimension Picker" on page 22-2
"Basemaps" on page 22-3
"View Latitude and Longitude Coordinates of Location" on page 22-6
"Limitations for Terrain and Buildings" on page 22-7
```


## Introduction

Transmitter and receiver site for RF propagation visualization involves a map display called Site Viewer. You can use the Home button to view the default home location of Site Viewer.


Site Viewer includes in-canvas pan, zoom, and rotation.
Site Viewer is a 3-D map display and requires hardware graphics support for WebGL ${ }^{\mathrm{TM}}$.
Opening Site Viewer using siteviewer allows customization of basemap, terrain, and buildings.

## Dimension Picker

The dimension picker offers three choices: 3-D, 2-D, and Columbus View.


- 3-D View - Provides a smooth globe. This is the default view.
- 2-D View - Provides a flat map in Mercator projection.
- Columbus View - Provides a flat map in Mercator projection that allows tilt and rotation.


## Basemaps

Basemaps are the underlying maps used by Site Viewer.


You can select the basemaps using the basemap picker. You need an active internet connection for all basemaps except Dark Water. The following options are available:

- Satellite
- Topographic
- Streets
- Streets-Light
- Streets-Dark
- OpenStreetMap
- Dark Water
- Gray Land
- Blue Green
- Gray Terrain
- Color Terrain
- Land Cover

By default, Site Viewer uses Satellites. You need an active internet connection for all basemaps except Dark Water.

Use Satellites for high-quality satellite imagery. Use Streets, or OpenStreetMap for imagery of roads, labels, and political boundaries.

Use The MathWorks ${ }^{\circledR}$ natural earth basemaps for low resolution but more controls and color scheme options in basemaps.

Basemaps


## View Latitude and Longitude Coordinates of Location

Use Site Viewer to view the latitude and longitude coordinates of any location. To view the latitude and longitude, right-click the map and select Show Location.


To remove the location, right-click the location and select Remove Location.


## Limitations for Terrain and Buildings

## Terrain

- Default terrain access requires internet connection. If no internet connection exists, then Site Viewer automatically uses 'none' in the property Terrain.
- Custom DTED terrain files for use with addCustomTerrain must be acquired outside of MATLAB for example by using USGS EarthExplorer.
- When using custom terrain, analysis is restricted to the terrain region. For example, an error occurs if trying to show a txsite or rxsite outside of the region.


## Buildings

- OpenStreetMap files obtained from https://www.openstreetmap.org represent crowd-sourced map data, and the completeness and accuracy of the buildings data may vary depending on the map location.
- When downloading data from https://www.openstreetmap.org, select an export area larger than the desired area to ensure that all expected building features are fully captured. Building features at the edge of the selected export area may be missing.
- Building geometry and features are interpreted from the file according to the recommendations of OpenStreetMap for 3-D buildings.


## See Also

opengl

## More About

- "Troubleshooting Site Viewer" on page 22-8
- "Access Basemaps and Terrain in Site Viewer" on page 22-9
- "System Requirements for Graphics" (MATLAB)
- "Resolving Low-Level Graphics Issues" (MATLAB)


## Troubleshooting Site Viewer

```
In this section...
"Internet Connection Failure" on page 22-8
"Graphics Environment" on page 22-8
```


## Internet Connection Failure

When you start Site Viewer up on showing a txsite or rxsite object, a check is made to make sure that you have an internet connection to retrieve the default basemap and terrain data.

If Site Viewer cannot connect to the Internet the following warning messages are displayed:

- Warning: Unable to access the Internet, showing Dark Water instead of Satellites. See Access Basemaps and Terrain in Site Viewer.
- Warning: Unable to access terrain data. See Access Basemaps and Terrain in Site Viewer.

If Site Viewer cannot connect to the Internet, then terrain data is not used and the Dark Water basemap is selected.

## Graphics Environment

Site Viewer can fail to open because of two reasons:

- In MATLAB, OpenGL ${ }^{\circledR}$ is set to software graphics. An error message is displayed in the command window, notifying you to upgrade the graphics hardware driver or select hardware graphics using OpenGL.

For more information, see opengl, and "Resolving Low-Level Graphics Issues" (MATLAB).

- When starting Site Viewer, JavaScript ${ }^{\circledR}$ for WebGL support fails. An error message is displayed in the command window, notifying you to update the graphics hardware driver.

For more information, see "Resolving Low-Level Graphics Issues" (MATLAB)

## See Also

opengl

## More About

- "Site Viewer" on page 22-2
- "Access Basemaps and Terrain in Site Viewer" on page 22-9
- "System Requirements for Graphics" (MATLAB)
- "Resolving Low-Level Graphics Issues" (MATLAB)


## Access Basemaps and Terrain in Site Viewer

## In this section...

"Access and Download Basemaps" on page 22-9
"Access Terrain" on page 22-9

## Access and Download Basemaps

To download MATLAB basemaps:
1 On the MATLAB Home tab, in the Environment section, click Add-Ons > Get Add-Ons.
2 In the Add-On Explorer, scroll to the MathWorks Features section, and click show all to find the basemap packages. You can also search for the basemap add-ons by name (listed in the following table) or click Features in Filter by Type.
3 Select the basemap data packages. For more information about basemaps, see geobubble.

| Basemap Name | Basemap Data Package Name |
| :--- | :--- |
| 'bluegreen' | MATLAB Basemap Data - bluegreen |
| 'grayland' | MATLAB Basemap Data - grayland |
| ' colorterrain' | MATLAB Basemap Data - colorterrain |
| 'grayterrain' | MATLAB Basemap Data - grayterrain |
| 'landcover' | MATLAB Basemap Data - landcover |

In addition, Site Viewer also supports three external basemaps that you can select from the basemap picker. The following options are available:

- Satellite
- Topographic
- Streets
- Streets-Light
- Streets-Dark
- OpenStreetMap

You need an active internet connection for all Site viewer basemaps except Dark Water. Use addCustomBasemap to access basemaps from user-specified URLs.

Note If basemap does not render correctly in Site Viewer (for example only the ocean is visible), check if the basemap server supports CORS (cross-origin resource sharing). Site Viewer does not support basemaps that do not support CORS.

## Access Terrain

To access terrain data for Site Viewer, you need an active internet connection. Use addCustomTerrain to access terrain data from DTED-files without an active internet connection.

## See Also

## More About

- "Site Viewer" on page 22-2
- "Troubleshooting Site Viewer" on page 22-8
- "System Requirements for Graphics" (MATLAB)
- "Resolving Low-Level Graphics Issues" (MATLAB)


## Access TIREM Software

The Terrain Integrated Rough Earth Model ${ }^{\mathrm{TM}}$ (TIREM ${ }^{\mathrm{TM}}$ ) is a propagation model for computing the path loss for irregular terrain and seawater scenarios. TIREM is developed, trademarked, and licensed by Alion Science. To use TIREM, you need to acquire it from Alion Science.

TIREM is designed to calculate the reference basic median propagation loss (path loss) based on the terrain profile along the great circle path between two antennas, for example, using digital terrain elevation data (DTED). You can use TIREM model to calculate the point-to-point path loss between sites over irregular terrain. The model combines physics with empirical data to provide path loss estimates. The TIREM propagation model can predict path loss at frequencies between 1 MHz and 1 THz.

Use tiremSetup to enable TIREM access from within MATLAB. The TIREM library folder contains the tirem3 shared library. The full library name is platform-dependent:

| Platform | Shared Library Name |
| :--- | :--- |
| Windows | libtirem3.dll or tirem3.dll |
| Linux | libtirem3.so |
| Mac | libtirem3.dylib |

## See Also

tiremSetup


[^0]:    scatterplot(y)

[^1]:    1000 symbols. Set
    'Computation Delay" to 1000.

