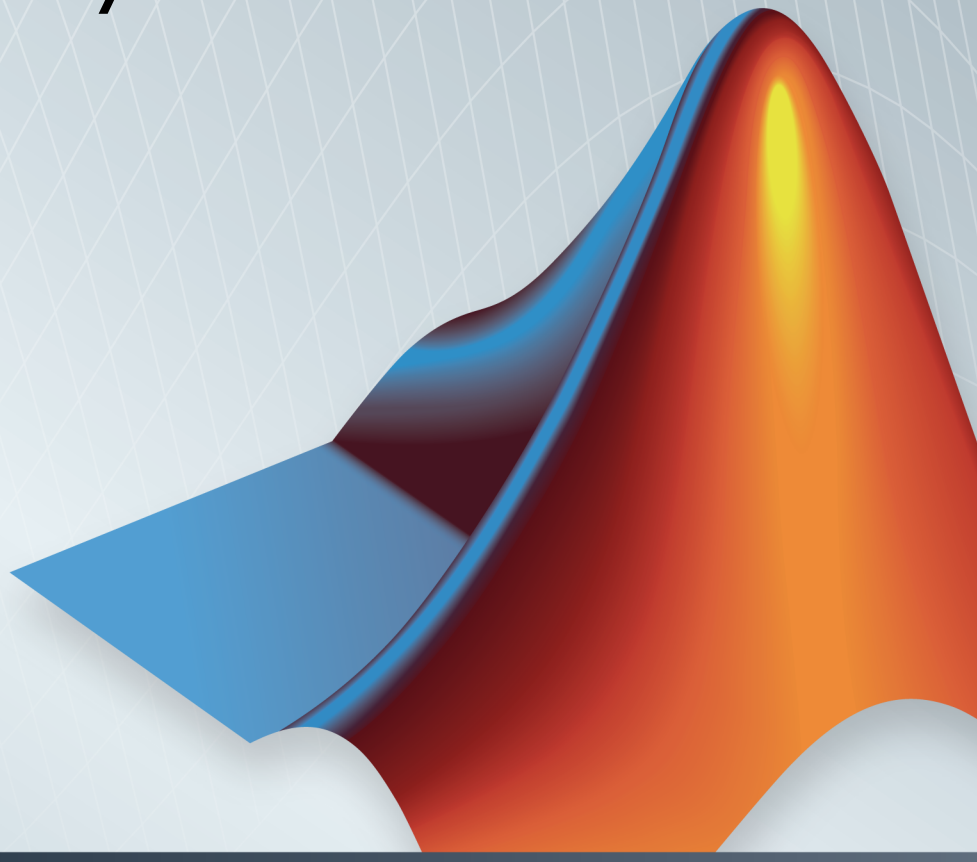
Communications System Toolbox™

User's Guide

R2014**b**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Communications System Toolbox™ User's Guide

Revision History

April 2011	First printing	New for Version 5.0 (Release 2011a)
September 2011	Online only	Revised for Version 5.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.3 (Release 2012b)
March 2013	Online only	Revised for Version 5.4 (Release 2013a)
September 2013	Online only	Revised for Version 5.5 (Release 2013b)
March 2014	Online only	Revised for Version 5.6 (Release 2014a)
October 2014	Online only	Revised for Version 5.7 (Release 2014b)

Contents

1 Input, Output, and Display

2 Data and Signal Management

3 Adaptive Equalizer Examples

4 System Design

5 OFDM Modulation

6 MSK

7 Reed-Solomon Coding

8 Galois Fields

9 Convolutional Coding

10 Channel Modeling and RF Impairments

11 Measurements

12 Filtering Section

13 Visual Analysis

14 C Code Generation

15 HDL Code Generation

16 Simulation Acceleration

17 Define New System Objects

18 Orphan Pages

Input, Output, and Display

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

Signal Terminology

This section defines important Communications System 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 one-dimensional 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

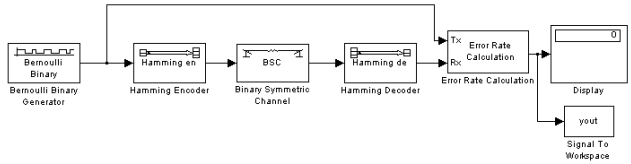
In this section...

Use a Signal To Workspace Block

This section explains how to send data from a Simulink® model to the MATLAB® workspace so you can analyze the results of simulations in greater detail.

You can use a Signal To Workspace block, from the Sinks library of the DSP System Toolbox™ product 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 “Reduce the Error Rate Using a Hamming Code” on page 4-80. To insert a Signal to Workspace block into the model, follow these steps:

1. Type doc_hamming at the MATLAB Help browser to open the model.
2. Drag a Signal To Workspace block, from the Sinks library in the DSP System Toolbox product, into the model window and connect it as shown in the following figure.



Hamming Code Model with a Signal To Workspace Block

Configure the Signal To Workspace Block

To configure the Signal 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 Signal to Workspace block by typing the following commands at the MATLAB prompt:

```
format short e
hammcode_BER
```

The vector output is the following:

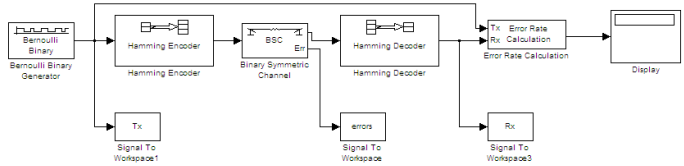
```
hammcode_BER =
5.4066e-003  1.0000e+002  1.8496e+004
```

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.



Send Signal and Error Data to the Workspace

1. To open the model shown in the previous figure, type `doc_channel` at the MATLAB command line.
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. Drag three Signal To Workspace blocks, from the Sinks library in the DSP System Toolbox product, into the model window and connect them as shown in the preceding figure.
4. Double-click the left Signal 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 **Frames** field, select `Log frames separately (3-D array)`. This preserves each frame as a separate column of the array `Tx`.
   - Click **OK**.
5. Double-click the middle Signal To Workspace block:
   - Type `errors` in the **Variable name** field.
   - In the **Frames** field, select `Log frames separately (3-D array)`.
   - Click **OK**.
6. Double-click the right Signal To Workspace block:
   - Type `Rx` in the **Variable name** field.
   - In the **Frames** field, select `Log frames separately (3-D array)`.
   - 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

```
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

Communications System Toolbox provides sinks and display devices that facilitate analysis of communication system performance. You can implement devices using either System objects, blocks, or functions.

In this section...

Data sources

You can use blocks or functions to generate random data to simulate a signal source. In addition, you can use Simulink blocks such as the Random Number block as a data source. You can open the Random Data Sources sublibrary by double-clicking its icon (found in the Comm Sources library of the main Communications System Toolbox block library).

Random Symbols

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

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

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

a =

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

```
     3     1     1     3
```

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

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

b =

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

Random Integers

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 = randi([2,10],5,4)

c =

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

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

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

In Simulink, 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 =

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

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

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

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

g =

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

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

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

```
binarymatrix1 = randsrc(100,1,[0 1]); % Possible values are 0,1.
binarymatrix2 = randi([0 1],100,1); % Two possible values
binarymatrix3 = randerr(100,1,[0 1;.5 .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

Blocks in the Noise Generators sublibrary of the Comm Sources library generate random data to simulate channel noise. You can use blocks in the Noise Generators sublibrary to generate random real numbers, depending on what distribution you want to use. The choices are listed in the following table.

Random Noise Generators

Blocks in the Noise Generators sublibrary of the Comm Sources library generate random data to simulate channel noise. You can use blocks in the Noise Generators sublibrary to generate random real numbers, depending on what distribution you want to use. The choices are listed in the following table.

Distribution	Block
Gaussian	Gaussian Noise Generator
Rayleigh	Rayleigh Noise Generator
Rician	Rician Noise Generator
Uniform on a bounded interval	Uniform Noise Generator

You can open the Noise Generators sublibrary by double-clicking its icon in the main Communications System Toolbox block library.

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. The function assumes that the load impedance is 1 ohm.

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

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

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

To send a signal through an additive white Gaussian noise channel, use the `awgn` function. See "AWGN Channel" for more information.

In Simulink, you use the Gaussian Noise Generator block to add Gaussian noise to a communications model.

Sequence Generators

You can use blocks in the Sequence Generators sublibrary of the Communications Sources library to generate sequences for spreading or synchronization in a communication system. You can open the Sequence Generators sublibrary by double-clicking its icon in the main Communications System Toolbox block library.

Blocks in the Sequence Generators sublibrary generate

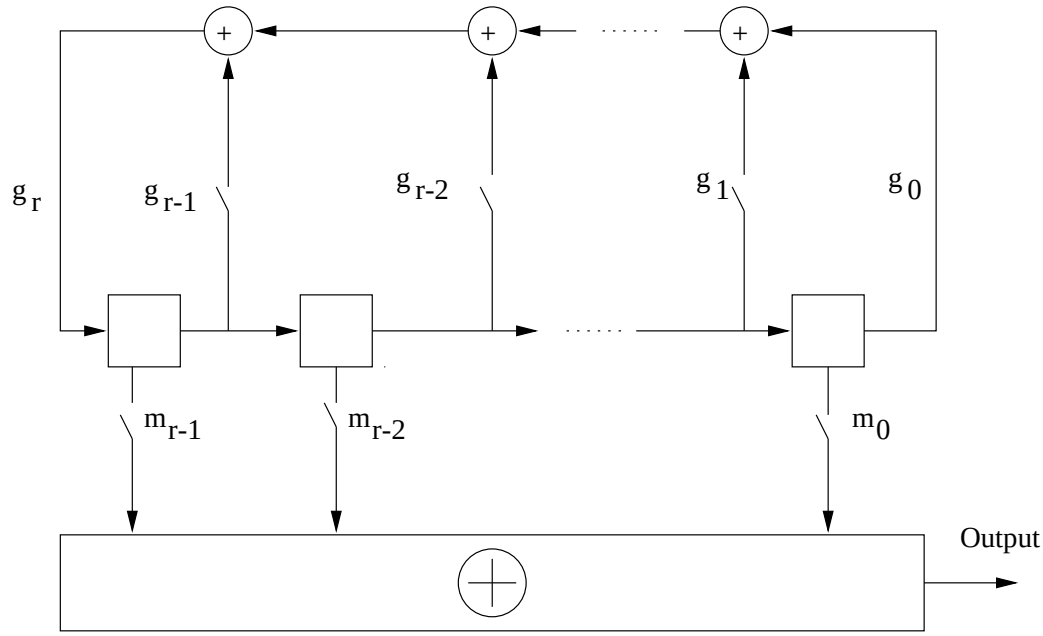
- Pseudorandom sequences
- Synchronization codes
- Orthogonal codes

Pseudorandom Sequences

The following table lists the blocks that generate pseudorandom or pseudonoise (PN) sequences. The applications of these sequences range from multiple-access spread spectrum communication systems to ranging, synchronization, and data scrambling.

Sequence	Block
Gold sequences	Gold Sequence Generator
Kasami sequences	Kasami Sequence Generator
PN sequences	PN Sequence Generator

All three blocks use shift registers to generate pseudorandom sequences. The following is a schematic diagram of a typical shift register.



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, $g_r z^r + g_{r-1} z^{r-1} + ... + g_0$. The coefficient g_i is 1 if there is a connection from the ith shift register to the adder, and 0 otherwise.

The Kasami Sequence Generator block and the PN Sequence Generator block use this polynomial description for their **Generator polynomial** parameter, while the Gold Sequence Generator block uses it for the **Preferred polynomial [1]** and **Preferred polynomial [2]** parameters.

The lower half of the preceding diagram shows how the output sequence can be shifted by a positive integer d, by delaying the output for d units of time. This is accomplished by a single connection along the dth arrow in the lower half of the diagram.

Synchronization Codes

The Barker Code Generator block generates 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	Block
Hadamard codes	Hadamard Code Generator
OVSF codes	OVSF Code Generator
Walsh codes	Walsh Code Generator

Scopes

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

- “Eye Diagrams” on page 1-13
- “Scatter Plots” on page 1-14
- “Signal Trajectories” on page 1-14

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

Block Name	Plots
Discrete-Time Eye Diagram Scope	Eye diagram of a discrete signal
Constellation Diagram	Constellation diagram of a signal
Discrete-Time Signal Trajectory Scope	Signal trajectory of a discrete 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 Discrete-Time Eye Diagram Scope 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 1-17.

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.

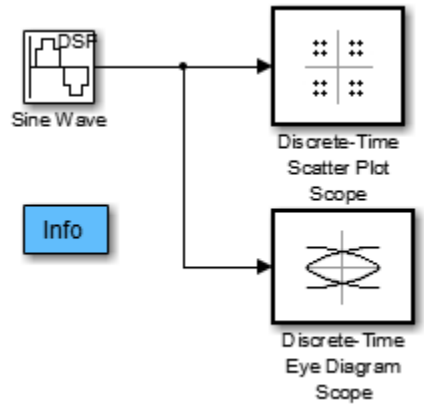
Signal Trajectories

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 Discrete-Time Signal Trajectory Scope block produces signal trajectories. Unlike the Constellation Diagram block, which displays points on the trajectory at discrete time intervals corresponding to the decision points, the Discrete-Time Signal Trajectory Scope displays a continuous picture of the signal's trajectory between decision points.

View a Sinusoid

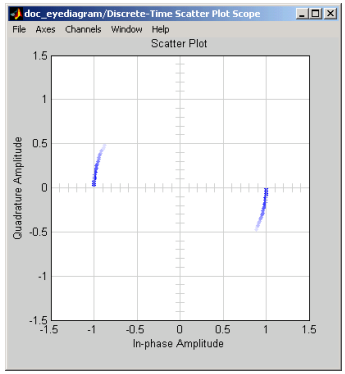
The following model produces a constellation diagram and an eye diagram from a complex sinusoidal signal. Because the decision time interval is almost, but not exactly, an integer multiple of the period of the sinusoid, the eye diagram exhibits drift over time. More specifically, successive traces in 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`.
- Discrete-Time Eye Diagram Scope, 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 55 35 35]);`.

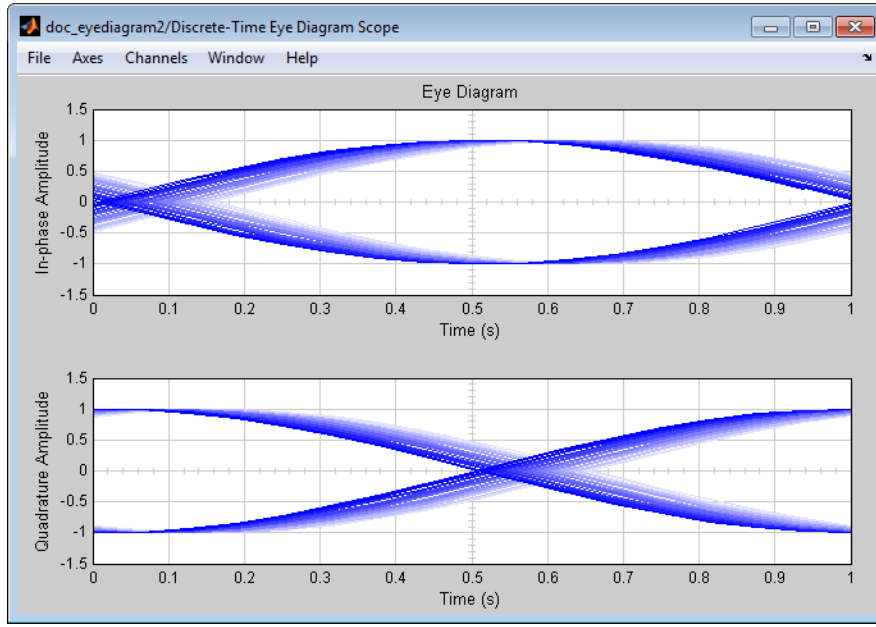
Connect the blocks as shown in the preceding figure. From the model window's **Simulation** menu, choose **Model Configuration parameters**. In the **Configuration Parameters** dialog box, set **Stop time** to `250`. Running the model produces the following scatter diagram plot.



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 Discrete-Time Signal Trajectory Scope 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.



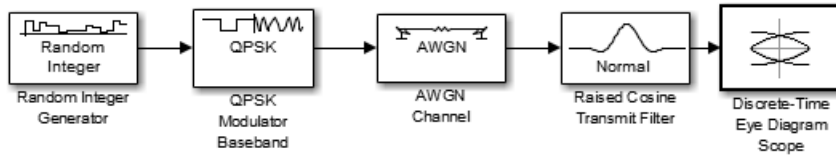
View a Modulated Signal

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

- "Eye Diagram of a Modulated Signal" on page 1-17
- "Constellation Diagram of a Modulated Signal" on page 1-20
- "Signal Trajectory of a Modulated Signal" on page 1-21

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.

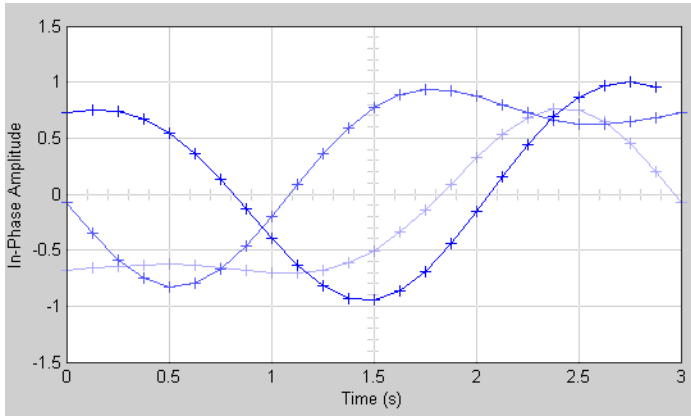


To open the model, enter `doc_signaldisplays` at the MATLAB command line. To build the model, gather and configure the following blocks:

- 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 System Toolbox, with default parameters
- AWGN Channel, in the Channels library of Communications System 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)`.
- Discrete-Time Eye Diagram Scope, 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.

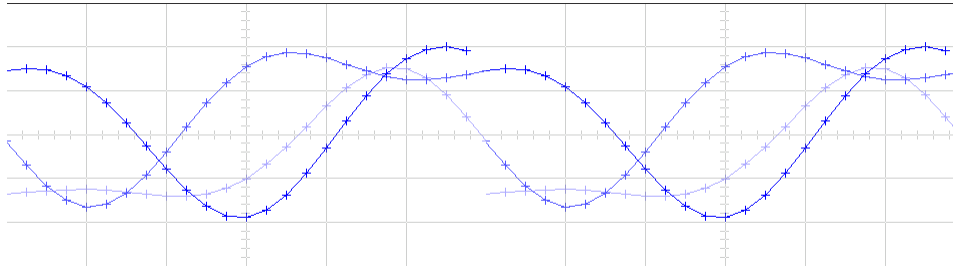
- On the **Figure Properties** panel, set **Eye diagram to display** to `In-phase only`.

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

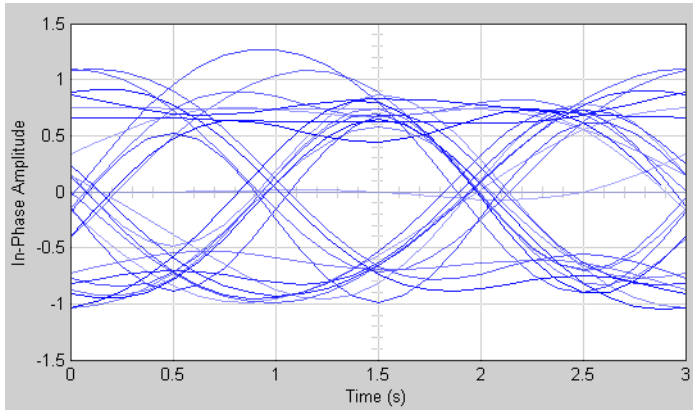


Three traces are displayed. Traces 2 and 3 are faded because **Color fading** under **Rendering Properties** is selected. This causes traces to be displayed less brightly the older they are. In this picture, Trace 1 is the most recent and Trace 3 is the oldest. Because **New traces per display** is set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



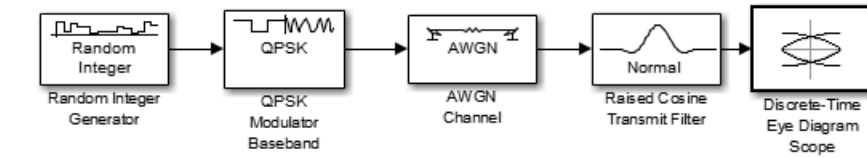
You can view a more realistic eye diagram by changing the value of **Traces displayed** to 40 and clearing the **Markers** field.



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

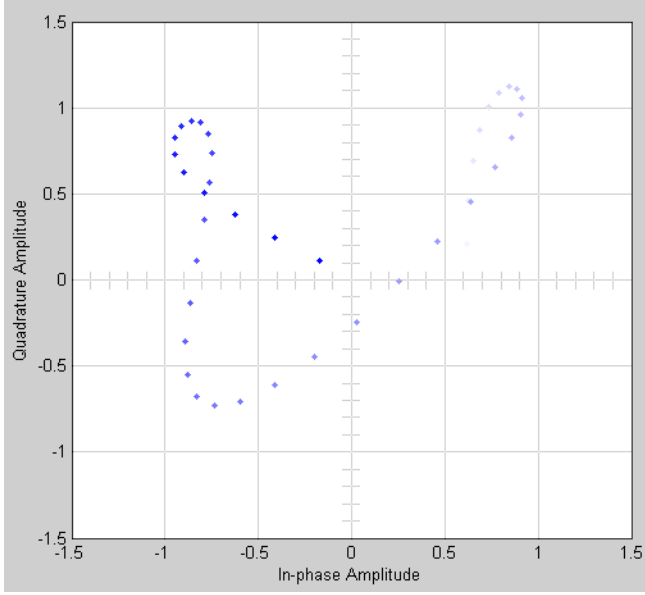
The following model creates a scatter plot of the same signal considered in "Eye Diagram of a Modulated Signal" on page 1-17.



To build the model, follow the instructions in "Eye Diagram of a Modulated Signal" on page 1-17 but replace the Discrete-Time Eye Diagram block with the following block:

- Constellation Diagram, in the Comms 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.

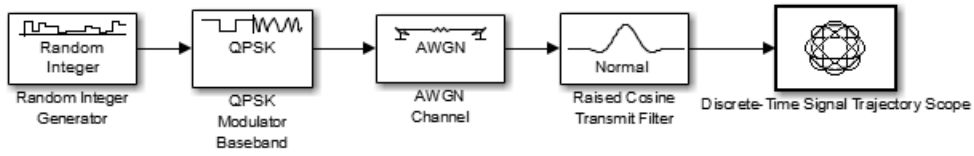
When you run the simulation, the Constellation Diagram block displays the following plot.



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

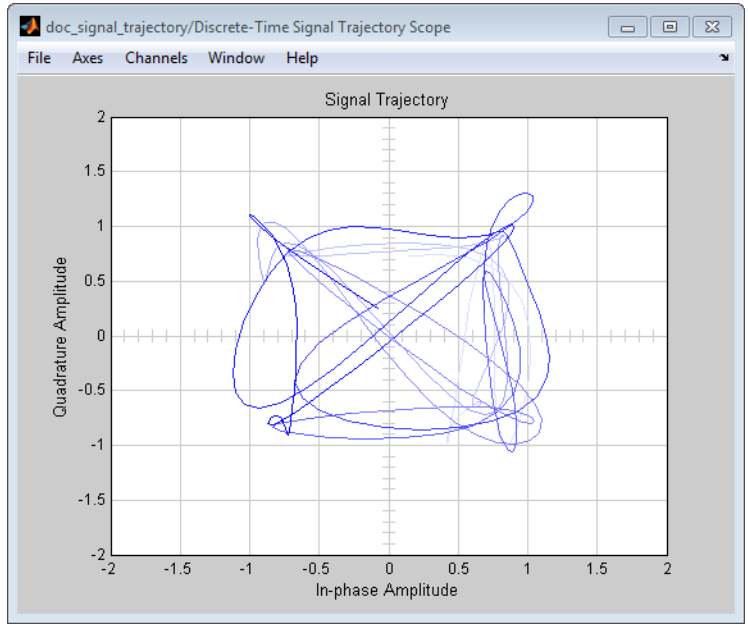
The following model creates a signal trajectory plot of the same signal considered in "Eye Diagram of a Modulated Signal" on page 1-17.



To build the model, follow the instructions in "Eye Diagram of a Modulated Signal" on page 1-17 but replace the Discrete-Time Eye Diagram block with the following block:

- Discrete-Time Signal Trajectory Scope, in the Comms Sinks library
  - Set **Samples per symbol** to `8`.
  - Set **Symbols displayed** 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 displayed**.
  - Set **New symbols per display** to `10`. This specifies the number of new symbols that appear each time the diagram is refreshed. The number of symbols that remain in the diagram from one refresh to the next is **Symbols displayed** minus **New symbols per display**.

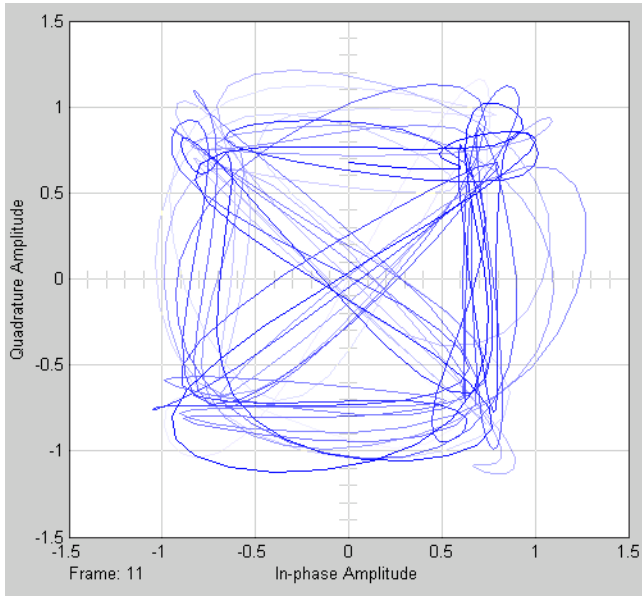
When you run the model, the Discrete-Time Signal Trajectory Scope displays a trajectory like the one below.



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 1-20 to compare the preceding signal trajectory to the scatter plot of the same signal. The Discrete-Time Signal Trajectory Scope block connects the points displayed by the Constellation Diagram block to display the signal trajectory.

If you increase **Symbols displayed** 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 displayed**.


1.5
1
0.5
0
-0.5
-1
-1.5
Quadrature Amplitude
-1.5
-1
-0.5
0
0.5
1
1.5
Frame: 11
In-phase Amplitude

Support SDR Hardware

Communications System Toolbox software can read a signal from external hardware devices using the Communications System 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 bitstream and also run your own design in the FPGA with the automated targeting workflow using HDL Coder™.

The support packages for SDR support both fixed bitstream and custom bitstream (user-provided logic) workflows (SDR Targeting).

For more about software-defined radio with MATLAB and Simulink, visit Software-Defined Radio (SDR) on the MathWorks® web site.

For a list of support packages for use with Communications System Toolbox, visit the Hardware Support Catalog for Communications System Toolbox.

Transmit and Receive Signals Over the Air with Software Defined Radios

Communications System Toolbox software can read a signal from external hardware devices using the Communications System 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 bitstream and also run your own design in the FPGA with the automated targeting workflow using HDL Coder.

The support packages for SDR support both fixed bitstream and custom bitstream (user-provided logic) workflows (SDR Targeting).

For more about software-defined radio with MATLAB and Simulink, visit Software-Defined Radio (SDR) on the MathWorks web site.

For a list of support packages for use with Communications System Toolbox, visit the Hardware Support Catalog for Communications System Toolbox.

Data and Signal Management

Matrices, Vectors, and Scalars

Simulink supports matrix signals, one-dimensional arrays, sample-based processing, and frame-based processing. This section describes how Communications System 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 one-dimensional 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 System Toolbox process scalar, vector, and matrix signals.

- In their numerical computations, blocks that process scalars do not distinguish between one-dimensional 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 Frame-Based Concepts" in the DSP System Toolbox documentation.

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?" in the DSP System Toolbox documentation.

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” in the Fixed-Point Designer™ documentation and “Fixed-Point Signal Processing” in the DSP System Toolbox documentation.

Access the Block Support Table

The Communications System Toolbox Block Support Table is available through the Simulink model Help menu. The table provides information about data type support and code generation coverage for all Communications System Toolbox blocks. To access the table, select **Help > Simulink > Block Data Types & Code Generation Support > Communications System Toolbox**.

You can also access the Communications System Toolbox Data Type Support Table by typing `showcommblockdatatypetable` at the MATLAB command line.

Delays

In this section...

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 Align Signals 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"
- Find Delay block reference page
- Align Signals 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.".

- “Example: Soft-Decision Decoding”. (See “Delay in Received Data”.)
- “Example: Delays from Demodulation”.

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 asymmetric digital subscriber line (ADSL) example model and aims 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.

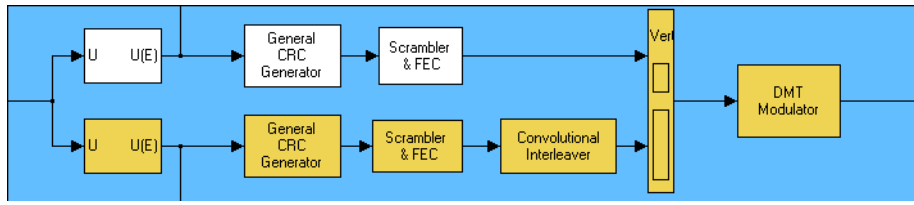
In the ADSL example, data follows one of two parallel paths, one with a nonzero delay and the other with a delay of zero. One path includes a convolutional interleaver and deinterleaver, while the other does not. Near the end of each path is an Error Rate Calculation block, whose **Receive delay** parameter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

Frame Periods in the Model

Before searching for individual delays, first observe that most signal lines throughout the model share the same frame period. To see this, select **Display > Sample Time**. 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 relative to the Error Rate Calculation block is one frame, or 776 samples.

Block	Delay, in Output Samples from Individual Block	Delay, in Frames	Delay, in Input Samples to Error Rate Calculation Block
Convolutional Interleaver and Convolutional 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

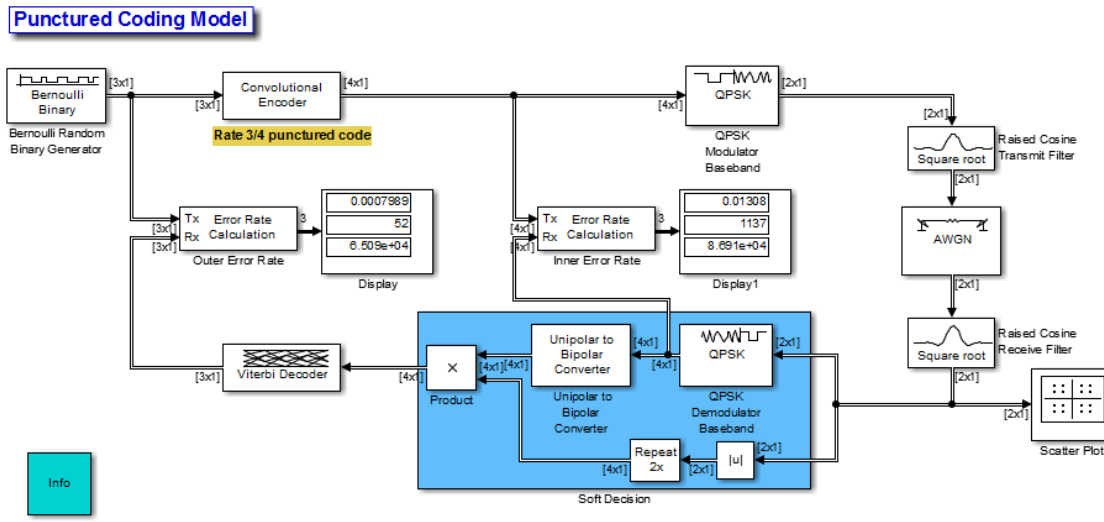
No other blocks in the interleaved path of the example cause any delays. Adding the delays from the interleaver/deinterleaver pair and the Delay block indicates that the total delay in the interleaved path is one frame.

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 outports 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.



Frame Periods in the Model

Before searching for individual delays, select **Display>Sample Time>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 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 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 2-10.

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.

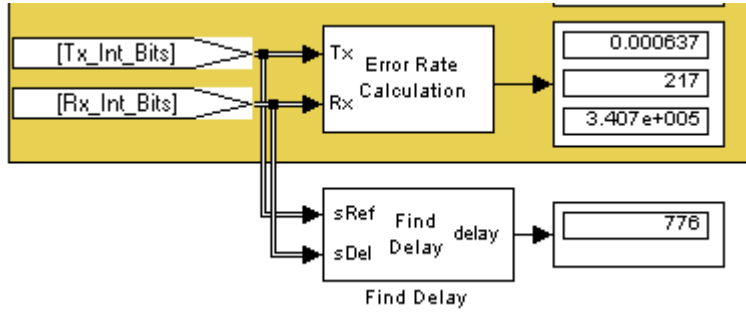
Use the Find Delay and Align Signals Blocks

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, or you could have avoided needing to know those numbers by using the Align Signals block. This section explains both techniques using the ADSL example model, `commadsl`, as an example. Applying the techniques to the punctured convolutional coding example, discussed in "Punctured Coding Model" on page 2-9, would be similar.

Using the Find Delay Block to Determine the Correct Receive Delay

Recall from "Path for Interleaved Data" on page 2-8 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.

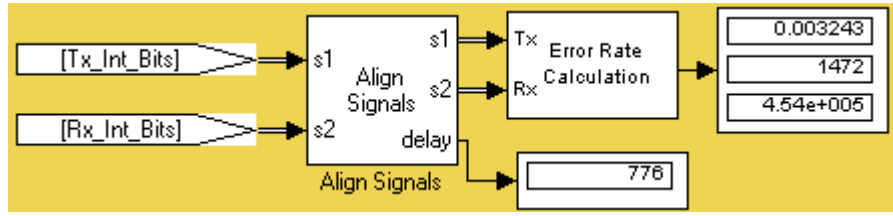
The new Display block now shows the value 776, as expected.

Using the Align Signals Block Before Computing the Error Rate

To use the Error Rate Calculation block to compute the value labeled `Interleaved BER` *without* having to set the **Receive delay** parameter to a nonzero value, you can use the Align Signals block to automatically align the transmitted and received signals before the Error Rate Calculation block performs its computations. Use this procedure:

1 Insert an Align Signals 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 Align Signals 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 Align Signals block cannot find the correct amount by which to delay one of the signals. If the `delay` output from the Align Signals block does not stabilize at a constant value, the correlation window length is probably too small.

4 Set the Error Rate Calculation block's **Receive delay** parameter to `0`. You might also want to set the block's **Computation delay** parameter to a nonzero value to account for the possibility that the Align Signals block takes a nonzero amount of time to stabilize on the correct amount by which to delay one of the signals.

5 Run the simulation.

The new Display block now shows the value 776. Also, the Align Signals block delays one signal relative to the other so that the signals are aligned. The Error Rate Calculation block therefore processes two signals that are properly aligned with each other and does not need to use a nonzero **Receive delay** parameter to attempt any further alignment.

Examining the delay output signal from the Align Signals block, using the Display block as in the figure above, is important because if the delay output signal does not stabilize at a constant value, the signals are not truly aligned and the error rate is not reliable. In this case, the Align Signals block's **Correlation window length** parameter is probably too small.

Manipulate Delays

- "Delays and Alignment Problems" on page 2-14
- "Aligning Words of a Block Code" on page 2-18
- "Aligning Words for Interleaving" on page 2-20
- "Aligning Words of a Concatenated Code" on page 2-22
- "Aligning Words for Nonlinear Digital Demodulation" on page 2-25

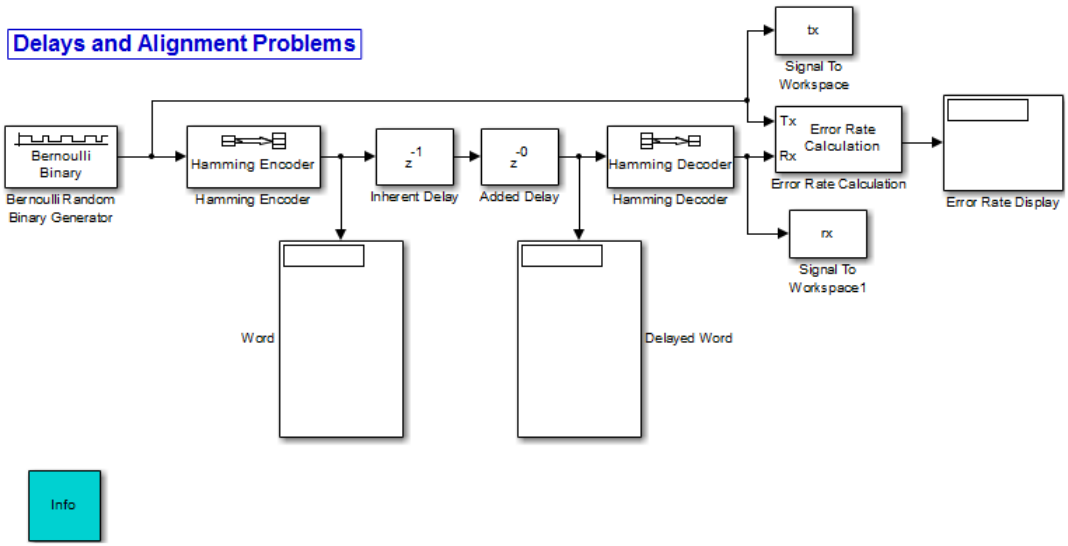
Delays and Alignment Problems

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 2-7).

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 2-7 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 (`rx`) 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.

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 7k-1 and 4k, respectively,

for any positive integer k. It is important that the sum of the inherent delay (1) and the added delay (7k-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.

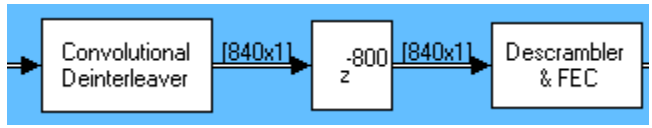
Misalignment of Codewords

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 2-8, 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 2-16, 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 840-sample 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.



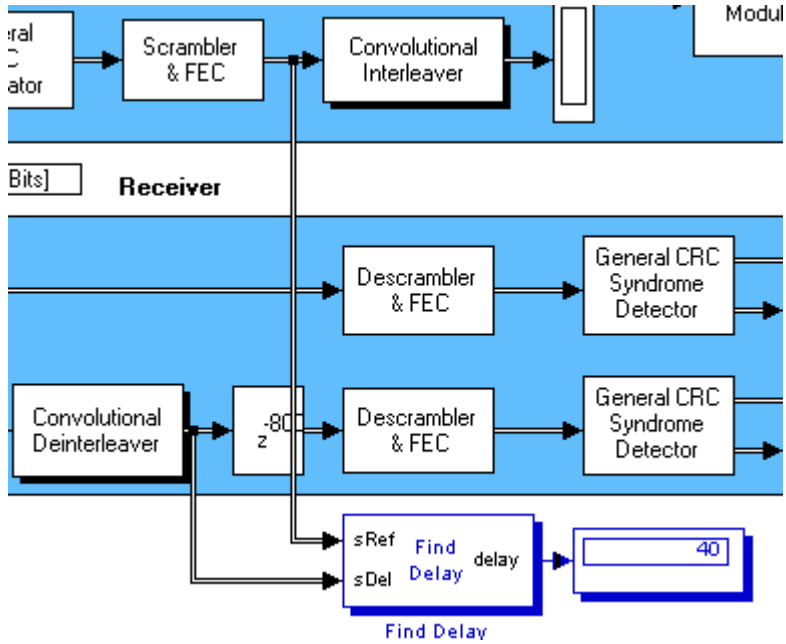
The **Delay** parameter in the Delay block is `800` because that is the minimum number of samples required to shift the 41st sample of one 840-sample frame to the first sample of the next 840-sample frame. In other words, the sum of the inherent 40-sample delay (from the interleaving/deinterleaving process) 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 2-8.

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).



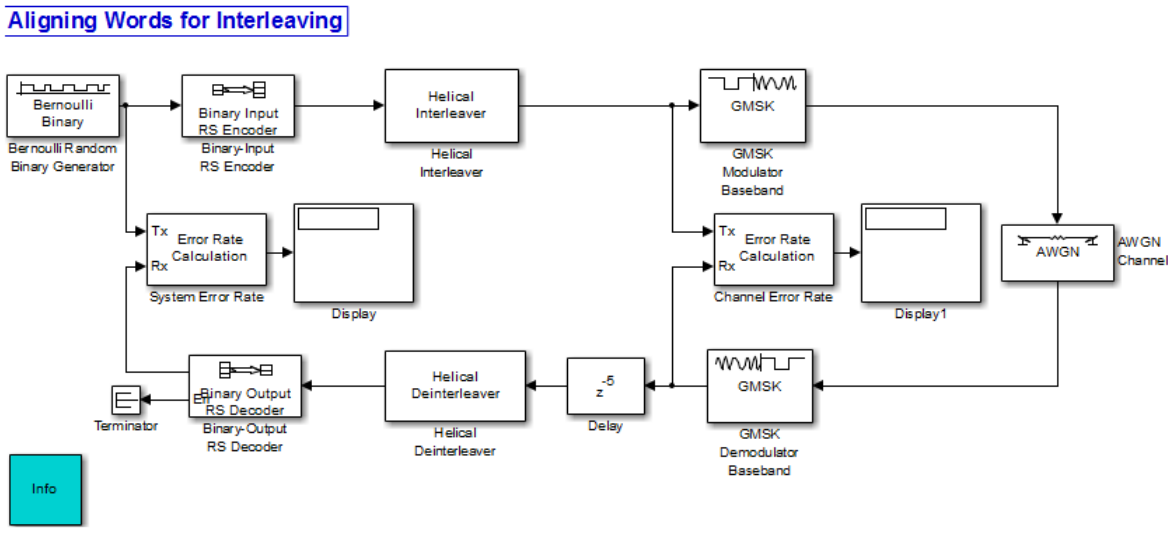
5 Show the dimensions of each signal in the model by selecting enabling **Display > Signals & Ports > Signal Dimensions**.

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.



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 from Individual Block	Reference
GMSK Demodulator Baseband	16	"Delays in Digital Modulation"
Helical Deinterleaver	42	"Delays of Convolutional Interleavers"
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 2-16, 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 17th 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.

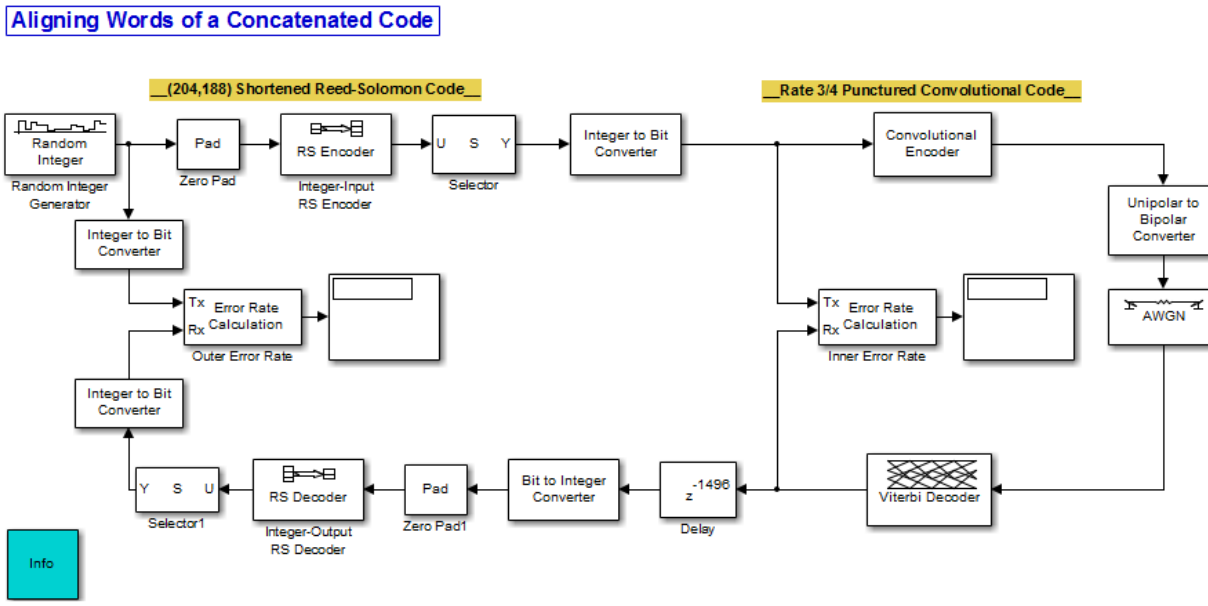
Computing Delays to Configure the Error Rate Calculation Blocks

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 samples
System Error Rate	`15*3`	Three fifteen-sample frames: one frame from the GMSK Demodulator Baseband and Delay blocks, and two frames from the interleaver/ deinterleaver 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.



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 2-16, 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

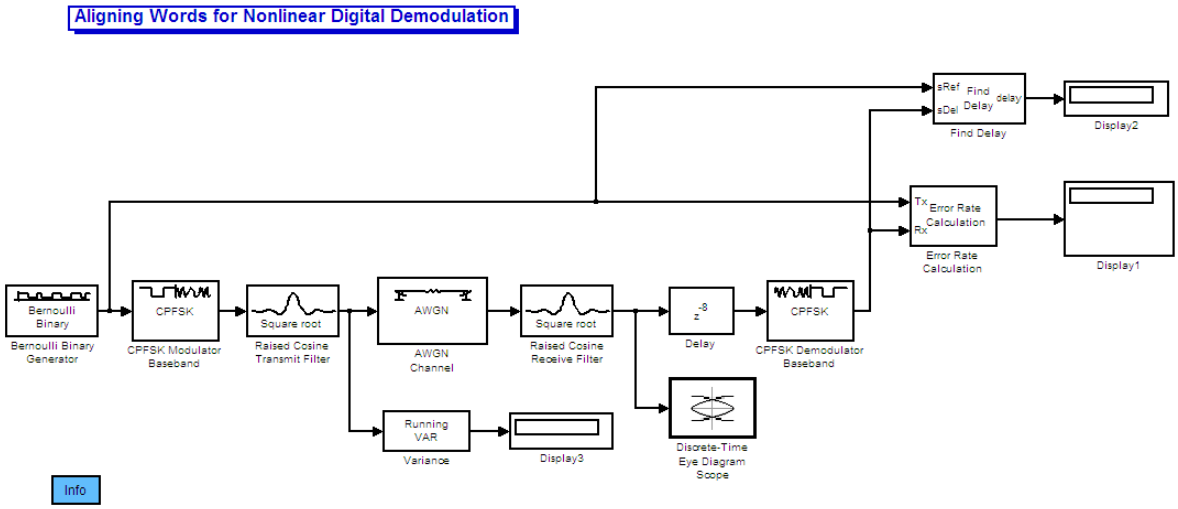
The model contains two Error Rate Calculation blocks, labeled Inner Error Rate and Outer Error Rate. Each of these blocks has a **Receive delay** parameter that must reflect the delay of the path between the block's `Tx` and `Rx` signals. The table below explains the **Receive delay** values in the two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



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: $GroupDelay \cdot Ts$

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:

$2 \cdot GroupDelay \cdot Ts$

or $8 \cdot Ts$

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, $8 \cdot Ts$). The total delay at the input of the CPFSK demodulator is $16 \cdot Ts$, 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” in the Communications System Toolbox *User's Guide*.

3

Adaptive Equalizer Examples

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
rng default       % Set random number generator for repeatability
```

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, ...
    'PhaseOffset',0, ...
    'SymbolMapping','Binary');
hPSKDemod = comm.PSKDemodulator(M, ...
    'PhaseOffset',0, ...
    'SymbolMapping','Binary');

PSKConstellation = constellation(hPSKMod).'; % PSK constellation
```

Training and Tail Sequences

Generate the training and tail sequences.

```
xTrainData = randi([0 M-1],nTrain,1);
xTailData  = randi([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

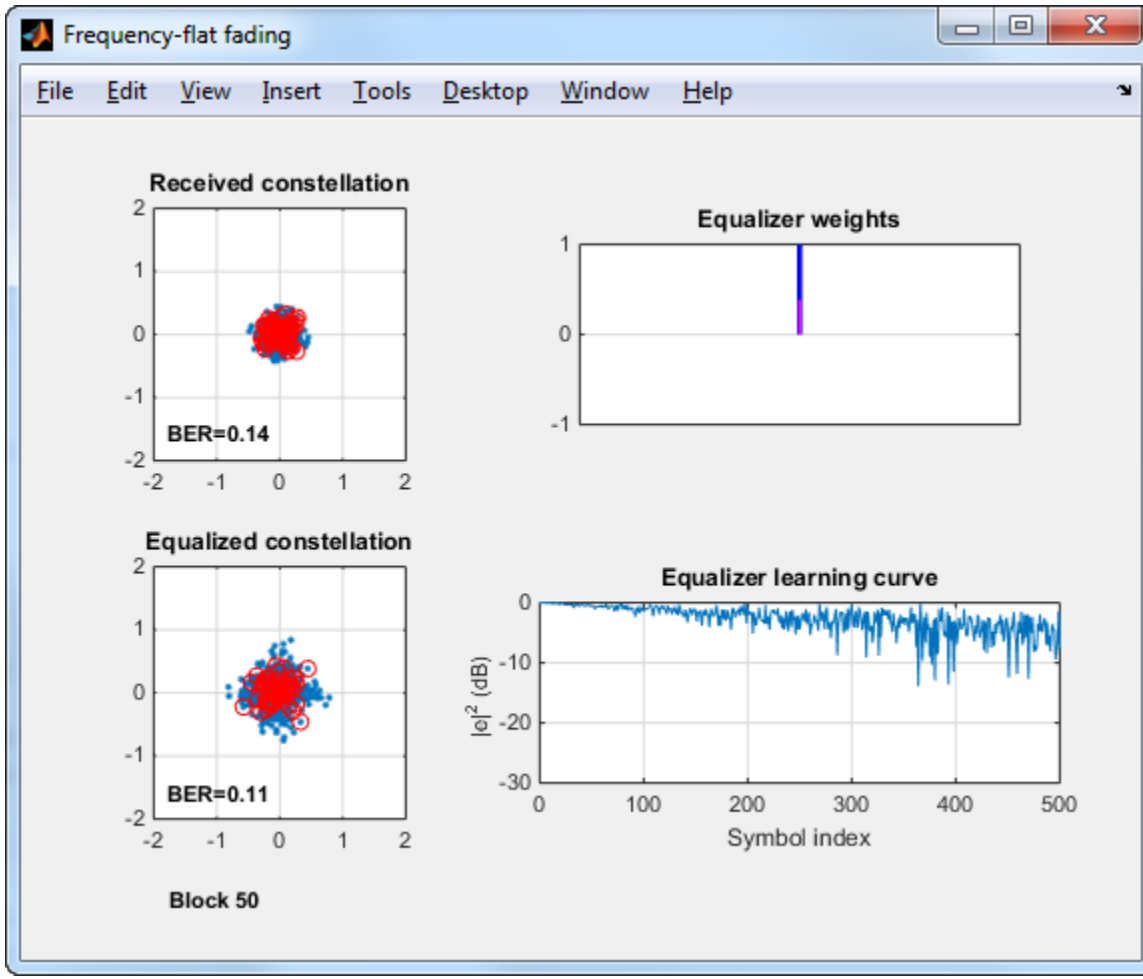
Begin with single-path, frequency-flat fading channel. For this channel, the receiver uses a simple 1-tap LMS (least mean square) equalizer, which implements automatic gain and phase control.

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 win

% 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: [1x1 struct]
       RandomStream: 'mt19937ar with seed'
               Seed: 73
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
           StepSize: 0.1000
       LeakageFactor: 1
             Weights: 0
        WeightInputs: 0
ResetBeforeFiltering: 1
 NumSamplesProcessed: 0
```



Simulation 2: Linear Equalization for Frequency-Selective Fading

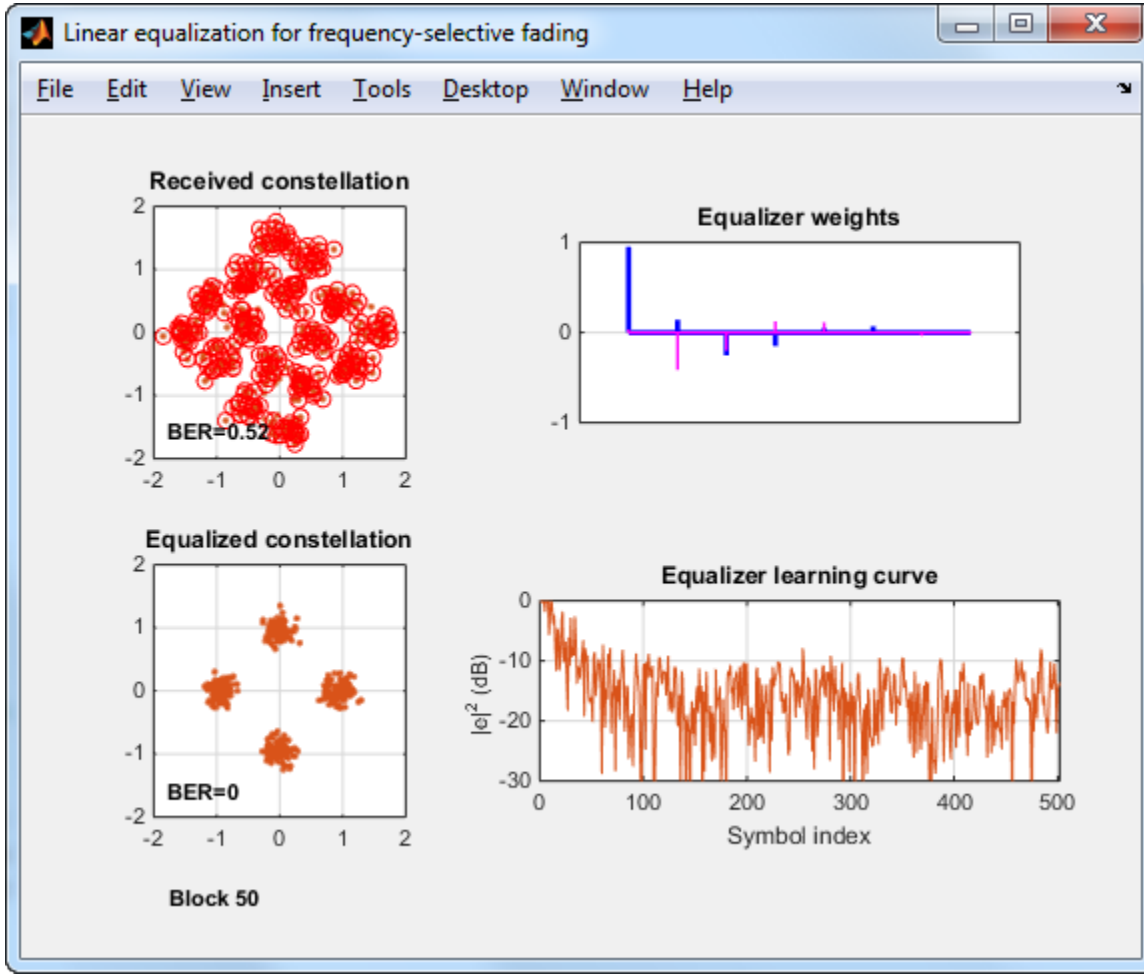
Simulate a three-path, frequency-selective Rayleigh fading channel. The receiver uses an 8-tap linear RLS (recursive least squares) equalizer with symbol-spaced taps.

```
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
            ForgetFactor: 0.9900
             InvCorrInit: 0.1000
           InvCorrMatrix: [8x8 double]
```

```
                Weights: [0 0 0 0 0 0 0 0]
           WeightInputs: [0 0 0 0 0 0 0 0]
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0


avgBER2 =

   3.0000e-04
```



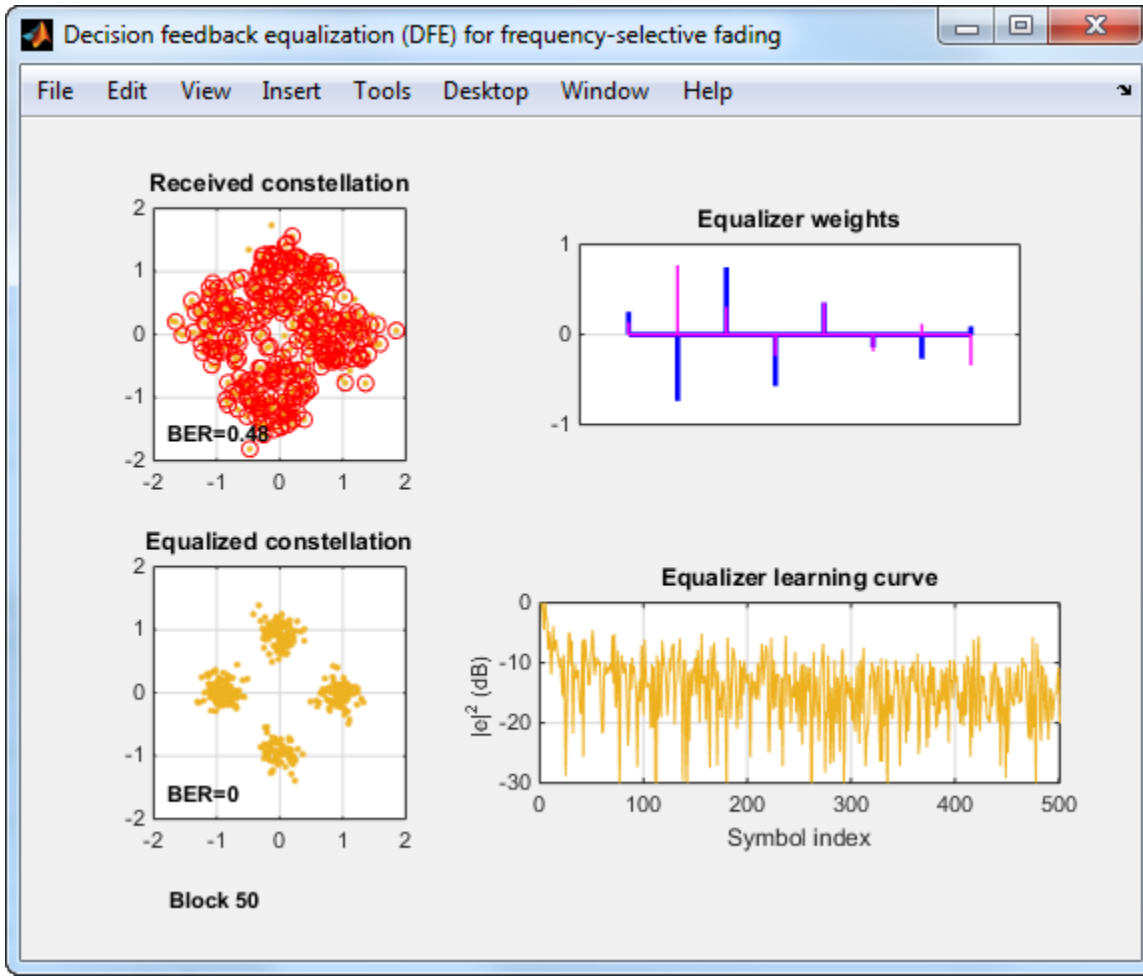
Simulation 3: Decision feedback Equalization (DFE) for Frequency-Selective Fading

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 -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
             ForgetFactor: 0.9900
              InvCorrInit: 0.1000
            InvCorrMatrix: [8x8 double]
                  Weights: [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

Adaptive Equalization

This model shows the behavior of adaptive equalizer algorithms at a receiver for modulated data transmitted along a channel.

Structure of the Example

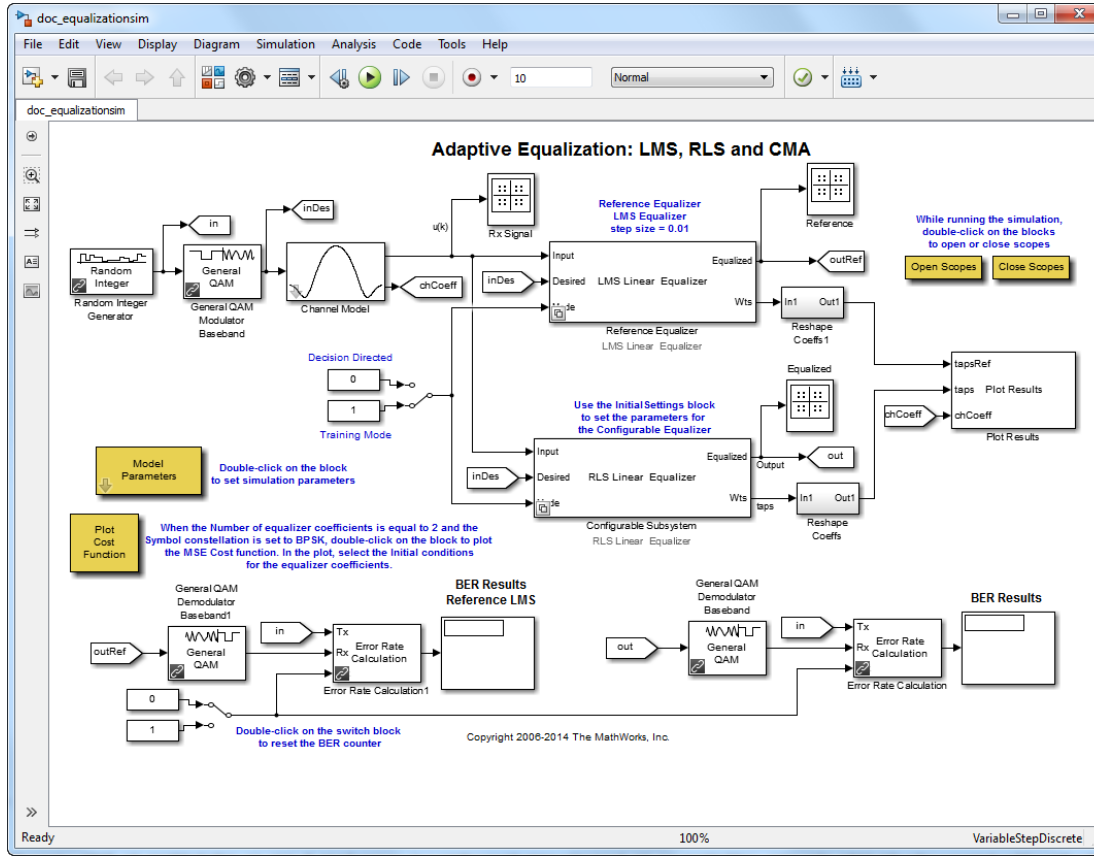
The example includes two equalizers, a reference equalizer that uses the least means square (LMS) algorithm and a configurable equalizer whose algorithm you can select from these choices:

- Least Mean Square (LMS)
- Sign LMS
- Normalized LMS
- Variable Step-Size LMS
- Recursive Least Squares (RLS)
- Constant Modulus Algorithm (CMA)

The example also creates plots that can help you understand how different algorithms behave.

On the MATLAB command line, open the model doc_equalizationsim.

```
doc_equalizationsim
```



Experimenting with the Example

This example provides several ways for you to change settings and observe the results.

Initial Settings

The Model Parameters block enables you to vary some parameters of the model, including

- The algorithm for the configurable equalizer
- The modulation scheme (symbol constellation)
- Channel coefficients

- The number of coefficients, or taps, in both equalizers

To access these parameters, double-click the Model Parameters block.

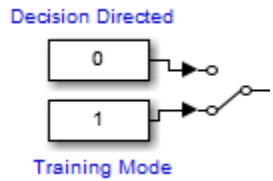
Cost Function and Initial Conditions

You can choose an initial set of weights for the equalizers when the Model Parameters block has the **Number of equalizer coefficients** set to 2 and the **Symbol Constellation** set to `BPSK`. To chose the initial set of weights, use this procedure:

1. Double-click the Plot Cost Function block to open a contour plot of the `MSE` cost function (as well as the constant modulus cost function if you selected `CMA` as the algorithm for the configurable equalizer).
2. Click in the plotting window to choose an initial set of weights for the equalizers in the model. Your choice takes effect the next time you run the simulation.

Equalizer Mode

During the simulation, each of the equalizer types (other than CMA) is capable of operating in training mode or decision-directed mode. In training mode, the desired symbol sequence exactly matches the transmitted symbol sequence (i.e., the receiver has knowledge of the transmitted data in this mode). In decision-directed mode, the "desired" symbols are derived from the output of the decision device. You can toggle between training and decision-directed mode by double-clicking the Switch block in the model.



Results and Displays

Error Statistics

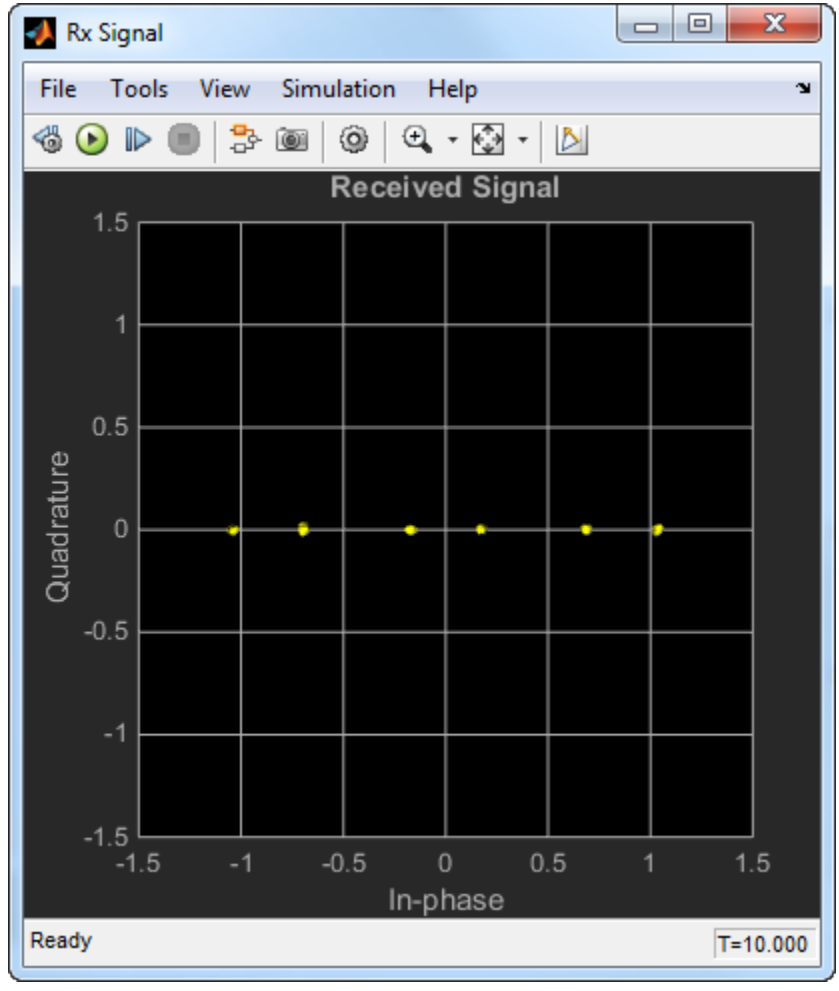
When you run the simulation, the display labeled BER Results Reference LMS shows error statistics for the link with the reference equalizer, while the display labeled BER Results shows error statistics for the link with the configurable equalizer. In particular, each set of error statistics is a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

You can reset the BER statistics during the simulation by double-clicking the Switch block connected to the **Rst** port of the Error Rate Calculation blocks.

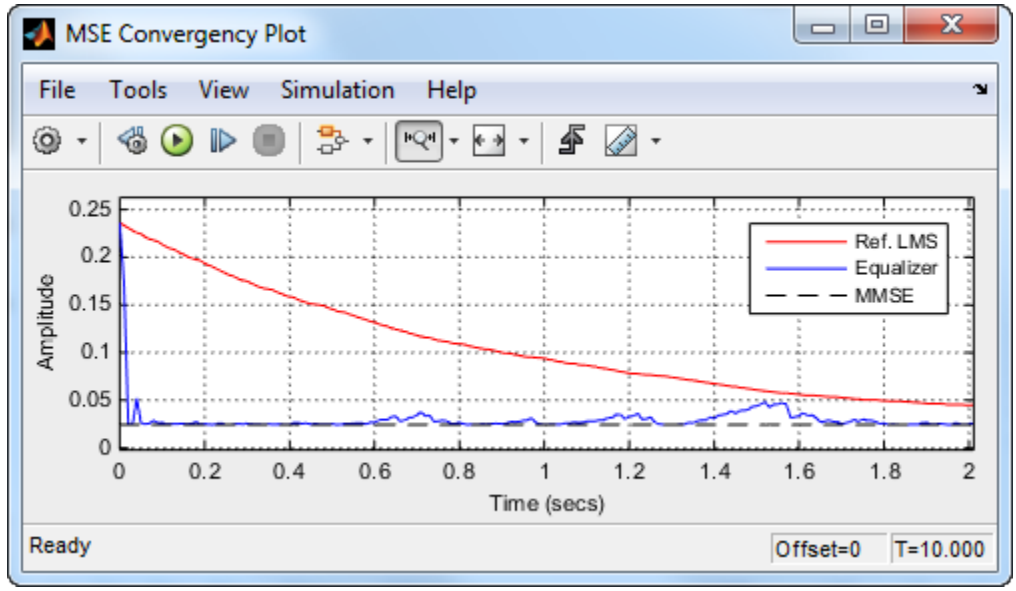
Scope Windows

During the simulation the model creates plots that show:

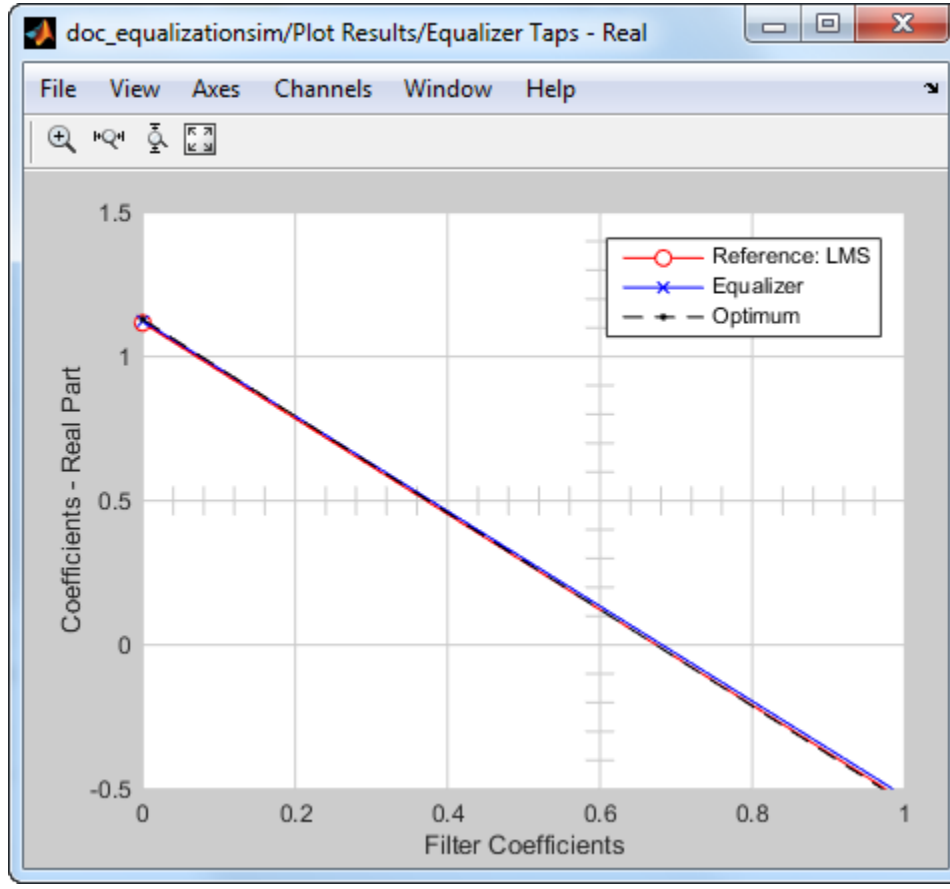
- A scatter plot of the received signal at the output of the channel.



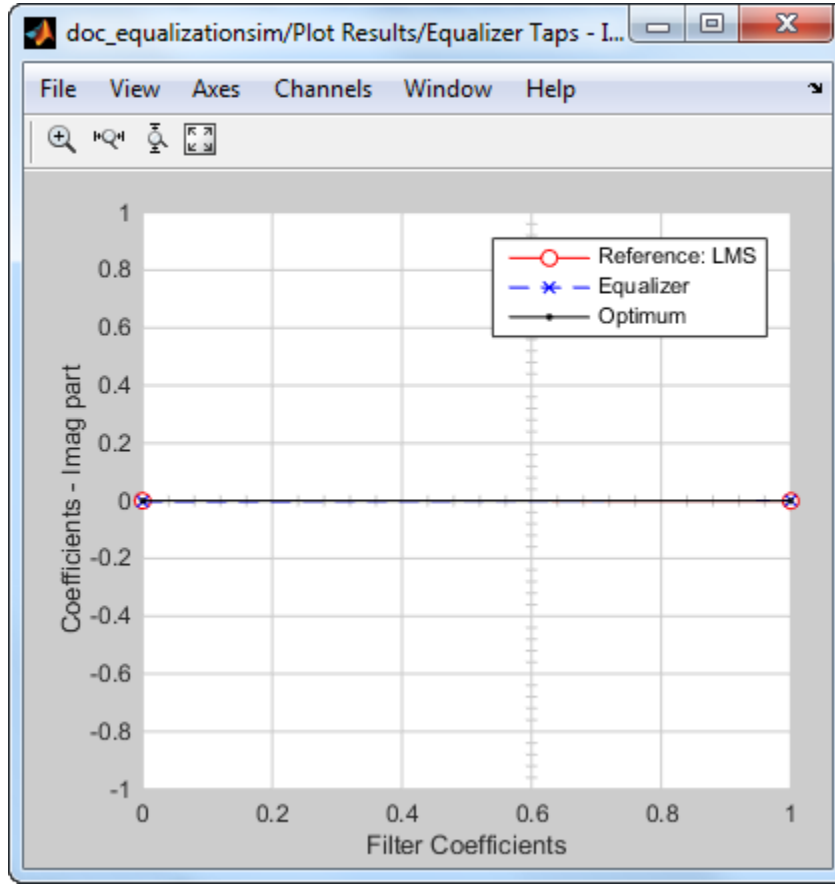
- An MSE convergence plot where you can see that the equalizers' cost functions converge to the minimum MSE.



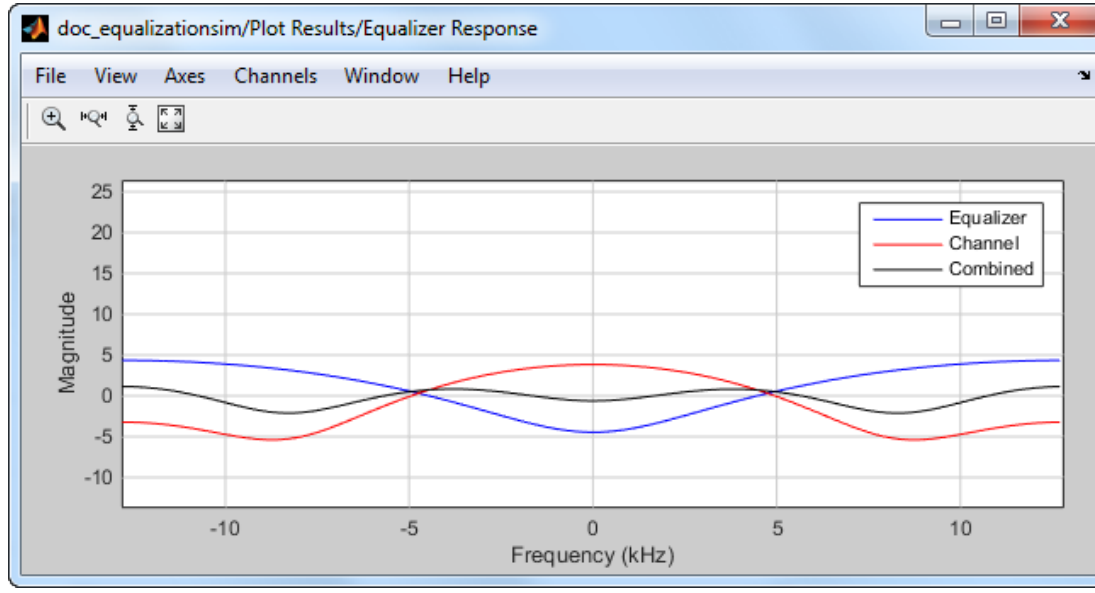
- The real part of the weights for the reference, configurable, and optimum equalizers.



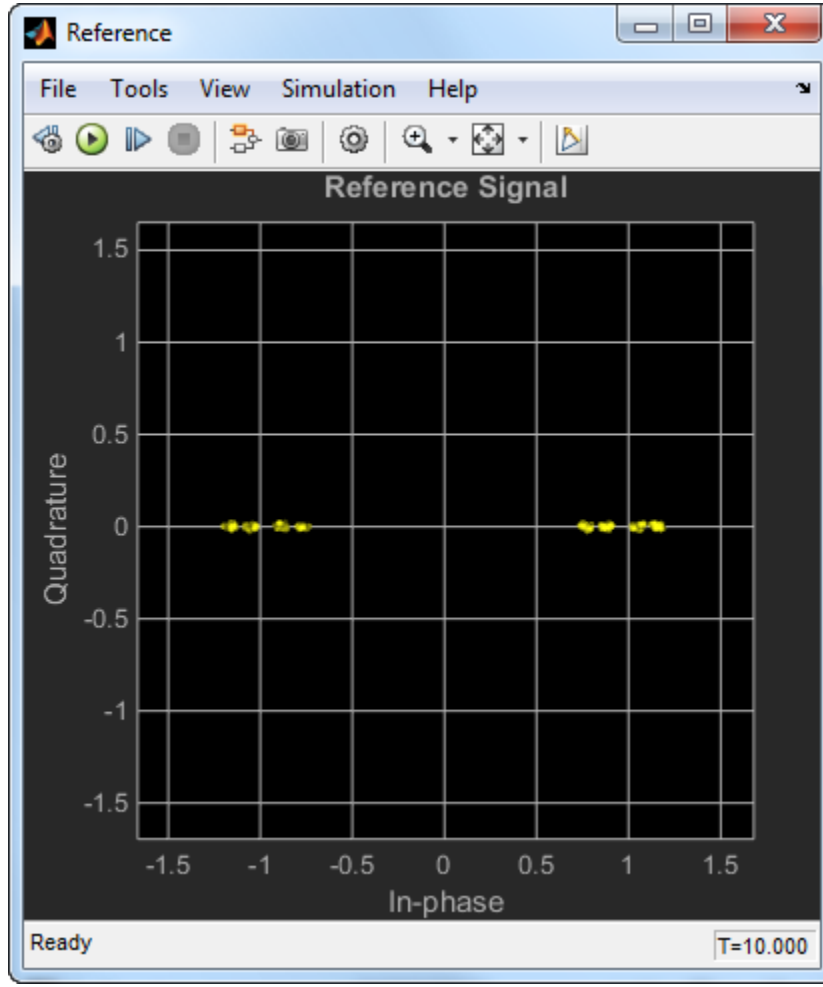
- The imaginary part of the weights for the reference, configurable, and optimum equalizers.



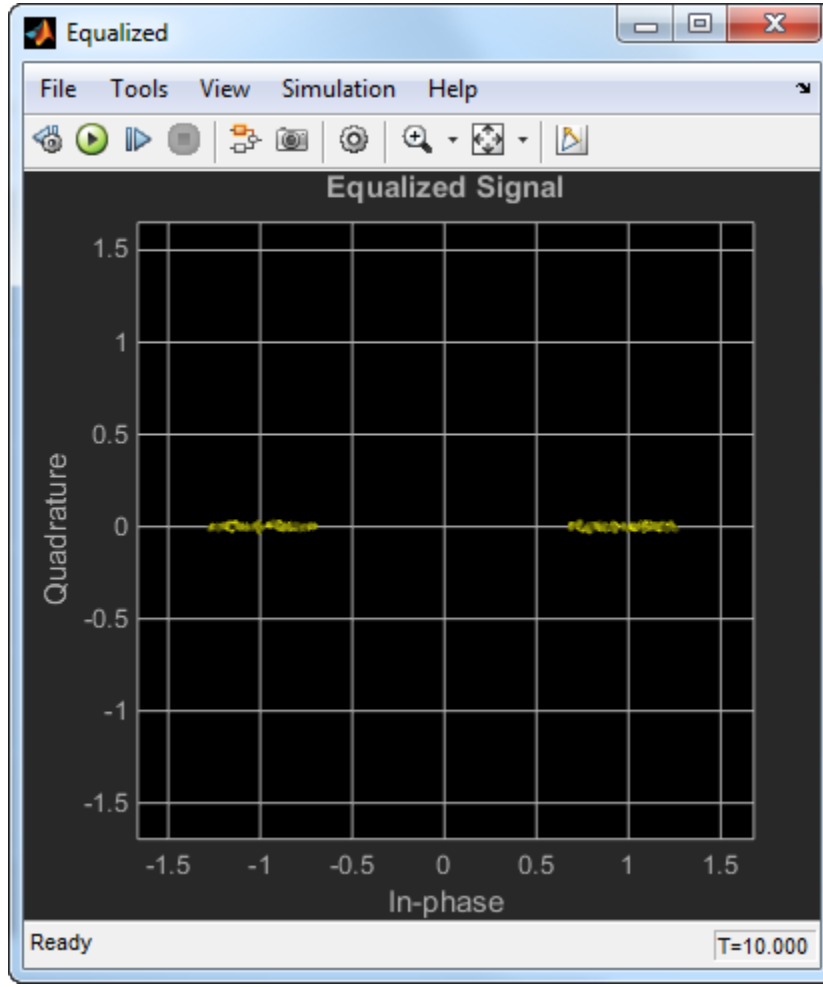
- The frequency responses of the channel, the channel after equalization (combined), and the equalizer itself. You can see that the frequency response of the equalizer is roughly the inverse of the channel response and that the post equalization or combined response is flatter.



- A scatter plot of the signal equalized by the reference equalizer.

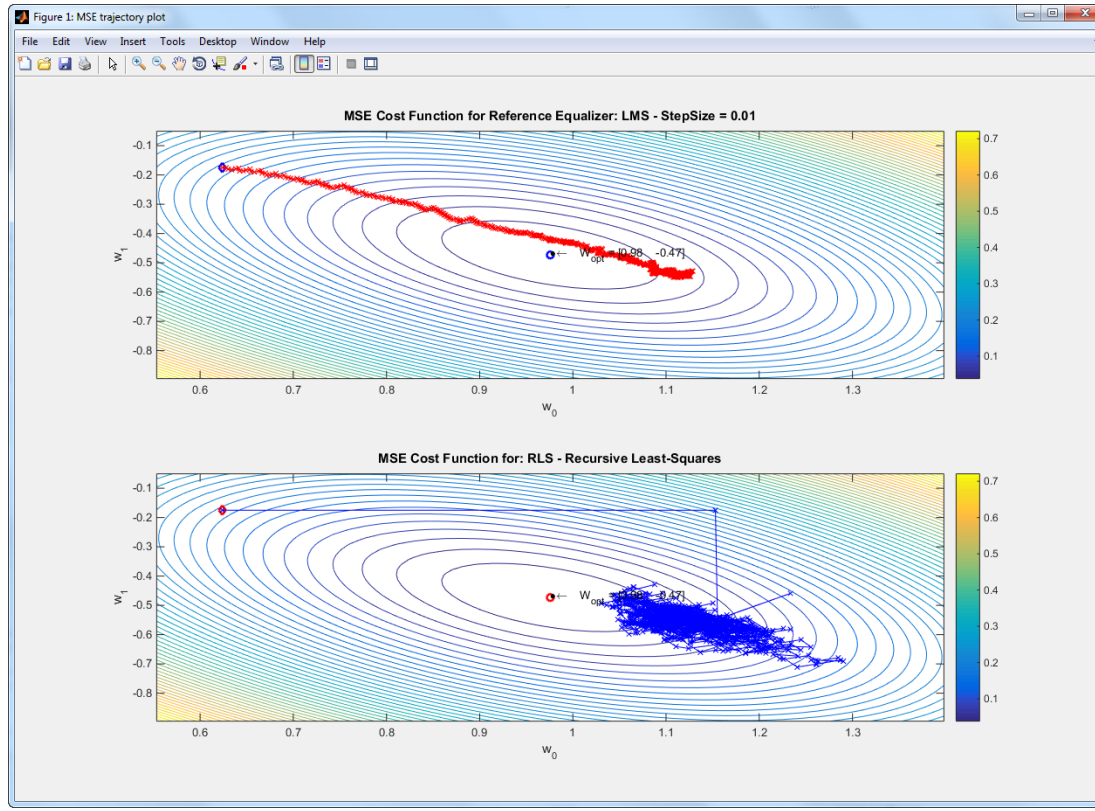


- A scatter plot of the signal equalized by the configurable equalizer.



- The cost functions for the equalizers, on the same axes with the minimum MSE. When the **Number of equalizer coefficients** parameter in the Model Parameters block is set to 2 and the **Symbol Constellation** parameter is set to BPSK, the model produces an additional plot at the end of a simulation. The new plot shows the trajectory of the two-element weight vector for each of the equalizers. On the same set of axes is a contour plot of the MSE cost function (or the constant modulus cost function, in case you selected CMA as the algorithm for the configurable equalizer).

You can see from the plot how the adaptive algorithm causes the weights to change so as to minimize the cost function.



The simulation runs more slowly when it is updating all the plots. To close the plotting windows and speed up the simulation, double-click the icon labeled Close Scopes.

To generate executable code, you will need to comment out the Plot Results subsystem, as it does not support code generation. Use `set_param('doc_equalizersim/Plot Results','Commented','on')` to do this and generate code for the model.

Selected Bibliography

[1] Haykin, S., *Adaptive Filter Theory*, Third Edition, Upper Saddle River, NJ, Prentice Hall, 1996.

[2] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chicester, England, Wiley, 1998.

[3] Johnson, C. R., et al., "Blind Equalization Using the Constant Modulus Criterion: A Review", Proc. IEEE, Vol. 86, No. 10, October 1998.

System Design

Source Coding

In this section...

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 and a codebook.

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

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    0.0024
```

Differential Pulse Code Modulation

Section Overview

The quantization in the section "Quantize a Signal" on page 4-13 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 4-2 and "Represent Codebooks" on page 4-3, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

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

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

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) +...+ p(m-1)x(k-m+1) + p(m)x(k-m)
```

then the corresponding predictor vector for toolbox functions is

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

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

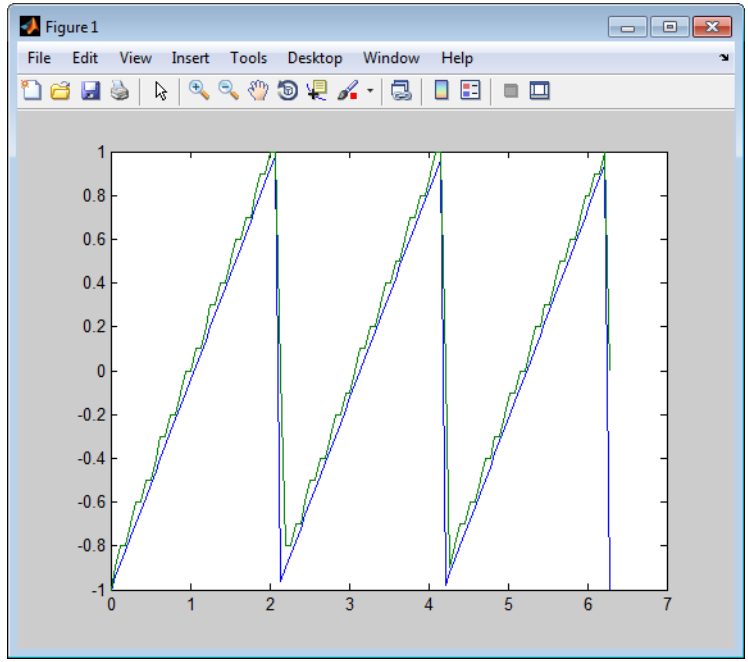
Example: DPCM Encoding and Decoding

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

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

The output is

```
distor =

    0.0327
```



Optimize DPCM Parameters

Section Overview

The section "Optimize Quantization Parameters" on page 4-4 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

Section Overview

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

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

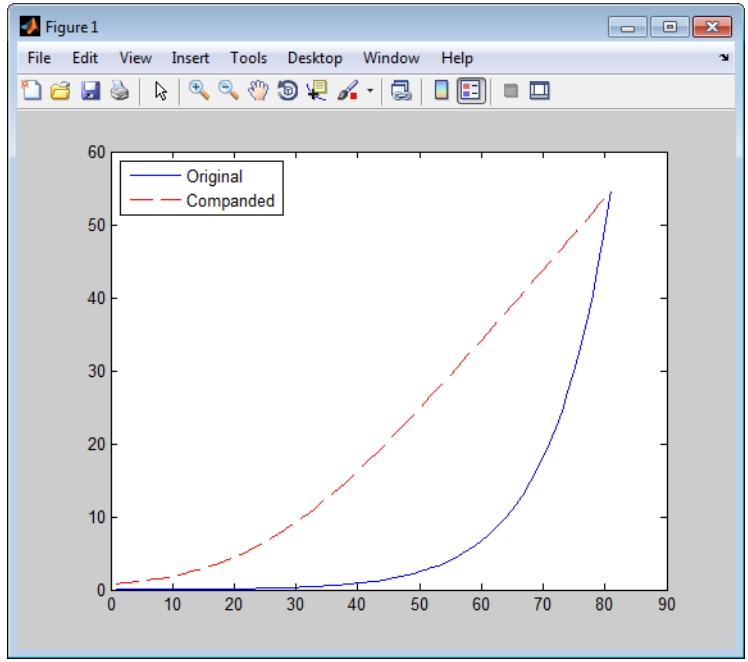
Example: μ-Law Compander

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

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

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

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

The output and figure are below.

```
ans =

    0.5348    0.0397
```



Huffman Coding

Section Overview

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

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

Note: For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding. To learn how to use arithmetic coding, see "Arithmetic Coding" on page 4-12.

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

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

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

```
dict =

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


ans =

     1


ans =

     1     1
```

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

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 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

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.

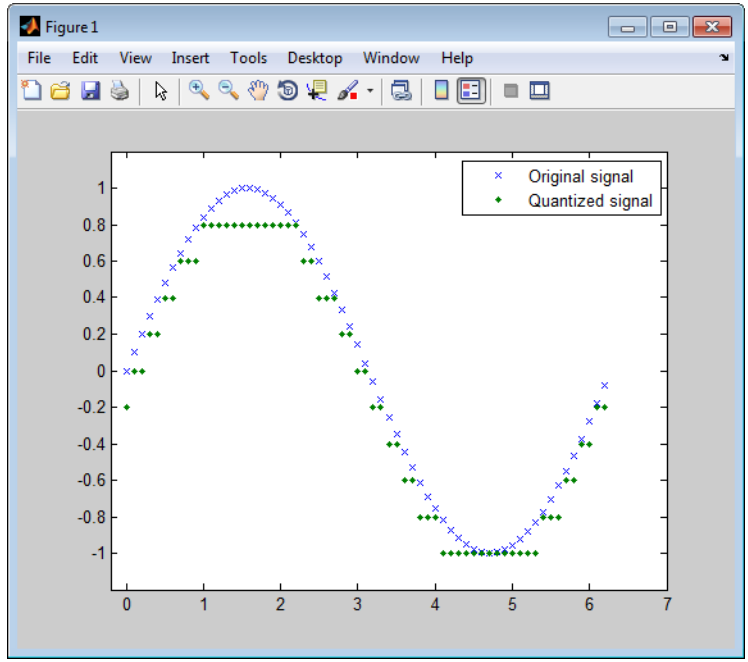
```
quantized =

  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])
```



Error Detection and Correction

In this section...

Cyclic Redundancy Check Codes

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

Communications System Toolbox supports CRC Coding using Simulink blocks, System objects, or MATLAB objects.

Blocks

The CRC block library contains four blocks that implement the CRC algorithm:

- General CRC Generator
- General CRC Syndrome Detector
- CRC-N Generator
- CRC-N Syndrome Detector

The General CRC Generator block computes a checksum for each input frame, appends it to the message word, and transmits the result. The General CRC Syndrome Detector block receives a transmitted word and calculates its checksum. The block has two outputs. The first is the message word without the transmitted checksum. The second output is a binary error flag, which is 0 if the checksum computed for the received word is zero, and 1 otherwise.

The CRC-N Generator block and CRC-N Syndrome Detector block are special cases of the General CRC Generator block and General CRC Syndrome Detector block, which use a predefined CRC-N polynomial, where N is the number of bits in the checksum.

See the General CRC Generator block Reference page for an example of Cyclic Redundancy Check Encoding.

System objects

The following System objects implement the CRC algorithm:

- `comm.CRCDetector`
- `comm.CRCGenerator`

These reference pages contain examples demonstrating the use of the object.

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 $T = M*x^r + C$, since multiplying by x^r corresponds to shifting the input vector r bits to the left. The algorithm chooses the checksum C so that T is divisible by a predefined polynomial P of degree r, called the *generator polynomial*.

The algorithm divides T by P, and sets the checksum equal to the binary vector corresponding to the remainder. That is, if $T = Q*P + R$, where R is a polynomial of degree less than r, the checksum is the binary vector corresponding to R. If necessary, the algorithm prepends zeros to the checksum so that it has length r.

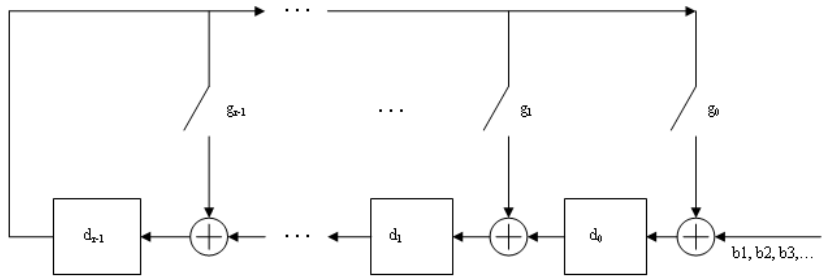
The CRC generation feature, which implements the transmission phase of the CRC algorithm, does the following:

1. Left shifts the input data vector by r bits and divides the corresponding polynomial by P.
2. Sets the checksum equal to the binary vector of length r, corresponding to the remainder from step 1.
3. Appends the checksum to the input data vector. The result is the output vector.

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

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

Note The implementation described in this section is one of many valid implementations of the CRC algorithm. Different implementations can yield different numerical results.

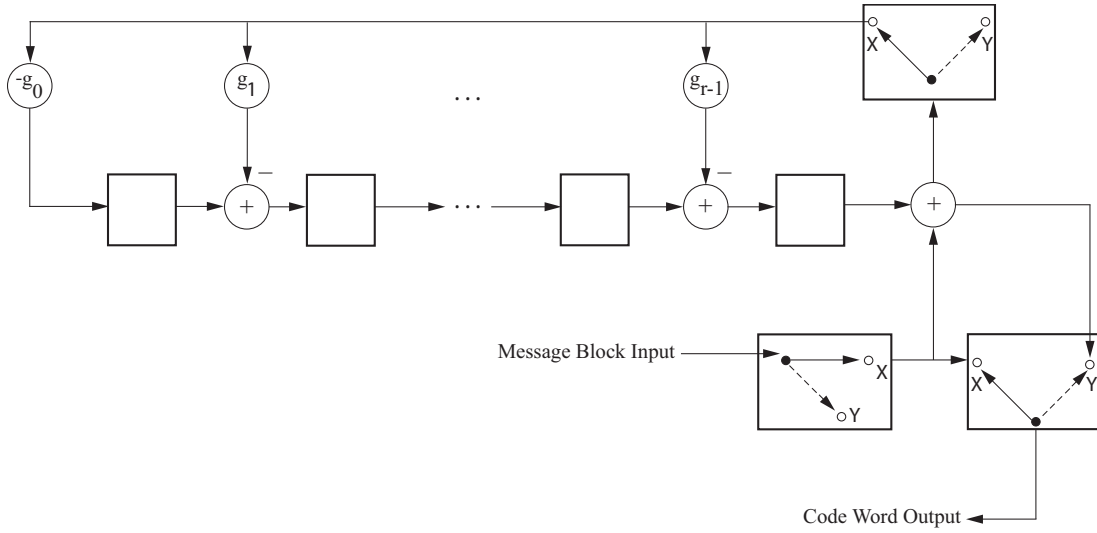


Bits enter the linear feedback shift register (LFSR) from the lowest index bit to the highest index bit. The sequence of input message bits represents the coefficients of a message polynomial in order of decreasing powers. The message vector is augmented with *r* zeros to flush out the LFSR, where *r*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

Example Using CRC Non-Direct Algorithm

Suppose the input frame is `[1 1 0 0 1 1 0]'`, corresponding to the polynomial M = $x^6 + x^5 + x^2 + x$, and the generator polynomial is P = $x^3 + x^2 + 1$, of degree r = 3. By polynomial division, M*x^3 = ($x^6 + x^3 + x$)*P + x. The remainder is R = x, so that the checksum is then `[0 1 0]'`. 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 $$c_0, c_1, \ldots, c_{n-1} = \underbrace{m_0, m_1, \ldots, m_{k-1}}_{X}, \underbrace{d_0, d_1, \ldots, d_{n-k-1}}_{Y}$$

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 kth 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.

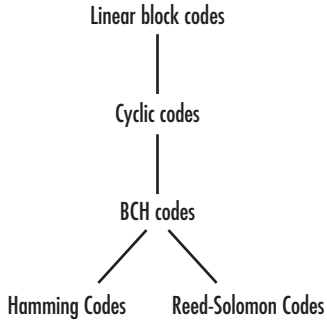
Block Codes

Block-Coding Features

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 System 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 System 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 this product are designed for error-control codes that use an alphabet having 2 or 2^m symbols.

Communications System Toolbox Support Functions

Functions in Communications System 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" in the *Communications System Toolbox User's Guide*.

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.

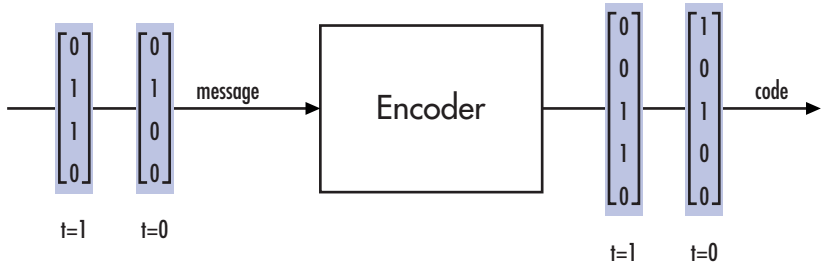
Data Formats for Block Coding

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 4-21. Reed-Solomon coding blocks also let you choose between binary and integer data, as described in "Integer Format (Reed-Solomon Only)" on page 4-22.

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 share the same sample time.

The figure below illustrates this situation. In this example, the encoder receives a four-bit message and produces 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 GF(2^M). In this case, the message vector must have length M*K and the corresponding code vector has length M*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

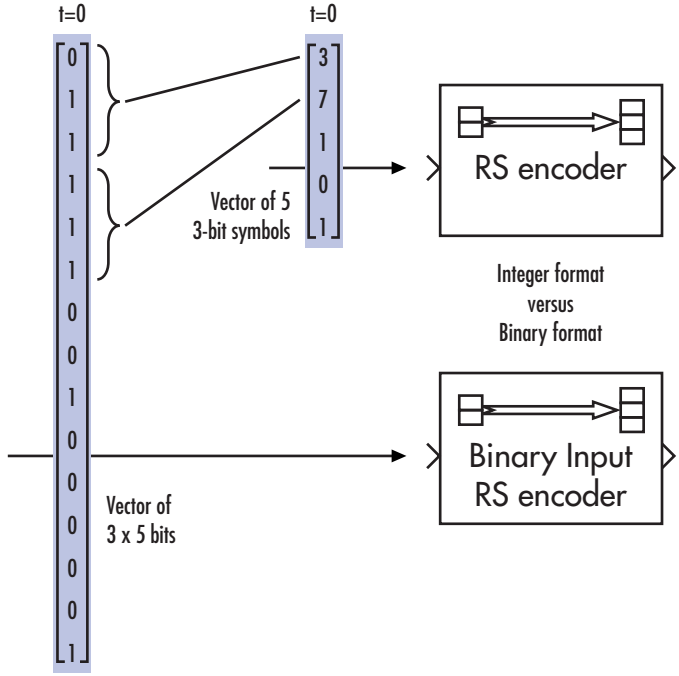
If you prefer to structure messages and codewords as scalar signals, where several samples jointly form a message word or codeword, you can use the Buffer and Unbuffer blocks in DSP System Toolbox. Be aware that buffering involves latency and multirate processing. See the reference page for the Buffer block for more details. If your model computes error rates, the initial delay in the coding-buffering combination influences the **Receive delay** parameter in the Error Rate Calculation block. If you are unsure about the sample times of signals in your model, click the **Display** menu and select **Sample Time > Colors**. Alternatively, attaching Probe blocks (from the Simulink Signal Attributes library) to connector lines might help.

Integer Format (Reed-Solomon Only)

A message word for an [N,K] Reed-Solomon code consists of M*K bits, which you can interpret as K symbols between 0 and 2^M. The symbols are binary sequences of length M, corresponding to elements of the Galois field GF(2^M), 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 Reed-Solomon 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 **M-ary number** vector are 2^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.

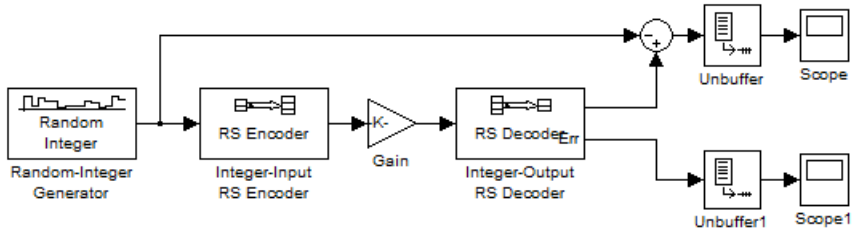
- Be sure the signal sizes are appropriate for the mask parameters. For example, if you use the Binary Cyclic Encoder block and set **Message length K** to `4`, the input signal must be a vector of length 4.

  If you are unsure about the size of signals in your model, clicking the **Display** menu select **Signals and Ports** >**Signal Dimension**.

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 very 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:

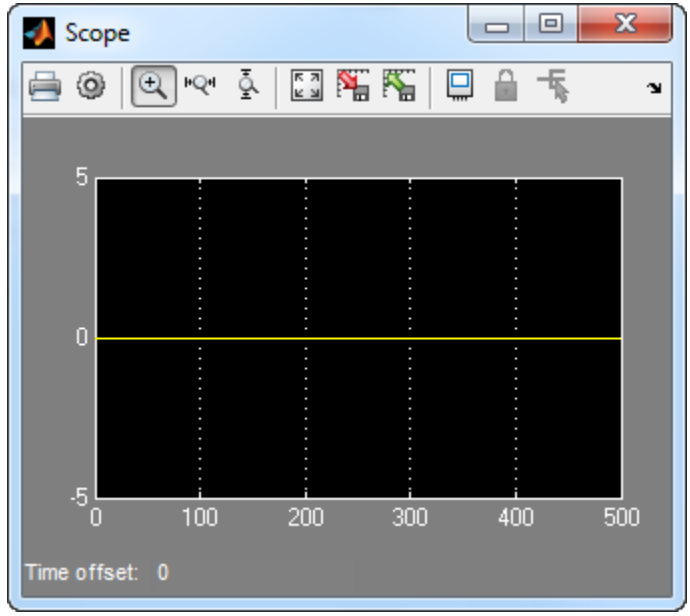
- Random Integer Generator, in the Comm Sources library

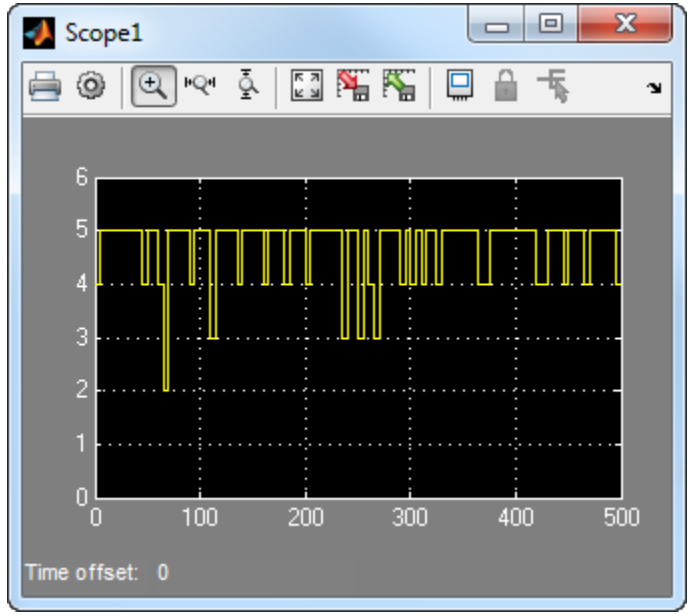
    - Set **M-ary number** to `15`.
    - Set **Initial seed** to a positive number, randseed(0) is chosen here.
    - Check the **Frame-based outputs** check box.
    - Set **Samples per frame** to `5`.
- Integer-Input RS Encoder
    - Set **Codeword length N** to `15`.
    - Set **Message length K** to `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 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 in the preceding figure. From the model window's **Simulation** menu, select **Model Configuration Parameters**. In the Configuration Parameters dialog box, set **Stop Time** to `500`.

The vector length numbers appear on the connecting lines only if you click the **Display** menu and select **Signals & Ports** > **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. Your plot of the error counts might differ somewhat, depending on your **Initial Seed** value in the Random Integer Generator block. (To make the axis range exactly match that of the first scope, right-click the plot area in the scope and select **Axes Properties**.)


Scope
5
0
-5
0
100
200
300
400
500
Time offset: 0



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. Read the part that applies to the coding technique you want to use: generic linear block code, cyclic code, Hamming code, BCH code, or 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. In order 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 [N,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 *vG*. If *G* has the form [I_k, *P*] or [*P*, I_k], where *P* is some K-by-(N-K) matrix and I_k is the K-by-K identity matrix, *G* is said to be in *standard form*. (Some authors, 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 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.

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^{N-K} rows. Each row gives a correction vector for one received codeword vector.

If you do not want to specify a decoding table explicitly, set that parameter to `0`. This causes the block to compute a decoding table using the `syndtable` function in Communications System Toolbox.

Cyclic Codes

For cyclic codes, the codeword length N must have the form 2^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^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 polynomial's coefficients in order of *ascending* powers of the variable. You can find generator polynomials for cyclic codes using the `cyclpoly` function in Communications System Toolbox.

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^M-1, where M is an integer greater than or equal to 3. The message length K must equal N-M.

Primitive Polynomials

Hamming codes rely on algebraic fields that have 2^M elements (or, more generally, p^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 polynomial's coefficients in order of *ascending* powers of the variable. You can find generator polynomials for Galois fields using the `gfprimfd` function in Communications System Toolbox.

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^M-1, where M is an integer between 3 and 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), m_2(x), ..., m_2t(x)], where:

- LCM represents the least common multiple,
- m_i(x) represents the minimum polynomial corresponding to α^i, α is a root of the default primitive polynomial for the field GF(`n+1`),
- and t represents the error-correcting capability of the code.

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 $N= 2^M-1$, where the 2^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^M-1. In this case, the encoder appends 2^M-1-N zero symbols to each message word and codeword. The error-correction capability of a shortened Reed-Solomon code is also `floor((N-K)/2))`. The Communications System 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 $GF(2^M)$ instead of GF(2). Each such symbol is specified by M bits. The nonbinary nature of the Reed-Solomon code symbols causes the Reed-Solomon 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 M*K (not K) for messages and the same integer M*N (not N) for codewords.

Error Information

The Reed-Solomon decoding blocks (Binary-Output RS Decoder and Integer-Output 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.

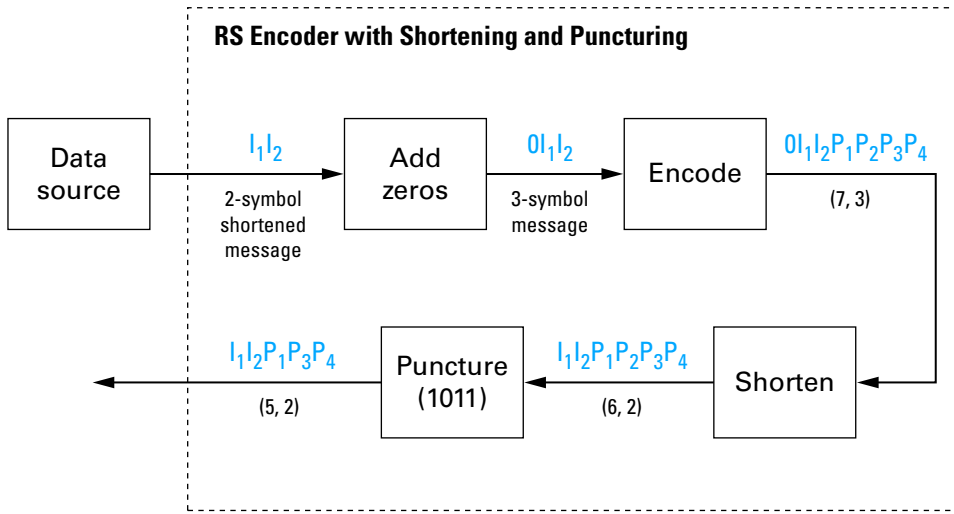
A *punctured codeword* has only parity symbols removed, and a *shortened codeword* has only information symbols removed. A codeword with erasures can have those erasures in either information symbols or parity symbols.

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.



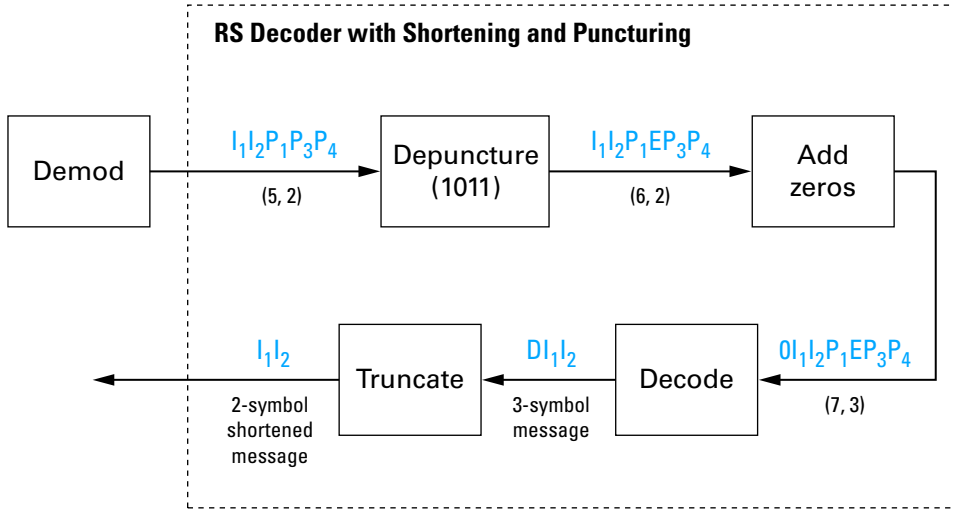
In this figure, the message source outputs two information symbols, designated by I_1I_2. (For a BCH example, the symbols are simply binary bits.) Because the code is a shortened (7,3) code, a zero must be added ahead of the information symbols, yielding

a three-symbol message of $0I_1I_2$. The modified message sequence is then RS encoded, and the added information zero is subsequently removed, which yields a result of $I_1I_2P_1P_2P_3P_4$. (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 $I_1I_2P_1P_3P_4$.

Decoder Example with Shortening and Puncturing.

The following figure shows how the RS dncoder operates on a shortened and punctured codeword.



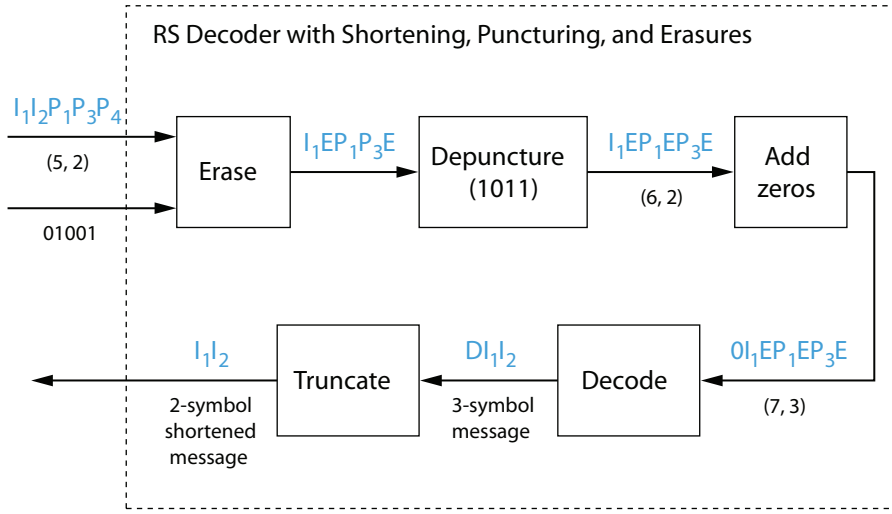
This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a (5,2) codeword, because it has been shortened from a (7,3) codeword by one symbol, and one symbol has also been punctured.

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 erasure-correcting capability of the code, which can correct

four erasures. The decoding operation results in the three-symbol message DI_1I_2. The first symbol is truncated, as in the preceding figure, yielding a final output of I_1I_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 $I_1I_2P_1P_3P_4$ 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 $I_1EP_1P_3E$, 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 $I_1EP_1EP_3E$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0I_1EP_1EP_3E$, such that a (7,3) codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of DI_1I_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_1I_2 vector.

For additional information, see the "Reed-Solomon Coding with Erasures, Punctures, and Shortening" 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"

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+x^4+x^2+x+1
genpolyRS = rsgenpoly(15,5)
```

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

```
genpolyCyclic =

     1     0     0     0     0     1     0     0     0     0     1


genpolyBCH = GF(2) array.

Array elements =

     1     0     1     0     0     1     1     0     1     1     1


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

Array elements =

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

The formats of these outputs vary:

- `cyclpoly` represents a generator polynomial using an integer row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable.
- `bchgenpoly` and `rsgenpoly` represent a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable.
- `rsgenpoly` uses coefficients in a Galois field other than the binary field GF(2). For more information on the meaning of these coefficients, see "How Integers Correspond to Galois Field Elements" on page 4-110 and "Polynomials over Galois Fields" on page 4-129.

Nonuniqueness of Generator Polynomials

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

Algebraic Expression for Generator Polynomials

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

Performing Other Block Code Tasks

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

- "Error Correction Versus Error Detection for Linear Block Codes" on page 4-35
- "Finding the Error-Correction Capability" on page 4-36
- "Finding Generator and Parity-Check Matrices" on page 4-36
- "Converting Between Parity-Check and Generator Matrices" on page 4-36

Error Correction Versus Error Detection for Linear Block Codes

You can use a liner block code to detect $d_{\min}$ -1 errors or to correct $t = \left[\frac{1}{2}(d_{\min} - 1)\right]$ 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^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

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 System 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 4-69.

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 System 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 4-55.
- If you design your encoder using a trellis diagram, you can construct the trellis structure in MATLAB and use it as the mask parameter.

Details about these representations are in the sections "Polynomial Description of a Convolutional Code" and "Trellis Description of a Convolutional Code" in the *Communications System Toolbox User's Guide*.

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 System 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]);
```

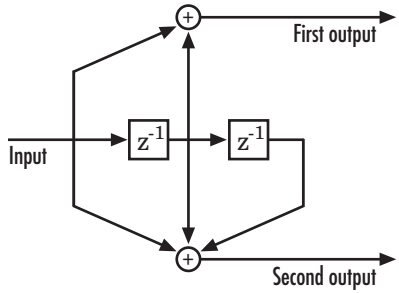
To use this encoder with one of the convolutional-coding blocks, simply place a `poly2trellis` command such as

```
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
- Generator polynomials
- Feedback connection polynomials (for feedback encoders only)

Constraint Lengths

The constraint lengths of the encoder form a vector whose length is the number of inputs in the encoder diagram. The elements of this vector indicate the number of bits stored in each shift register, *including* the current input bits.

In the figure above, the constraint length is three. It is a scalar because the encoder has one input stream, and its value is one plus the number of shift registers for that input.

Generator Polynomials

If the encoder diagram has k inputs and n outputs, the code generator matrix is a k-by-n matrix. The element in the ith row and jth column indicates how the ith input contributes to the jth output.

For *systematic* bits of a systematic feedback encoder, match the entry in the code generator matrix with the corresponding element of the feedback connection vector. See "Feedback Connection Polynomials" on page 4-40 below for details.

In other situations, you can determine the (i,j) entry in the matrix as follows:

1. Build a binary number representation by placing a 1 in each spot where a connection line from the shift register feeds into the adder, and a 0 elsewhere. The leftmost spot in the binary number represents the current input, while the rightmost spot represents the oldest input that still remains in the shift register.
2. Convert this binary representation into an octal representation by considering consecutive triplets of bits, starting from the rightmost bit. The rightmost bit in each triplet is the least significant. If the number of bits is not a multiple of three, place zero bits at the left end as necessary. (For example, interpret 1101010 as 001 101 010 and convert it to 152.)

For example, the binary numbers corresponding to the upper and lower adders in the figure above are 110 and 111, respectively. These binary numbers are equivalent to the octal numbers 6 and 7, respectively, so the generator polynomial matrix is [6 7].

Note: You can perform the binary-to-octal conversion in MATLAB by using code like `str2num(dec2base(bin2dec('110'),8))`.

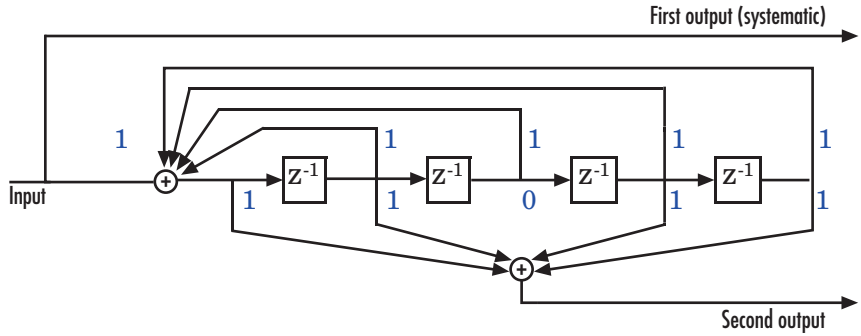
For a table of some good convolutional code generators, refer to [2] in the section "Selected Bibliography for Block Coding" on page 4-36, especially that book's appendices.

Feedback Connection Polynomials

If you are representing a feedback encoder, you need a vector of feedback connection polynomials. The length of this vector is the number of inputs in the encoder diagram. The elements of this vector indicate the feedback connection for each input, using an octal format. First build a binary number representation as in step 1 above. Then convert the binary representation into an octal representation as in step 2 above.

If the encoder has a feedback configuration and is also systematic, the code generator and feedback connection parameters corresponding to the systematic bits must have the same values.

For example, the diagram below shows a rate 1/2 systematic encoder with feedback.



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

The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits. The feedback polynomial is represented by the binary vector [1 1 1 1 1], corresponding to the upper row of binary digits in the diagram. These digits indicate connections from the outputs of the registers to the adder. The initial 1 corresponds to the input bit. The octal representation of the binary number 11111 is 37.

The second generator polynomial is represented by the binary vector [1 1 0 1 1], corresponding to the lower row of binary digits in the diagram. The octal number corresponding to the binary number 11011 is 33.

Using the Polynomial Description in MATLAB

To use the polynomial description with the functions `convenc` and `vitdec`, first convert it into a trellis description using the `poly2trellis` function. For example, the command below computes the trellis description of the encoder pictured in the section "Polynomial Description of a Convolutional Code" on page 4-39.

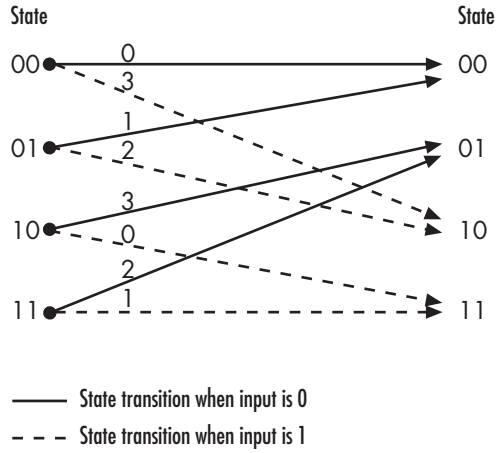
```
trellis = poly2trellis(3,[6 7]);
```

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 trellises in MATLAB, and gives an example of a MATLAB trellis.

The figure below depicts a trellis for the convolutional encoder from the previous section. The encoder has four states (numbered in binary from 00 to 11), a one-bit input, and a two-bit output. (The ratio of input bits to output bits makes this encoder a rate-1/2 encoder.) Each solid arrow shows how the encoder changes its state if the current input is zero, and each dashed arrow shows how the encoder changes its state if the current input is one. The octal numbers above each arrow indicate the current output of the encoder.



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

Note that any polynomial description of a convolutional encoder is equivalent to some trellis description, although some trellises have no corresponding polynomial descriptions.

Specifying a Trellis in MATLAB

To specify a trellis in MATLAB, use a specific form of a MATLAB structure called a trellis structure. A trellis structure must have five fields, as in the table below.

Fields of a Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputsymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

Note: While your trellis structure can have any name, its fields must have the *exact* names as in the table. Field names are case sensitive.

In the `nextStates` matrix, each entry is an integer between 0 and `numStates`-1. The element in the ith row and jth column denotes the next state when the starting state is i-1 and the input bits have decimal representation j-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the `nextStates` matrix stores the next states when the current set of input values is {0,...,0,1}. To learn how to assign numbers to states, see the reference page for `istrellis`.

In the `outputs` matrix, the element in the ith row and jth column denotes the encoder's output when the starting state is i-1 and the input bits have decimal representation j-1. To convert to decimal value, use the first output bit as the MSB.

How to Create a MATLAB Trellis Structure

Once you know what information you want to put into each field, you can create a trellis structure in any of these ways:

- Define each of the five fields individually, using `structurename.fieldname` notation. For example, set the first field of a structure called `s` using the command below. Use additional commands to define the other fields.

```
s.numInputSymbols = 2;
```

The reference page for the `istrellis` function illustrates this approach.

- Collect all field names and their values in a single `struct` command. For example:

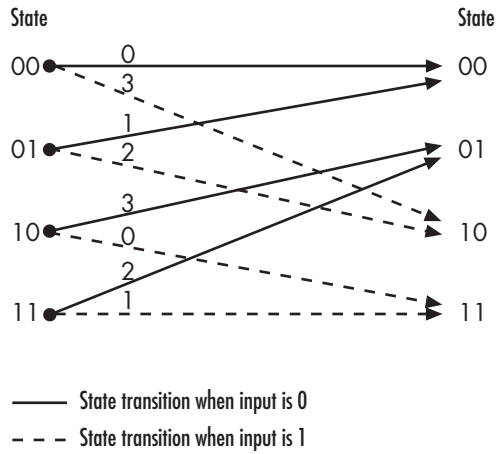
```
s = struct('numInputSymbols',2,'numOutputSymbols',2,...
   'numStates',2,'nextStates',[0 1;0 1],'outputs',[0 0;1 1]);
```

- Start with a polynomial description of the encoder and use the `poly2trellis` function to convert it to a valid trellis structure. The polynomial description of a convolutional encoder is described in "Polynomial Description of a Convolutional Code" on page 4-39.

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.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]);  % Define trellis.
hConvEnc = comm.ConvolutionalEncoder(t); % Create a ConvolutionalEncoder
                                         % System Object
code = step(hConvEnc, ones(100,1));      % Encode a string of ones.
```

The first command converts a polynomial description of a feedforward convolutional encoder to the corresponding trellis description. The second command encodes 100 bits, or 50 two-bit symbols. Because the code rate in this example is 2/3, the output vector `code` contains 150 bits (that is, 100 input bits times 3/2).

To check whether your trellis corresponds to a catastrophic convolutional code, use the `iscatastrophic` function.

Hard-Decision Decoding

To decode using hard decisions, use the `vitdec` function with the flag `'hard'` and with *binary* input data. Because the output of `convenc` is binary, hard-decision decoding can use the output of `convenc` directly, without additional processing. This example extends the previous example and implements hard-decision decoding.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Define trellis.
tb = 2; % Traceback length for decoding
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(t);
% Create a ViterbiDecoder System object
```

```
hVitDec = comm.ViterbiDecoder(t, 'InputFormat', 'hard', ...
    'TracebackDepth', tb, 'TerminationMethod', 'Truncated');
code = step(hConvEnc, ones(100,1));      % Encode a string of ones.
decoded = step(hVitDec, code); % Decode.
```

Soft-Decision Decoding

To decode using soft decisions, use the `vitdec` function with the flag `'soft'`. Specify the number, `nsdec`, of soft-decision bits and use input data consisting of integers between 0 and `2^nsdec-1`.

An input of 0 represents the most confident 0, while an input of `2^nsdec-1` represents the most confident 1. Other values represent less confident decisions. For example, the table below lists interpretations of values for 3-bit soft decisions.

Input Values for 3-bit Soft Decisions

Input Value	Interpretation
0	Most confident 0
1	Second most confident 0
2	Third most confident 0
3	Least confident 0
4	Least confident 1
5	Third most confident 1
6	Second most confident 1
7	Most confident 1

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 soft-decision decoding, the example uses `quantiz` to map the noisy data values to appropriate decision-value integers between 0 and 7. The second argument in `quantiz` is a partition vector that determines which data values map to 0, 1, 2, etc. The partition is chosen so that values near 0 map to 0, and values near 1 map to 7. (You can refine the partition to obtain better decoding performance if your application requires it.) Finally, the example decodes the code and computes the bit error rate. When comparing the decoded data with the original message, the example must take the decoding delay into account. The continuous operation mode of `vitdec` causes a delay

equal to the traceback length, so `msg(1)` corresponds to `decoded(tblen+1)` rather than to `decoded(1)`.

```
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 =
```

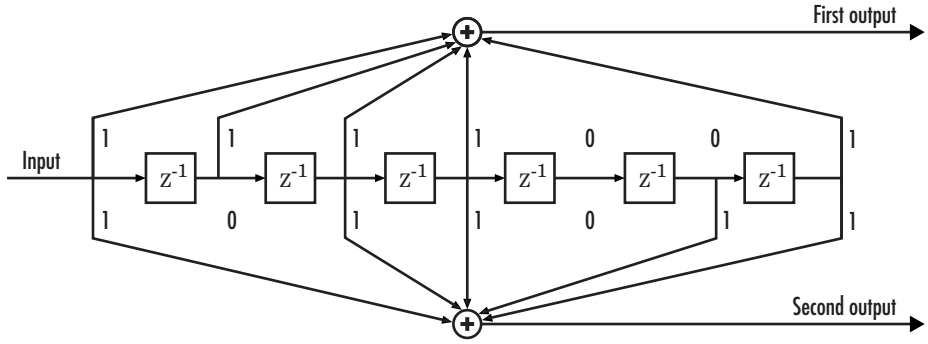
```
0.0013
```

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".

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 System 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 Soft-Output 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^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^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** 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 Error Rate Calculation block's **Receive delay** parameter 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 (49).

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_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_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_d is the pairwise error probability, given by

$$P_d = \frac{1}{2}\operatorname{erfc}\left(\sqrt{dR\frac{E_b}{N_0}}\right)$$

where R is the code rate of 1/2, and `erfc` is the MATLAB complementary error function, defined by

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2}\,dt$$

Values for the coefficients c_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_b for E_b/N_0 values in the range from 1 to 3.5, in increments of 0.5:

Simulating Multiple Times to Collect Bit Error Rates

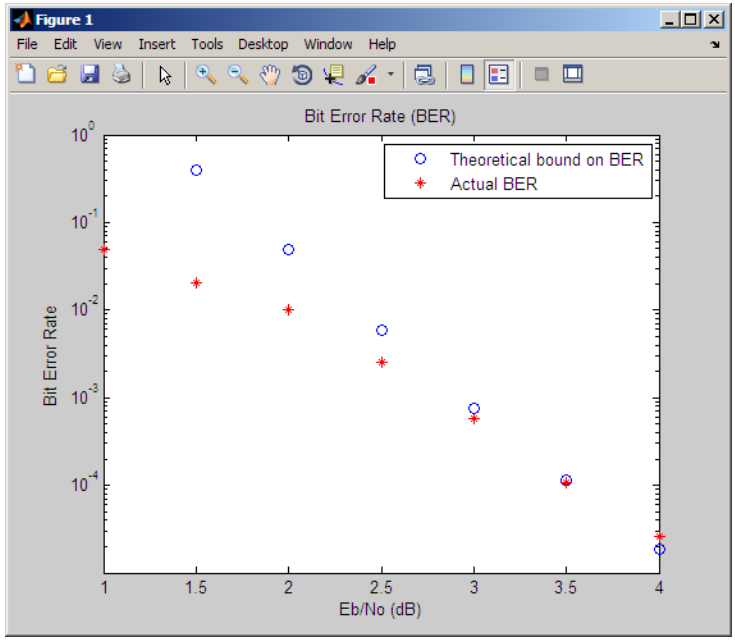
You can efficiently vary the simulation parameters by using the `sim` function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB, in increments of 0.5 dB. It collects all bit error 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 1e-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;
```

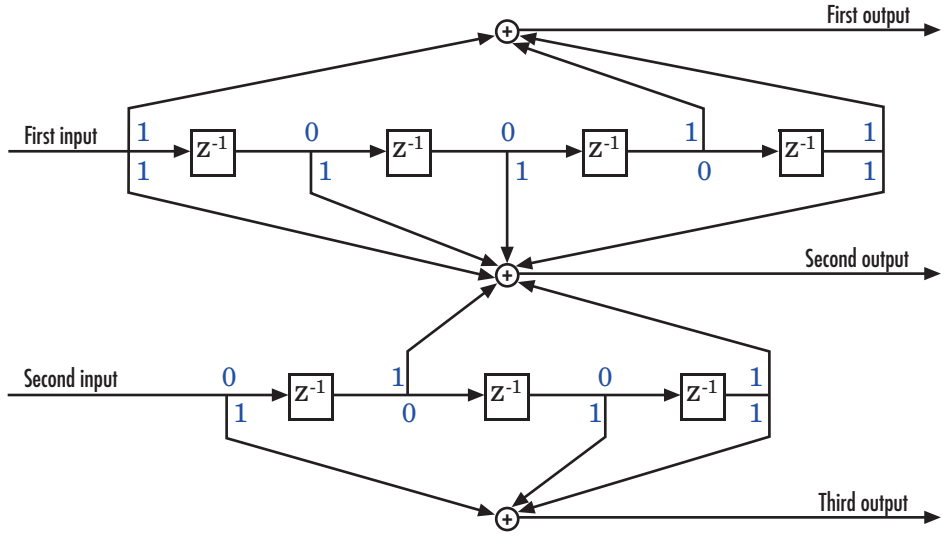
Note The estimate for P_b assumes that the decoder uses unquantized data, that is, an infinitely fine quantization. By contrast, the simulation in this example uses 8-level (3-bit) quantization. Because of this quantization, the simulated bit error rate is not quite as low as the bound when the signal-to-noise 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 [5 4].

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13. The full value of the code generator matrix is [23 35 0; 0 5 13].

To use the constraint length and code generator parameters in the `convenc` and `vitdec` functions, use the `poly2trellis` function to convert those parameters into a trellis structure. The command to do this is below.

```
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Define trellis.
```

Using the Encoder

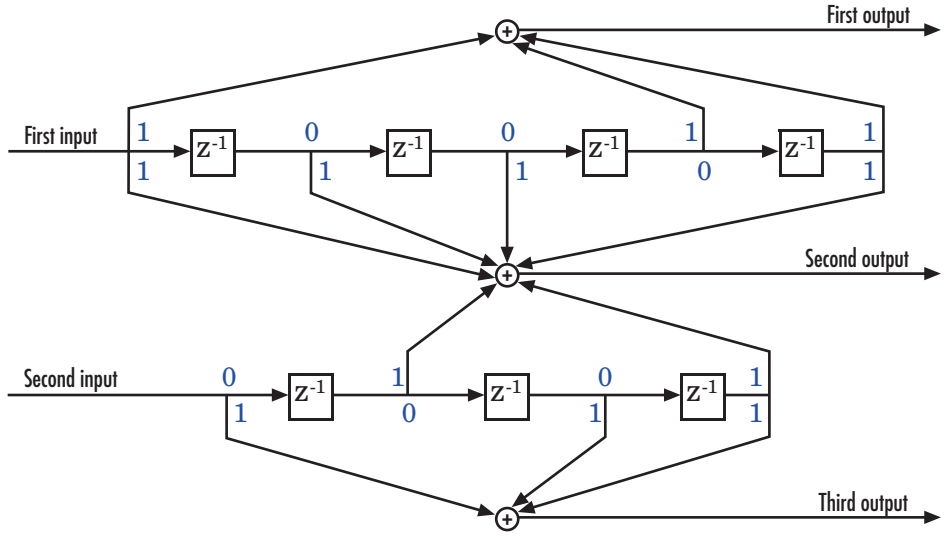
Below is a script that uses this encoder.

```
len = 1000;

msg = 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 0s, 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.

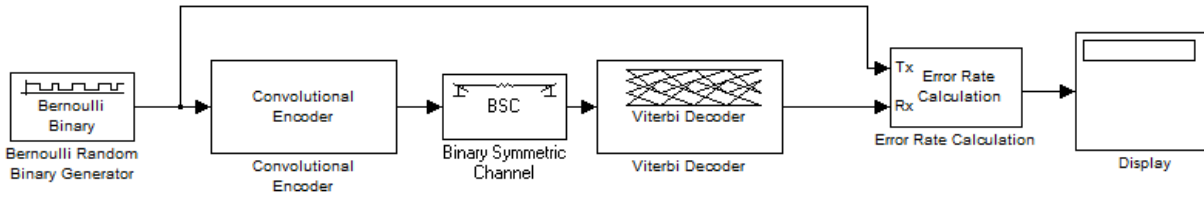
The encoder's constraint length is a vector of length 2 since the encoder has two inputs. The elements of this vector indicate the number of bits stored in each shift register, including the current input bits. Counting memory spaces in each shift register in the diagram and adding one for the current inputs leads to a constraint length of [5 4].

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, notice that the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13. The full value of the code generator matrix is [27 33 0; 0 5 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 `randseed` 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 35 0; 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 `randseed` 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 in the figure. From the model window's **Simulation** menu, select **Model Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `inf`.

Notes on the Model

The matrix size annotations appear on the connecting lines only if you click the **Display** menu and select **Signals & Ports > 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

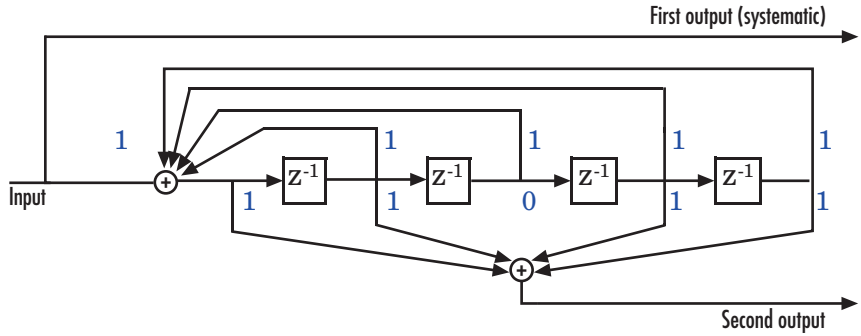
This example processes a punctured convolutional code. It begins by generating 30,000 random bits and encoding them using a rate-3/4 convolutional encoder with a puncture pattern of [1 1 1 0 0 1]. The resulting vector contains 40,000 bits, which are mapped to values of -1 and 1 for transmission. The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

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

```
len = 30000; msg = randi([0 1], len, 1); % Random data
t = poly2trellis(7, [133 171]); % Define trellis.
% 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 [1 1 1 1 1], 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 1 0 1 1], 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” in the Communications System Toolbox documentation.

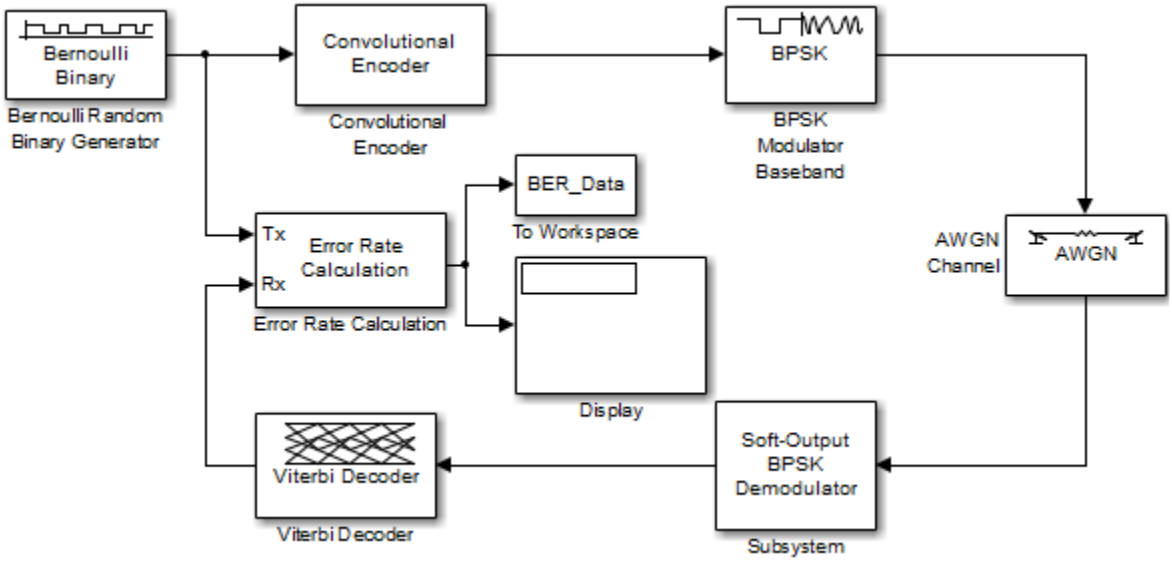
Soft-Decision Decoding

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 4-61

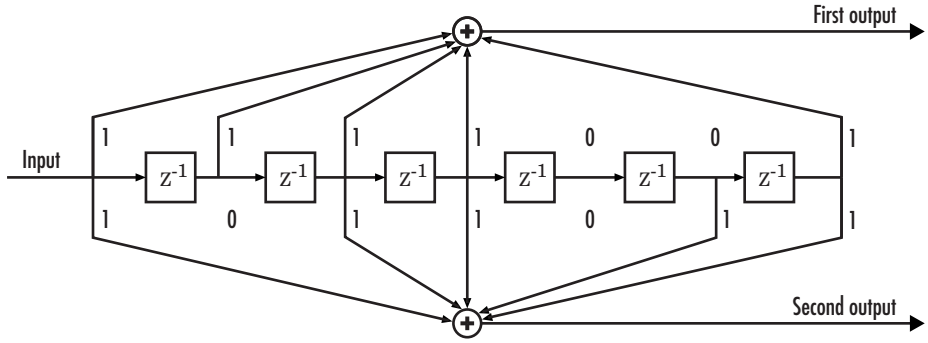
Overview of the Simulation

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*log10(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 Soft-Output 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^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^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

Decision Value	Interpretation
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** 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 Error Rate Calculation block's **Receive delay** parameter 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 (49).

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_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_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_d is the pairwise error probability, given by

$$P_d = \frac{1}{2}\text{erfc}\left(\sqrt{dR\frac{E_b}{N_0}}\right)$$

where R is the code rate of 1/2, and `erfc` is the MATLAB complementary error function, defined by

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$$

Values for the coefficients c_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_b for E_b/N_0 values in the range from 1 to 3.5, in increments of 0.5:

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.

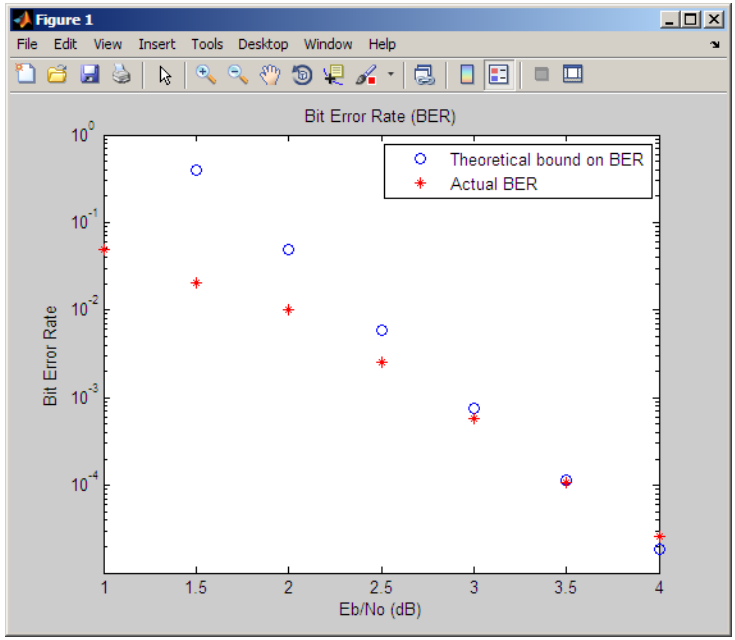
Note First open the model by clicking here in the MATLAB Help browser. Then execute these commands, which might take a few minutes.

```
% 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 1e-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_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.



Tailbiting Encoding Using Feedback Encoders

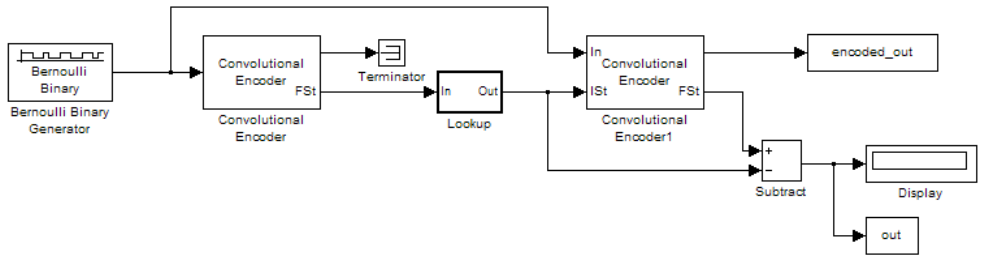
This example demonstrates Tailbiting encoding using feedback encoders. For feedback encoders, the ending state depends on the entire block of data. To accomplish tailbiting, you must calculate for a given information vector (of N bits), the initial state, that leads to the same ending state after the 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 $X_N^{[zs]}$. 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 $X_N^{[zs]}$ 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 (n-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

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:

To learn how to represent words for BCH or Reed-Solomon codes, see "Represent Words for BCH Codes" on page 4-88 or "Represent Words for Reed-Solomon Codes" on page 4-95.

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

     1     0     0     1     1

  Columns 6 through 10

     0     1     0     1     0

  Columns 11 through 12

     1     1


ans =

  Columns 1 through 5

     1     1     1     0     0

  Columns 6 through 10

     1     0     0     1     0

  Columns 11 through 15

     1     0     0     1     1

  Columns 16 through 18

     0     1     1
```

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

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 [n,k] cyclic, Hamming, and generic linear block codes. The table below lists the items and the coding techniques for which they are most relevant.

Parameters Used in Block Coding Techniques

Parameter	Block Coding Technique
“Generator Matrix” on page 4-73	Generic linear block
“Parity-Check Matrix” on page 4-73	Generic linear block
“Generator Polynomial” on page 4-75	Cyclic

Parameter	Block Coding Technique
"Decoding Table" on page 4-75	Generic linear block, Hamming

Generator Matrix

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

Some examples of generator matrices are in the next section, "Parity-Check Matrix" on page 4-73.

Parity-Check Matrix

Decoding an [n,k] linear block code requires an (n-k)-by-n parity-check matrix H. It satisfies $GH^{tr} = 0 \pmod 2$, where H^{tr} denotes the matrix transpose of H, G is the code's generator matrix, and this zero matrix is k-by-(n-k). If $G = [I_k\ P]$ then $H = [-P^{tr}\ I_{n-k}]$. Most functions in this 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 [n,k] binary linear block code.

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k\ P]$ or $[P\ I_k]$	k-by-n
Parity-check	$[-P'\ I_{n-k}]$ or $[I_{n-k}\ -P']$	(n-k)-by-n

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

Examples

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

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

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

genmat =

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

The next example finds parity-check and generator matrices for a [7,3] cyclic code. The `cyclpoly` function is mentioned below in "Generator Polynomial" on page 4-75.

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

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

genmat =

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

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

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

parmat =

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

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

Generator Polynomial

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

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

```
genpoly = cyclpoly(7,3)

genpoly =

     1     0     1     1     1
```

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

Decoding Table

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

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

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 = [1 0 0 1 1 1 1] % Suppose this is the received vector.
syndrome = rem(recd * parmat',2);
syndrome_de = bi2de(syndrome,'left-msb'); % Convert to decimal.
disp(['Syndrome = ',num2str(syndrome_de),...
      ' (decimal), ',num2str(syndrome),' (binary)'])
corrvect = trt(1+syndrome_de,:) % Correction vector
% Now compute the corrected codeword.
correctedcode = rem(corrvect+recd,2)
```

The output is below.

```
recd =

     1     0     0     1     1     1     1

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

corrvect =

     0     0     0     0     1     0     0


correctedcode =

     1     0     0     1     0     1     1
```

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 codes, cyclic codes, and Hamming codes.

Generic Linear Block Codes

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

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

encodes the information in `msg` using the [`n`,`k`] code that the generator matrix `genmat` determines. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary

representations. See “Represent Words for Linear Block Codes” on page 4-69 or the reference page for `encode` for a description of the formats of `msg` and `code`.

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

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

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

Example: Generic Linear Block Coding

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

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

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

```
err_words =

     1
     2
```

Cyclic Codes

A cyclic code is a linear block code with the property that cyclic shifts of a codeword (expressed as a series of bits) are also codewords. An alternative characterization of cyclic

codes is based on its generator polynomial, as mentioned in "Generator Polynomial" on page 4-75 and discussed in [5].

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

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

encodes the information in `msg` using the [`n`,`k`] code determined by the generator polynomial `genpoly`. `genpoly` is an optional argument for `encode`. The default generator polynomial is `cyclpoly(n,k)`. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See "Represent Words for Linear Block Codes" on page 4-69 or the reference page for `encode` for a description of the formats of `msg` and `code`.

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

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

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

Example

You can modify the example in the section "Generic Linear Block Codes" on page 4-76 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`.

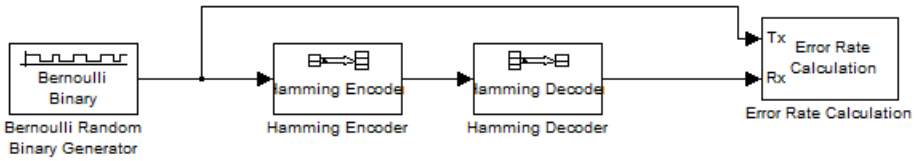
Hamming Codes

The reference pages for `encode` and `decode` contain examples of encoding and decoding Hamming codes. Also, the section "Decoding Table" on page 4-75 illustrates error correction in a Hamming code.

Hamming Codes

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 `randseed` function.
  - Check the **Frame-based outputs** check box.
  - Set **Samples per frame** to `4`.
- Hamming Encoder, with default parameter values

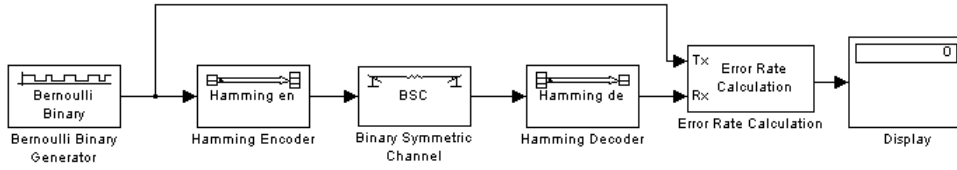
- Hamming Decoder, with default parameter values
- Error Rate Calculation, in the Comm Sinks library, with default parameter values

Connect the blocks as in the preceding figure. Click the **Display** menu and select **Signals & Ports > Signal Dimensions**. After updating the diagram if necessary (**Simulation > Update Diagram**), 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

Section Overview

This section describes how to reduce the error rate by adding an error-correcting code. The following figure shows an example that uses a Hamming code.



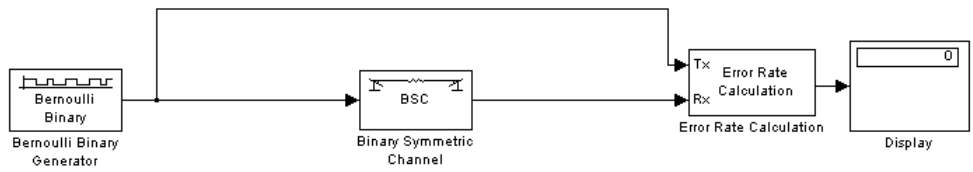
Hamming Code Model

To open a complete version of the model, type doc_hamming at the MATLAB prompt.

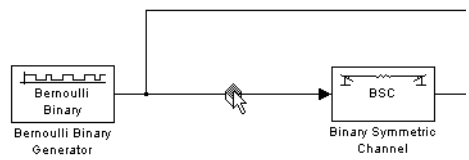
Building the Hamming Code Model

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. Drag the following blocks from the Simulink Library Browser into the model window:
   - Hamming Encoder block, from the Block sublibrary of the Error Detection and Correction library
   - Hamming Decoder block, from the Block sublibrary of the Error Detection and Correction library
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 block, the Error Rate Calculation block, and the Display block to the right by clicking and dragging. This creates more space between the Binary Symmetric Channel block and the blocks next to it. The model should now look like the following figure.



5. Click the Hamming Encoder block and drag it 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.



6 Click the Hamming Decoder block and drag it on top of the line between the Binary Symmetric Channel block and the Error Rate Calculation block.

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, 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.

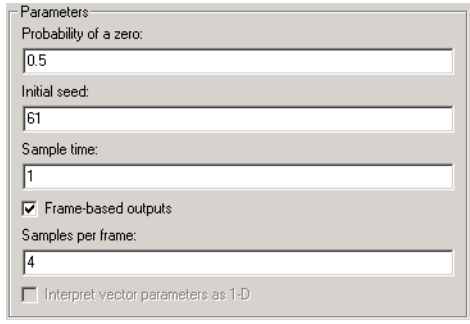
To learn more about the Communications System Toolbox block coding features, see "Block Codes"in the online documentation.

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 Select the box next to **Frame-based outputs**.

2 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 Frame-Based Processing" 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, select the box next to **Frame-based outputs** in the dialog for the source, and set **Samples per frame** to the required value.

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 > Start**. 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.

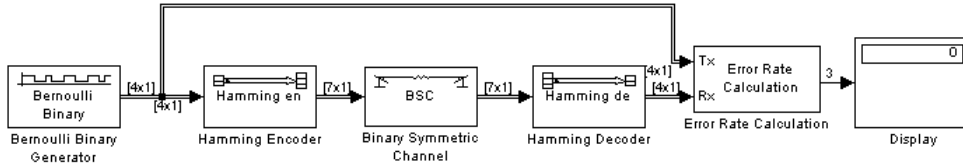
You expect an error rate of approximately .001 for the following reason: The probability of two or more errors occurring in a codeword of length 7 is
$1 - (0.99)^7 - 7(0.99)^6(0.01) = 0.002$

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 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

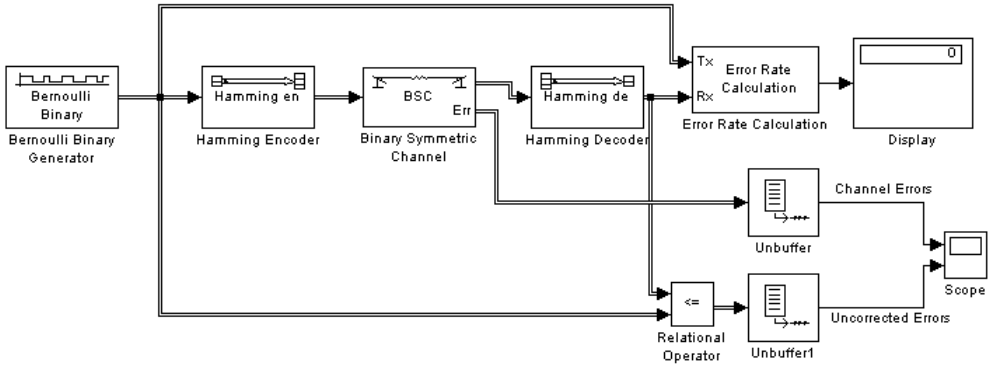
You can display the sizes of data frames in different parts of the model by clicking the **Display** menu and selecting **Signals & Ports > Signal Dimensions**. The line leading out of the Bernoulli Binary Generator block is labeled `[4x1]`, indicating that its output consists of column vectors of size 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]`.



Displaying Frame Sizes

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 Hamming Code Model, 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 and click the **Parameters** button on the toolbar. Set **Number of axes** to 2 and 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

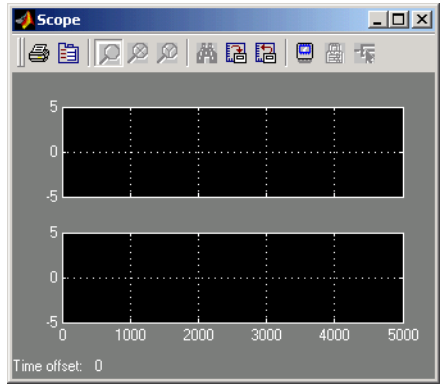
Double-click the Error Rate Calculation block and clear the box next to **Stop simulation** in the block's dialog box.

Scope Block

The Scope block displays the channel errors and uncorrected errors. To configure the block,

1. Double-click the block to open the scope, if it is not already open.
2. Click the **Parameters** button on the toolbar.
3. Set **Time range** to 5000.
4. Click the **Data history** tab.
5. Type 30000 in the **Limit data points to last** field, and click **OK**.

The scope should now appear as shown.



To configure the axes, follow these steps:

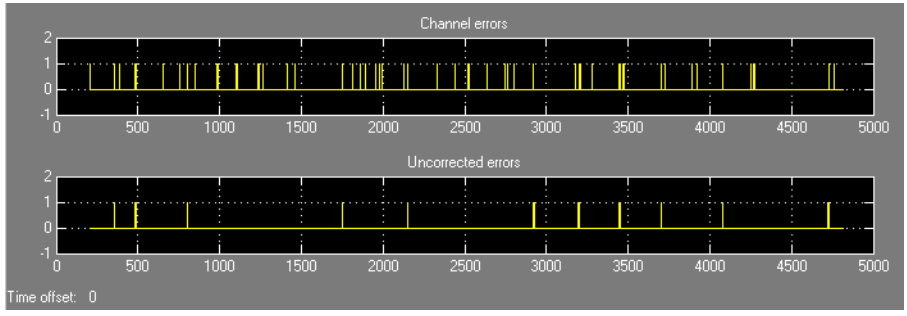
1. Right-click the vertical axis at the left side of the upper scope.
2. In the context menu, select **Axes properties**.
3. In the **Y-min** field, type `-1`.
4. In the **Y-max** field, type `2`, and click **OK**.
5. Repeat the same steps for the vertical axis of the lower scope.
6. Widen the scope 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 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 block displays the error data. At the end of each 5000 time steps, the scope appears as shown in the following figure. The scope then clears the displayed data and displays the next 5000 data points.



Scope with Model Running

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.

To zoom in on the scope so that you can see individual errors, 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.

Zooming In on the Scope

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" explains how to send the error data to the MATLAB workspace for more detailed analysis.

BCH Codes

- “Algorithms for BCH Errors-only Decoding” on page 4-91

Represent Words for BCH Codes

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

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

```
h = comm.BCHEncoder
msg = [1 0 0 1 0; 1 0 1 1 1]; % Messages in a Galois array
obj = comm.BCHEncoder;
c1 = step(obj, msg(1,:)');
c2 = step(obj, msg(2,:)');
cbch = [c1 c2].'
```

The output is

```
  Columns 1 through 5

     1     0     0     1     0
     1     0     1     1     1

  Columns 6 through 10

     0     0     1     1     1
     0     0     0     0     1

  Columns 11 through 15

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

Parameters for BCH Codes

BCH codes use special values of n and k:

- n, the codeword length, is an integer of the form 2^m-1 for some integer m > 2.
- k, the message length, is a positive integer less than n. However, only some positive integers less than n are valid choices for k. See the 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 4-88 and "Parameters for BCH Codes" on page 4-88.

The topics are

- "Example: BCH Coding Syntaxes" on page 4-89
- "Detect and Correct Errors in a BCH Code Using MATLAB" on page 4-90

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
hEnc = comm.BCHEncoder('CodewordLength', n, 'MessageLength', k);
hDec = comm.BCHDecoder('CodewordLength', n, 'MessageLength', k);
c1 = step(hEnc,msg); % BCH encoding
d1 = step(hDec,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(c11,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(hDec,c1_append);
```

```
chk = 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
hEnc = comm.BCHEncoder('CodewordLength', n, 'MessageLength', k, ...
    'GeneratorPolynomialSource', 'Property', 'GeneratorPolynomial', gp);
hDec = comm.BCHDecoder('CodewordLength', n, 'MessageLength', k, ...
    'GeneratorPolynomialSource', 'Property', 'GeneratorPolynomial', gp);
c = step(hEnc, 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(hDec, 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 `c + noise` takes place in the Galois field GF(2) because `c` is a Galois array in GF(2).

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

```
chk2 =
```

```
     1


nerrs =

     3
     3
     3
     3
```

Excessive Noise in BCH Codewords

In the previous example, 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 4-100.

Algorithms for BCH 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 $2t$ terms of the infinite degree syndrome polynomial, $S(z)$.
2. If those $2t$ terms of $S(z)$ are all equal to 0, then the code has no errors , no correction needs to be performed, and the decoding algorithm ends.
3. If one or more terms of $S(z)$ are nonzero, calculate the error locator polynomial, $\Lambda(z)$, via the Berlekamp algorithm.
4. Calculate the error evaluator polynomial, $\Omega(z)$, via

   $$\Lambda(z)S(z) = \Omega(z) \bmod z^{2t}$$

5. Correct an error in the codeword according to

   $$e_{i_m} = \frac{\Omega(\alpha^{-i_m})}{\Lambda'(\alpha^{-i_m})}$$

where e_{i_m} is the error magnitude in the i_m th position in the codeword, m is a value less than the error-correcting capability of the code, $\Omega(z)$ is the error magnitude polynomial, $\Lambda'(z)$ is the formal derivative [5] of the error locator polynomial, $\Lambda(z)$, and α is the primitive element of the Galois field of the code.

Further description of several of the steps is given in the following sections.

Syndrome Calculation

For narrow-sense codes, the $2t$ terms of $S(z)$ are calculated by evaluating the received codeword at successive powers of α (the field's primitive element) from 0 to $2t$-1. In other words, if we assume one-based indexing of codewords $C(z)$ and the syndrome polynomial $S(z)$, and that codewords are of the form $[c_1\ c_1\ \ldots\ c_N]$, 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

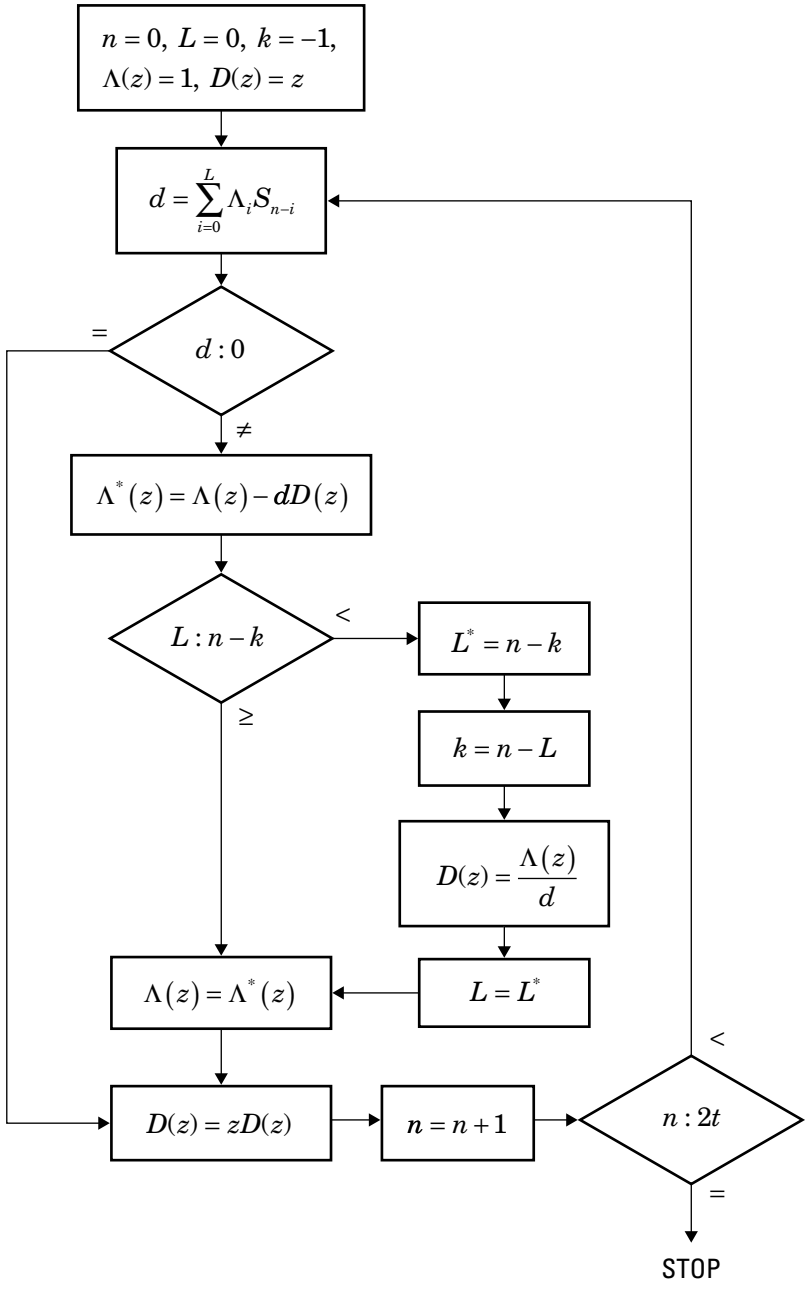
The error locator polynomial, $\Lambda(z)$, is found using the Berlekamp algorithm. A complete description of this algorithm is found in [2], but we summarize the algorithm as follows.

We define the following variables.

Variable	Description
n	Iterator variable
k	Iterator variable
L	Length of the feedback register used to generate the first $2t$ terms of $S(z)$
$D(z)$	Correction polynomial

Variable	Description
d	Discrepancy

The following diagram shows the iterative procedure (i.e., the Berlekamp algorithm) used to find $\Lambda(z)$.


$n = 0,\ L = 0,\ k = -1,$
$\Lambda(z) = 1,\ D(z) = z$
$d = \sum_{i=0}^{L} \Lambda_i S_{n-i}$
$d : 0$
=
≠
$\Lambda^*(z) = \Lambda(z) - dD(z)$
$L : n - k$
<
≥
$L^* = n - k$
$k = n - L$
$D(z) = \frac{\Lambda(z)}{d}$
$L = L^*$
$\Lambda(z) = \Lambda^*(z)$
$D(z) = zD(z)$
$n = n + 1$
$n : 2t$
<
=
STOP

Error Evaluator Polynomial Calculation

The error evaluator polynomial, $\Omega(z)$, is simply the convolution of $\Lambda(z)$ and $S(z)$.

Reed-Solomon Codes

Represent Words for Reed-Solomon Codes

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

Note: For information about Galois arrays and how to create them, see "Representing Elements of Galois Fields" on page 4-108 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);
c1 = 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.

Allowable Values of Integer Parameters

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

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

Generator Polynomial

The `rsgenpoly` function produces generator polynomials for Reed-Solomon codes. `rsgenpoly` represents a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable. If each symbol has m bits, the Galois row vector is in the field GF(2^m). For example, the command

```
r = rsgenpoly(15,13)

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

Array elements =

     1     6     8
```

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

Algebraic Expression for Generator Polynomials

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

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

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 4-95 and "Parameters for Reed-Solomon Codes" on page 4-96.

This section illustrates how to use the `RS Encoder` and `RS Decoder` System objects. The topics are

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 `gf`.
- Vary the position of the parity symbols within the codewords, choosing either the end (default) or beginning.

This example also shows that corresponding syntaxes of 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);
c1 = step(hEnc, msg);
d1 = step(hDec, c1);


% 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    = [1 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);
chk = isequal(msg,d3_append)
```

The output is

```
chk =

     1
```

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^m`, and the addition operation `c + noise` takes place in the Galois field GF(`2^m`) because `c` is a Galois array in GF(`2^m`).

The output from the example is below. The nonzero value of `ans` indicates that the decoder was able to correct the corrupted codewords and recover the original message.

The values in the vector `nerrs` indicates that the decoder corrected `t` errors in each codeword.

```
ans =

     1

nerrs =

     2
     2
     2
     2
```

Excessive Noise in Reed-Solomon Codewords

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

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

In both cases, the decoder returns the wrong message. However, you can tell when a decoding failure occurs because `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^m-1. A shortened [n,k] Reed-Solomon code implicitly uses an [n_1,k_1] encoder, where

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

The 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,[1 1 1 1 1]');
hEnc = comm.RSEncoder(5,3);
shortenedcode = step(hEnc,[1 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
hEnc = comm.RSEncoder(n_pad,k_pad);
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+x^4+x^2+x+1
genpolyRS = rsgenpoly(15,5)
```

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

```
genpolyCyclic =

     1     0     0     0     0     1     0     0     0     0     1


genpolyBCH = GF(2) array.

Array elements =

     1     0     1     0     0     1     1     0     1     1     1


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

Array elements =

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

The formats of these outputs vary:

- `cyclpoly` represents a generator polynomial using an integer row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable.
- `bchgenpoly` and `rsgenpoly` represent a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable.
- `rsgenpoly` uses coefficients in a Galois field other than the binary field GF(2). For more information on the meaning of these coefficients, see "How Integers Correspond to Galois Field Elements" on page 4-110 and "Polynomials over Galois Fields" on page 4-129.

Nonuniqueness of Generator Polynomials

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

Algebraic Expression for Generator Polynomials

The generator polynomials produced by `bchgenpoly` and `rsgenpoly` have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where A is a primitive element for an appropriate Galois

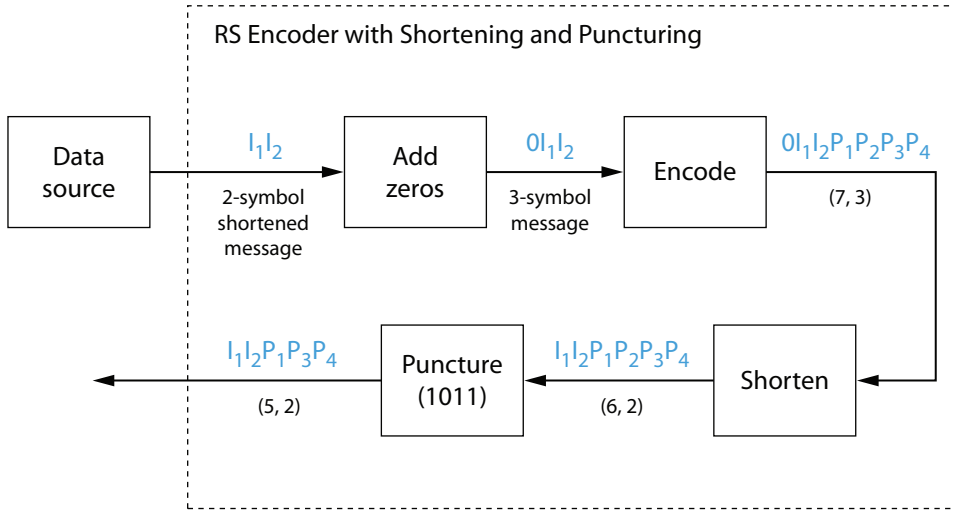
field, and b and t are integers. See the functions' reference pages for more information about this expression.

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.



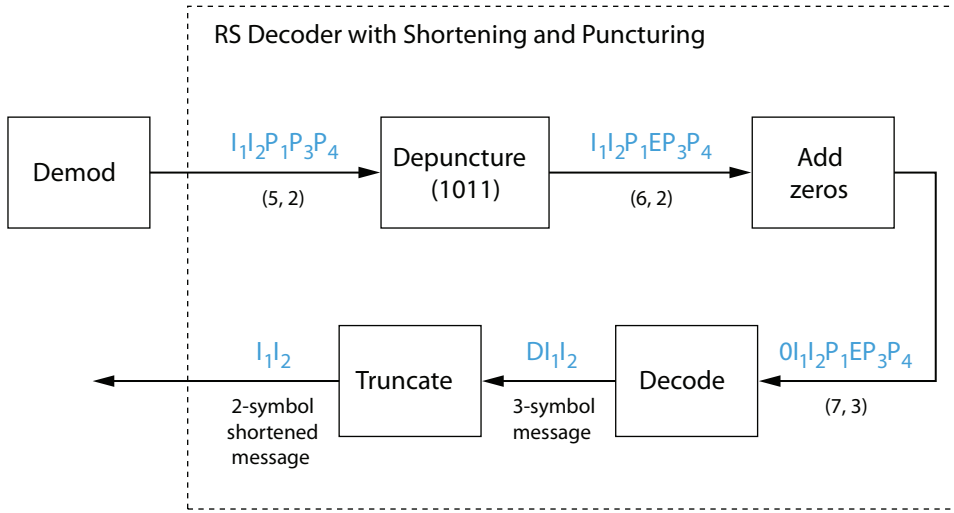
In this figure, the message source outputs two information symbols, designated by I_1I_2. (For a BCH example, the symbols are simply binary bits.) Because the code is a shortened (7,3) code, a zero must be added ahead of the information symbols, yielding a three-symbol message of $0I_1I_2$. The modified message sequence is then RS encoded, and the added information zero is subsequently removed, which yields a result of $I_1I_2P_1P_2P_3P_4$. (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 $I_1I_2P_1P_3P_4$.

Decoder Example with Shortening and Puncturing

The following figure shows how the RS encoder operates on a shortened and punctured codeword.

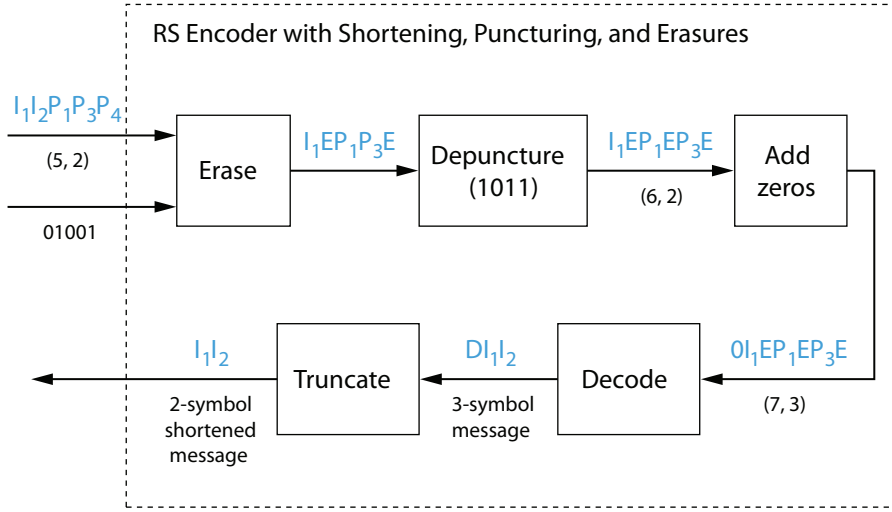


This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a (5,2) codeword, because it has been shortened from a (7,3) codeword by one symbol, and one symbol has also been punctured.

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 erasure-correcting capability of the code, which can correct four erasures. The decoding operation results in the three-symbol message DI_1I_2. The first symbol is truncated, as in the preceding figure, yielding a final output of I_1I_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 $I_1I_2P_1P_3P_4$ 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 $I_1EP_1P_3E$, 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 $I_1EP_1EP_3E$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0I_1EP_1EP_3E$, such that a (7,3) codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of DI_1I_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_1I_2 vector.

For additional information, see the "Reed-Solomon Coding with Erasures, Punctures, and Shortening" example.

LDPC Codes

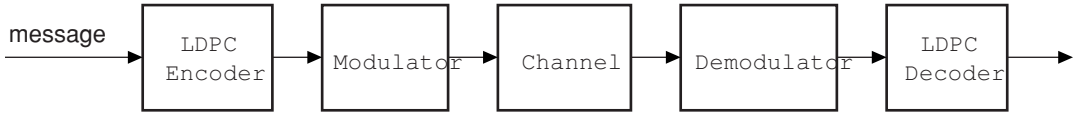
Low-Density Parity-Check (LDPC) codes are linear error control codes with:

- Sparse parity-check matrices
- Long block lengths that can attain performance near the Shannon limit (see LDPC Encoder and LDPC Decoder)

Communications System 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^m members are used in error-control coding and are denoted GF(2^m). This

chapter describes how to work with fields that have 2^m members, where m is an integer between 1 and 16. The sections in this chapter are as follows.

If you need to use Galois fields having an odd number of elements, see "Galois Fields of Odd Characteristic".

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 System 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 GF(2^m) is a cyclic generator of the group of nonzero elements of GF(2^m). This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power.

- A *primitive polynomial* for GF(2^m) is the minimal polynomial of some primitive element of GF(2^m). It is the binary-coefficient polynomial of smallest nonzero degree having a certain primitive element as a root in GF(2^m). As a consequence, a primitive polynomial has degree m and is irreducible.

The definitions imply that a primitive element is a root of a corresponding primitive polynomial.

Representing Elements of Galois Fields

Section Overview

This section describes how to create a *Galois array*, which is a MATLAB expression that represents the elements of a Galois field. This section also describes how MATLAB technical computing software interprets the numbers that you use in the representation, and includes several examples.

Creating a Galois Array

To begin working with data from a Galois field GF(`2^m`), you must set the context by associating the data with crucial information about the field. The `gf` function performs this association and creates a Galois array in MATLAB. This function accepts as inputs

- The Galois field data, `x`, which is a MATLAB array whose elements are integers between 0 and `2^m-1`.
- (*Optional*) An integer, `m`, that indicates `x` is in the field GF(`2^m`). Valid values of `m` are between 1 and 16. The default is 1, which means that the field is GF(2).
- (*Optional*) A positive integer that indicates which primitive polynomial for GF(`2^m`) you are using in the representations in `x`. If you omit this input argument, `gf` uses a default primitive polynomial for GF(`2^m`). For information about this argument, see "Specifying the Primitive Polynomial" on page 4-112.

The output of the `gf` function is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate the variable, MATLAB works within the Galois field you have specified. For example, if you apply the `log` function to a Galois array, MATLAB computes the logarithm in the Galois field and *not* in the field of real or complex numbers.

When MATLAB Implicitly Creates a Galois Array

Some operations on Galois arrays require multiple arguments. If you specify one argument that is a Galois array and another that is an ordinary MATLAB array, MATLAB interprets both as Galois arrays in the same field. It implicitly invokes the `gf` function on the ordinary MATLAB array. This implicit invocation simplifies your syntax because you can omit some references to the `gf` function. For an example of the simplification, see "Example: Addition and Subtraction" on page 4-115.

Example: Creating Galois Field Variables

The code below creates a row vector whose entries are in the field GF(4), and then adds the row to itself.

```
x = 0:3; % A row vector containing integers
m = 2; % Work in the field GF(2^2), or, GF(4).
a = gf(x,m) % Create a Galois array in GF(2^m).

b = a + a % Add a to itself, creating b.
```

The output is

```
a = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     1     2     3


b = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     0     0     0
```

The output shows the values of the Galois arrays named `a` and `b`. Each output section indicates

- The field containing the variable, namely, GF(2^2) = GF(4).

- The primitive polynomial for the field. In this case, it is the toolbox's default primitive polynomial for GF(4).
- The array of Galois field values that the variable contains. In particular, the array elements in `a` are exactly the elements of the vector `x`, and the array elements in `b` are four instances of the zero element in GF(4).

The command that creates `b` shows how, having defined the variable `a` as a Galois array, you can add `a` to itself by using the ordinary `+` operator. MATLAB performs the vectorized addition operation in the field GF(4). The output shows that

- Compared to `a`, `b` is in the same field and uses the same primitive polynomial. It is not necessary to indicate the field when defining the sum, `b`, because MATLAB remembers that information from the definition of the addends, `a`.
- The array elements of `b` are zeros because the sum of any value with itself, in a Galois field *of characteristic two*, is zero. This result differs from the sum `x + x`, which represents an addition operation in the infinite field of integers.

Example: Representing Elements of GF(8)

To illustrate what the array elements in a Galois array mean, the table below lists the elements of the field GF(8) as integers and as polynomials in a primitive element, A. The table should help you interpret a Galois array like

```
gf8 = gf([0:7],3); % Galois vector in GF(2^3)
```

Integer Representation	Binary Representation	Element of GF(8)
0	000	0
1	001	1
2	010	A
3	011	A + 1
4	100	A^2
5	101	$A^2 + 1$
6	110	$A^2 + A$
7	111	$A^2 + A + 1$

How Integers Correspond to Galois Field Elements

Building on the GF(8) example above, this section explains the interpretation of array elements in a Galois array in greater generality. The field GF(2^m) has 2^m distinct elements, which this toolbox labels as 0, 1, 2,..., 2^m-1. These integer labels correspond to elements of the Galois field via a polynomial expression involving a primitive element of the field. More specifically, each integer between 0 and 2^m-1 has a binary representation in m bits. Using the bits in the binary representation as coefficients in a polynomial, where the least significant bit is the constant term, leads to a binary polynomial whose order is at most m-1. Evaluating the binary polynomial at a primitive element of GF(2^m) leads to an element of the field.

Conversely, any element of GF(2^m) can be expressed as a binary polynomial of order at most m-1, evaluated at a primitive element of the field. The m-tuple of coefficients of the polynomial corresponds to the binary representation of an integer between 0 and 2^m.

Below is a symbolic illustration of the correspondence of an integer X, representable in binary form, with a Galois field element. Each b_k is either zero or one, while A is a primitive element.

$$X = b_{m-1} \cdot 2^{m-1} + \cdots + b_2 \cdot 4 + b_1 \cdot 2 + b_0$$
$$\leftrightarrow b_{m-1} \cdot A^{m-1} + \cdots + b_2 \cdot A^2 + b_1 \cdot A + b_0$$

Example: Representing a Primitive Element

The code below defines a variable alph that represents a primitive element of the field GF(2^4).

```
m = 4; % Or choose any positive integer value of m.
alph = gf(2,m) % Primitive element in GF(2^m)
```

The output is

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

Array elements =

     2
```

The Galois array alph represents a primitive element because of the correspondence among

- The integer 2, specified in the gf syntax
- The binary representation of 2, which is 10 (or 0010 using four bits)

- The polynomial A + 0, where A is a primitive element in this field (or $0A^3 + 0A^2 + A + 0$ using the four lowest powers of A)

Primitive Polynomials and Element Representations

This section builds on the discussion in "Creating a Galois Array" on page 4-108 by describing how to specify your own primitive polynomial when you create a Galois array. The topics are

If you perform many computations using a nondefault primitive polynomial, see "Speed and Nondefault Primitive Polynomials" on page 4-135.

Specifying the Primitive Polynomial

The discussion in "How Integers Correspond to Galois Field Elements" on page 4-110 refers to a primitive element, which is a root of a primitive polynomial of the field. When you use the `gf` function to create a Galois array, the function interprets the integers in the array with respect to a specific default primitive polynomial for that field, unless you explicitly provide a different primitive polynomial. A list of the default primitive polynomials is on the reference page for the `gf` function.

To specify your own primitive polynomial when creating a Galois array, use a syntax like

```
c = gf(5,4,25) % 25 indicates the primitive polynomial for GF(16).
```

instead of

```
c1= gf(5,4); % Use default primitive polynomial for GF(16).
```

The extra input argument, `25` in this case, specifies the primitive polynomial for the field GF(`2^m`) in a way similar to the representation described in "How Integers Correspond to Galois Field Elements" on page 4-110. In this case, the integer 25 corresponds to a binary representation of 11001, which in turn corresponds to the polynomial $D^4 + D^3 + 1$.

Note: When you specify the primitive polynomial, the input argument must have a binary representation using exactly `m+1` bits, not including unnecessary leading zeros. In other words, a primitive polynomial for GF(`2^m`) always has order `m`.

When you use an input argument to specify the primitive polynomial, the output reflects your choice by showing the integer value as well as the polynomial representation.

```
d = gf([1 2 3],4,25)
```

```
d = GF(2^4) array. Primitive polynomial = D^4+D^3+1 (25 decimal)

Array elements =

     1     2     3
```

Note: After you have defined a Galois array, you cannot change the primitive polynomial with respect to which MATLAB interprets the array elements.

Finding Primitive Polynomials

You can use the `primpoly` function to find primitive polynomials for GF(`2^m`) and the `isprimitive` function to determine whether a polynomial is primitive for GF(`2^m`). The code below illustrates.

```
m = 4;
defaultprimpoly = primpoly(m) % Default primitive poly for GF(16)
allprimpolys = primpoly(m,'all') % All primitive polys for GF(16)
i1 = isprimitive(25) % Can 25 be the prim_poly input in gf(...)?
i2 = isprimitive(21) % Can 21 be the prim_poly input in gf(...)?
```

The output is below.

```
Primitive polynomial(s) =

D^4+D^1+1

defaultprimpoly =

    19

Primitive polynomial(s) =

D^4+D^1+1
D^4+D^3+1

allprimpolys =

    19
    25

i1 =
```

```
     1

i2 =

     0
```

Effect of Nondefault Primitive Polynomials on Numerical Results

Most fields offer multiple choices for the primitive polynomial that helps define the representation of members of the field. When you use the `gf` function, changing the primitive polynomial changes the interpretation of the array elements and, in turn, changes the results of some subsequent operations on the Galois array. For example, exponentiation of a primitive element makes it easy to see how the primitive polynomial affects the representations of field elements.

```
a11 = gf(2,3); % Use default primitive polynomial of 11.
a13 = gf(2,3,13); % Use D^3+D^2+1 as the primitive polynomial.
z = a13.^3 + a13.^2 + 1 % 0 because a13 satisfies the equation
nz = a11.^3 + a11.^2 + 1 % Nonzero. a11 does not satisfy equation.
```

The output below shows that when the primitive polynomial has integer representation `13`, the Galois array satisfies a certain equation. By contrast, when the primitive polynomial has integer representation `11`, the Galois array fails to satisfy the equation.

```
z = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     0


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

Array elements =

     6
```

The output when you try this example might also include a warning about lookup tables. This is normal if you did not use the `gftable` function to optimize computations involving a nondefault primitive polynomial of 13.

Arithmetic in Galois Fields

- "Section Overview" on page 4-115

Section Overview

You can perform arithmetic operations on Galois arrays by using familiar MATLAB operators, listed in the table below. Whenever you operate on a pair of Galois arrays, both arrays must be in the same Galois field.

Operation	Operator
Addition	`+`
Subtraction	`-`
Elementwise multiplication	`.*`
Matrix multiplication	`*`
Elementwise left division	`./`
Elementwise right division	`.\`
Matrix left division	`/`
Matrix right division	`\`
Elementwise exponentiation	`.^`
Elementwise logarithm	`log()`
Exponentiation of a square Galois matrix by a scalar integer	`^`

For multiplication and division of polynomials over a Galois field, see "Addition and Subtraction of Polynomials" on page 4-129.

Example: Addition and Subtraction

The code below adds two Galois arrays to create an addition table for GF(8). Addition uses the ordinary + operator. The code below also shows how to index into the array `addtb` to find the result of adding 1 to the elements of GF(8).

```
m = 3;
```

```
e = repmat([0:2^m-1],2^m,1);
f = gf(e,m); % Create a Galois array.
addtb = f + f' % Add f to its own matrix transpose.

addone = addtb(2,:); % Assign 2nd row to the Galois vector addone.
```

The output is below.

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

Array elements =

     0     1     2     3     4     5     6     7
     1     0     3     2     5     4     7     6
     2     3     0     1     6     7     4     5
     3     2     1     0     7     6     5     4
     4     5     6     7     0     1     2     3
     5     4     7     6     1     0     3     2
     6     7     4     5     2     3     0     1
     7     6     5     4     3     2     1     0
```

As an example of reading this addition table, the (7,4) entry in the `addtb` array shows that `gf(6,3)` plus `gf(3,3)` equals `gf(5,3)`. Equivalently, the element A^2+A plus the element A+1 equals the element A^2+1. The equivalence arises from the binary representation of 6 as 110, 3 as 011, and 5 as 101.

The subtraction table, which you can obtain by replacing `+` by `-`, is the same as `addtb`. This is because subtraction and addition are identical operations in a field *of characteristic two*. In fact, the zeros along the main diagonal of `addtb` illustrate this fact for GF(8).

Simplifying the Syntax

The code below illustrates scalar expansion and the implicit creation of a Galois array from an ordinary MATLAB array. The Galois arrays `h` and `h1` are identical, but the creation of `h` uses a simpler syntax.

```
g = gf(ones(2,3),4); % Create a Galois array explicitly.
h = g + 5; % Add gf(5,4) to each element of g.
h1 = g + gf(5*ones(2,3),4) % Same as h.
```

The output is below.

```
h1 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)
```

```
Array elements =

     4     4     4
     4     4     4
```

Notice that 1+5 is reported as 4 in the Galois field. This is true because the 5 represents the polynomial expression A^2+1, and $1+(A^2+1)$ in GF(16) is A^2. Furthermore, the integer that represents the polynomial expression A^2 is 4.

Example: Multiplication

The example below multiplies individual elements in a Galois array using the `.*` operator. It then performs matrix multiplication using the `*` operator. The elementwise multiplication produces an array whose size matches that of the inputs. By contrast, the matrix multiplication produces a Galois scalar because it is the matrix product of a row vector with a column vector.

```
m = 5;
row1 = gf([1:2:9],m); row2 = gf([2:2:10],m);
col = row2'; % Transpose to create a column array.
ep = row1 .* row2; % Elementwise product.
mp = row1 * col; % Matrix product.
```

Multiplication Table for GF(8)

As another example, the code below multiplies two Galois vectors using matrix multiplication. The result is a multiplication table for GF(8).

```
m = 3;
els = gf([0:2^m-1]',m);
multb = els * els' % Multiply els by its own matrix transpose.
```

The output is below.

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

Array elements =

     0     0     0     0     0     0     0     0
     0     1     2     3     4     5     6     7
     0     2     4     6     3     1     7     5
     0     3     6     5     7     4     1     2
     0     4     3     7     6     2     5     1
     0     5     1     4     2     7     3     6
```

```
     0     6     7     1     5     3     2     4
     0     7     5     2     1     6     4     3
```

Example: Division

The examples below illustrate the four division operators in a Galois field by computing multiplicative inverses of individual elements and of an array. You can also compute inverses using `inv` or using exponentiation by -1.

Elementwise Division

This example divides 1 by each of the individual elements in a Galois array using the `./` and `.\` operators. These two operators differ only in their sequence of input arguments. Each quotient vector lists the multiplicative inverses of the nonzero elements of the field. In this example, MATLAB expands the scalar 1 to the size of `nz` before computing; alternatively, you can use as arguments two arrays of the same size.

```
m = 5;
nz = gf([1:2^m-1],m); % Nonzero elements of the field
inv1 = 1 ./ nz; % Divide 1 by each element.
inv2 = nz .\ 1; % Obtain same result using .\ operator.
```

Matrix Division

This example divides the identity array by the square Galois array `mat` using the `/` and `\` operators. Each quotient matrix is the multiplicative inverse of `mat`. Notice how the transpose operator (`'`) appears in the equivalent operation using `\`. For square matrices, the sequence of transpose operations is unnecessary, but for nonsquare matrices, it is necessary.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minv1 = eye(3) / mat; % Compute matrix inverse.
minv2 = (mat' \ eye(3)')'; % Obtain same result using \ operator.
```

Example: Exponentiation

The examples below illustrate how to compute integer powers of a Galois array. To perform matrix exponentiation on a Galois array, you must use a square Galois array as the base and an ordinary (not Galois) integer scalar as the exponent.

Elementwise Exponentiation

This example computes powers of a primitive element, A, of a Galois field. It then uses these separately computed powers to evaluate the default primitive polynomial at A.

The answer of zero shows that A is a root of the primitive polynomial. The `.^` operator exponentiates each array element independently.

```
m = 3;
av = gf(2*ones(1,m+1),m); % Row containing primitive element
expa = av .^ [0:m]; % Raise element to different powers.
evp = expa(4)+expa(2)+expa(1) % Evaluate D^3 + D + 1.
```

The output is below.

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

Array elements =

     0
```

Matrix Exponentiation

This example computes the inverse of a square matrix by raising the matrix to the power -1. It also raises the square matrix to the powers 2 and -2.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minvs = mat ^ (-1); % Matrix inverse
matsq = mat^2; % Same as mat * mat
matinvssq = mat^(-2); % Same as minvs * minvs
```

Example: Elementwise Logarithm

The code below computes the logarithm of the elements of a Galois array. The output indicates how to express each *nonzero* element of GF(8) as a power of the primitive element. The logarithm of the zero element of the field is undefined.

```
gf8_nonzero = gf([1:7],3); % Vector of nonzero elements of GF(8)
expformat = log(gf8_nonzero) % Logarithm of each element
```

The output is

```
expformat =

     0     1     3     2     6     4     5
```

As an example of how to interpret the output, consider the last entry in each vector in this example. You can infer that the element `gf(7,3)` in GF(8) can be expressed as either

- A^5, using the last element of `expformat`
- A^2+A+1, using the binary representation of 7 as 111. See "Example: Representing Elements of GF(8)" on page 4-110 for more details.

Logical Operations in Galois Fields

Section Overview

You can apply logical tests to Galois arrays and obtain a logical array. Some important types of tests are testing for the equality of two Galois arrays and testing for nonzero values in a Galois array.

Testing for Equality

To compare corresponding elements of two Galois arrays that have the same size, use the operators == and ~=. The result is a logical array, each element of which indicates the truth or falsity of the corresponding elementwise comparison. If you use the same operators to compare a scalar with a Galois array, MATLAB technical computing software compares the scalar with each element of the array, producing a logical array of the same size.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg1 = (r1 .* r2 == [1 1 1]) % Does each element equal one?
lg2 = (r1 .* r2 == 1) % Same as above, using scalar expansion
lg3 = (r1 ~= r2) % Does each element differ from its inverse?
```

The output is below.

```
lg1 =

     1     1     1


lg2 =

     1     1     1


lg3 =
```

```
     0     1     1
```

Comparison of isequal and ==

To compare entire arrays and obtain a logical *scalar* result rather than a logical array, use the built-in `isequal` function. However, `isequal` uses strict rules for its comparison, and returns a value of `0` (false) if you compare

- A Galois array with an ordinary MATLAB array, even if the values of the underlying array elements match
- A scalar with a nonscalar array, even if all elements in the array match the scalar

The example below illustrates this difference between `==` and `isequal`.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg4 = isequal(r1 .* r2, [1 1 1]); % False
lg5 = isequal(r1 .* r2, gf(1,m)); % False
lg6 = isequal(r1 .* r2, gf([1 1 1],m)); % True
```

Testing for Nonzero Values

To test for nonzero values in a Galois vector, or in the columns of a Galois array that has more than one row, use the `any` or `all` function. These two functions behave just like the ordinary MATLAB functions `any` and `all`, except that they consider only the underlying array elements while ignoring information about which Galois field the elements are in. Examples are below.

```
m = 3; randels = gf(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 Arrays

Basic array operations on Galois arrays are in the table below. The functionality of these operations is analogous to the MATLAB operations having the same syntax.

Operation	Syntax
Index into array, possibly using colon operator instead of a vector of explicit indices	`a(vector)` or `a(vector,vector1)`, where `vector` and/or `vector1` can be `":"` instead of a vector
Transpose array	`a'`
Concatenate matrices	`[a,b]` or `[a;b]`
Create array having specified diagonal elements	`diag(vector)` or `diag(vector,k)`
Extract diagonal elements	`diag(a)` or `diag(a,k)`
Extract lower triangular part	`tril(a)` or `tril(a,k)`
Extract upper triangular part	`triu(a)` or `triu(a,k)`
Change shape of array	`reshape(a,k1,k2)`

The code below uses some of these syntaxes.

```
m = 4; a = gf([0:15],m);
a(1:2) = [13 13]; % Replace some elements of the vector a.
b = reshape(a,2,8); % Create 2-by-8 matrix.
c = [b([1 1 2],1:3); a(4:6)]; % Create 4-by-3 matrix.
d = [c, a(1:4)']; % Create 4-by-4 matrix.
dvec = diag(d); % Extract main diagonal of d.
dmat = diag(a(5:9)); % Create 5-by-5 diagonal matrix
dtril = tril(d); % Extract upper and lower triangular
dtriu = triu(d); % parts of d.
```

Basic Information About Galois Arrays

You can determine the length of a Galois vector or the size of any Galois array using the `length` and `size` functions. The functionality for Galois arrays is analogous to that of the MATLAB operations on ordinary arrays, except that the output arguments from

`size` and `length` are always integers, not Galois arrays. The code below illustrates the use of these functions.

```
m = 4; e = gf([0:5],m); f = reshape(e,2,3);
lne = length(e); % Vector length of e
szf = size(f); % Size of f, returned as a two-element row
[nr,nc] = size(f); % Size of f, returned as two scalars
nc2 = size(f,2); % Another way to compute number of columns
```

Positions of Nonzero Elements

Another type of information you might want to determine from a Galois array are the positions of nonzero elements. For an ordinary MATLAB array, you might use the `find` function. However, for a Galois array, you should use `find` in conjunction with the `~=` operator, as illustrated.

```
x = [0 1 2 1 0 2]; m = 2; g = gf(x,m);
nzx = find(x); % Find nonzero values in the ordinary array x.
nzg = find(g~=0); % Find nonzero values in the Galois array g.
```

Linear Algebra in Galois Fields

Inverting Matrices and Computing Determinants

To invert a square Galois array, use the `inv` function. Related is the `det` function, which computes the determinant of a Galois array. Both `inv` and `det` behave like their ordinary MATLAB counterparts, except that they perform computations in the Galois field instead of in the field of complex numbers.

Note: A Galois array is singular if and only if its determinant is exactly zero. It is not necessary to consider roundoff errors, as in the case of real and complex arrays.

The code below illustrates matrix inversion and determinant computation.

```
m = 4;
randommatrix = gf(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 array, use the `rank` function. It behaves like the ordinary MATLAB `rank` function when given exactly one input argument. The example below illustrates how to find the rank of square and nonsquare Galois arrays.

```
m = 3;
asquare = gf([4 7 6; 4 6 5; 0 6 1],m);
r1 = rank(asquare);
anonsquare = gf([4 7 6 3; 4 6 5 1; 0 6 1 1],m);
r2 = rank(anonsquare);
[r1 r2]
```

The output is

```
ans =

     2     3
```

The values of `r1` and `r2` indicate that `asquare` has less than full rank but that `anonsquare` has full rank.

Factoring Square Matrices

To express a square Galois array (or a permutation of it) as the product of a lower triangular Galois array and an upper triangular Galois array, use the `lu` function. This function accepts one input argument and produces exactly two or three output arguments. It behaves like the ordinary MATLAB `lu` function when given the same syntax. The example below illustrates how to factor using `lu`.

```
tofactor = gf([6 5 7 6; 5 6 2 5; 0 1 7 7; 1 0 5 1],3);
[L,U]=lu(tofactor); % lu with two output arguments
c1 = isequal(L*U, tofactor) % True
tofactor2 = gf([1 2 3 4;1 2 3 0;2 5 2 1; 0 5 0 0],3);
[L2,U2,P] = lu(tofactor2); % lu with three output arguments
c2 = isequal(L2*U2, P*tofactor2) % True
```

Solving Linear Equations

To find a particular solution of a linear equation in a Galois field, use the `\` or `/` operator on Galois arrays. The table below indicates the equation that each operator addresses, assuming that `A` and `B` are previously defined Galois arrays.

Operator	Linear Equation	Syntax	Equivalent Syntax Using \
Backslash (\)	`A * x = B`	`x = A \ B`	Not applicable
Slash (/)	`x * A = B`	`x = B / A`	`x = (A'\B')'`

The results of the syntax in the table depend on characteristics of the Galois array `A`:

- If `A` is square and nonsingular, the output `x` is the unique solution to the linear equation.
- If `A` is square and singular, the syntax in the table produces an error.
- If `A` is not square, MATLAB attempts to find a particular solution. If `A'*A` or `A*A'` is a singular array, or if `A` is a tall matrix that represents an overdetermined system, the attempt might fail.

Note: An error message does not necessarily indicate that the linear equation has no solution. You might be able to find a solution by rephrasing the problem. For example, `gf([1 2; 0 0],3) \ gf([1; 0],3)` produces an error but the mathematically equivalent `gf([1 2],3) \ gf([1],3)` does not. The first syntax fails because `gf([1 2; 0 0],3)` is a singular square matrix.

Example: Solving Linear Equations

The examples below illustrate how to find particular solutions of linear equations over a Galois field.

```
m = 4;
A = gf(magic(3),m); % Square nonsingular matrix
```

```
Awide=[A, 2*A(:,3)]; % 3-by-4 matrix with redundancy on the right
Atall = Awide'; % 4-by-3 matrix with redundancy at the bottom
B = gf([0:2]',m);
C = [B; 2*B(3)];
D = [B; B(3)+1];
thesolution = A \ B; % Solution of A * x = B
thesolution2 = B' / A; % Solution of x * A = B'
ck1 = all(A * thesolution == B) % Check validity of solutions.
ck2 = all(thesolution2 * A == B')
% Awide * x = B has infinitely many solutions. Find one.
onesolution = Awide \ B;
ck3 = all(Awide * onesolution == B) % Check validity of solution.
% Atall * x = C has a solution.
asolution = Atall \ C;
ck4 = all(Atall * asolution == C) % Check validity of solution.
% Atall * x = D has no solution.
notasolution = Atall \ D;
ck5 = all(Atall * notasolution == D) % It is not a valid solution.
```

The output from this example indicates that the validity checks are all true (`1`), except for `ck5`, which is false (`0`).

Signal Processing Operations in Galois Fields

Section Overview

You can perform some signal-processing operations on Galois arrays, such as filtering, convolution, and the discrete Fourier transform.

This section describes how to perform these operations.

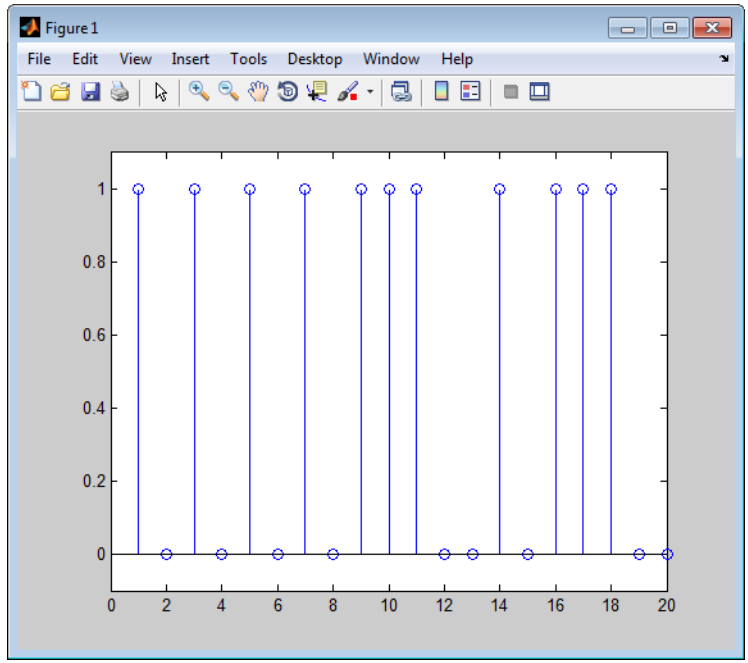
Other information about the corresponding operations for ordinary real vectors is in the Signal Processing Toolbox™ documentation.

Filtering

To filter a Galois vector, use the `filter` function. It behaves like the ordinary MATLAB `filter` function when given exactly three input arguments.

The code and diagram below give the impulse response of a particular filter over GF(2).

```
m = 1; % Work in GF(2).
b = gf([1 0 0 1 0 1 0 1],m); % Numerator
a = gf([1 0 1 1],m); % Denominator
x = gf([1,zeros(1,19)],m);
y = filter(b,a,x); % Filter x.
figure; stem(y.x); % Create stem plot.
axis([0 20 -.1 1.1])
```



Convolution

Communications System 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 4-130. 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 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([1 2 3]',m);
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 array that you can use to perform or invert the discrete Fourier transform on a Galois vector

In all cases, the vector being transformed must be a Galois vector of length 2^m-1 in the field GF(2^m). The following example illustrates the use of these functions. You can check, using the `isequal` function, that `y` equals `y1`, `z` equals `z1`, and `z` equals `x`.

```
m = 4;
x = gf(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

You can use Galois vectors to represent polynomials in an indeterminate quantity x, with coefficients in a Galois field. Form the representation by listing the coefficients of the polynomial in a vector in order of descending powers of x. For example, the vector

```
gf([2 1 0 3],4)
```

represents the polynomial $Ax^3 + 1x^2 + 0x + (A+1)$, where

- A is a primitive element in the field $GF(2^4)$.
- x is the indeterminate quantity in the polynomial.

You can then use such a Galois vector to perform arithmetic with, evaluate, and find roots of polynomials. You can also find minimal polynomials of elements of a Galois field.

Addition and Subtraction of Polynomials

To add and subtract polynomials, use + and - on equal-length Galois vectors that represent the polynomials. If one polynomial has lower degree than the other, you must pad the shorter vector with zeros at the beginning so the two vectors have the same length. The example below shows how to add a degree-one and a degree-two polynomial.

```
lin = gf([4 2],3); % A^2 x + A, which is linear in x
linpadded = gf([0 4 2],3); % The same polynomial, zero-padded
quadr = gf([1 4 2],3); % x^2 + A^2 x + A, which is quadratic in x
% Can't do lin + quadr because they have different vector lengths.
sumpoly = [0, lin] + quadr; % Sum of the two polynomials
sumpoly2 = linpadded + quadr; % The same sum
```

Multiplication and Division of Polynomials

To multiply and divide polynomials, use `conv` and `deconv` on Galois vectors that represent the polynomials. Multiplication and division of polynomials is equivalent to convolution and deconvolution of vectors. The `deconv` function returns the quotient of the two polynomials as well as the remainder polynomial. Examples are below.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
bpoly = gf([1 1],m); % x + 1
xpoly = gf([1 0],m); % x
% Product is A^2 x^3 + x^2 + (A^2 + A) x + (A + 1).
cpoly = conv(apoly,bpoly);
[a2,remd] = deconv(cpoly,bpoly); % a2==apoly. remd is zero.
[otherpol,remd2] = deconv(cpoly,xpoly); % remd is nonzero.
```

The multiplication and division operators in “Arithmetic in Galois Fields” on page 4-114 multiply elements or matrices, not polynomials.

Evaluating Polynomials

To evaluate a polynomial at an element of a Galois field, use `polyval`. It behaves like the ordinary MATLAB `polyval` function when given exactly two input arguments. The example below evaluates a polynomial at several elements in a field and checks the results using `.^` and `.*` in the field.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
x0 = gf([0 1 2],m); % Points at which to evaluate the polynomial
y = polyval(apoly,x0)

a = gf(2,m); % Primitive element of the field, corresponding to A.
y2 = a.^2.*x0.^2 + (a.^2+1).*x0 + (a+1) % Check the result.
```

The output is below.

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

```
Array elements =

     3     2    10


y2 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10
```

The first element of `y` evaluates the polynomial at `0` and, therefore, returns the polynomial's constant term of `3`.

Roots of Polynomials

To find the roots of a polynomial in a Galois field, use the `roots` function on a Galois vector that represents the polynomial. This function finds roots that are in the same field that the Galois vector is in. The number of times an entry appears in the output vector from `roots` is exactly its multiplicity as a root of the polynomial.

Note: If the Galois vector is in GF(2^m), the polynomial it represents might have additional roots in some extension field GF($(2^m)^k$). However, `roots` does not find those additional roots or indicate their existence.

The examples below find roots of cubic polynomials in GF(8).

```
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 GF(4) as well as in GF(16), so `roots4` and `roots16` are nonempty.

Notice that `roots4` and `roots16` are not equal to each other. They differ in these ways:

- `roots4` is a GF(4) array, while `roots16` is a GF(16) array. MATLAB keeps track of the underlying field of a Galois array.
- The array elements in `roots4` and `roots16` differ because they use representations with respect to different primitive polynomials. For example, `2` (which represents a primitive element) is an element of the vector `roots4` because the default primitive polynomial for GF(4) is the same polynomial that `gf4poly` represents. On the other hand, `2` is not an element of `roots16` because the primitive element of GF(16) is not a root of the polynomial that `gf16poly` represents.

Minimal Polynomials

The minimal polynomial of an element of $GF(2^m)$ is the smallest degree nonzero binary-coefficient polynomial having that element as a root in $GF(2^m)$. To find the minimal polynomial of an element or a column vector of elements, use the `minpol` function.

The code below finds that the minimal polynomial of `gf(6,4)` is $D^2 + D + 1$ and then checks that `gf(6,4)` is indeed among the roots of that polynomial in the field GF(16).

```
m = 4;
e = gf(6,4);
em = minpol(e) % Find minimal polynomial of e. em is in GF(2).

emr = roots(gf([0 0 1 1 1],m)) % Roots of D^2+D+1 in GF(2^m)
```

The output is

```
em = GF(2) array.

Array elements =

     0     0     1     1     1


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

Array elements =

     6
     7
```

To find out which elements of a Galois field share the same minimal polynomial, use the `cosets` function.

Manipulating Galois Variables

Section Overview

This section describes techniques for manipulating Galois variables or for transferring information between Galois arrays and ordinary MATLAB arrays.

Note: These techniques are particularly relevant if you write MATLAB file functions that process Galois arrays. For an example of this type of usage, enter `edit gf/conv` in the Command Window and examine the first several lines of code in the editor window.

Determining Whether a Variable Is a Galois Array

To find out whether a variable is a Galois array rather than an ordinary MATLAB array, use the `isa` function. An illustration is below.

```
mlvar = eye(3);
gfvar = gf(mlvar,3);
no = isa(mlvar,'gf'); % False because mlvar is not a Galois array
yes = isa(gfvar,'gf'); % True because gfvar is a Galois array
```

Extracting Information from a Galois Array

To extract the array elements, field order, or primitive polynomial from a variable that is a Galois array, append a suffix to the name of the variable. The table below lists the exact suffixes, which are independent of the name of the variable.

Information	Suffix	Output Value
Array elements	`.x`	MATLAB array of type `uint16` that contains the data values from the Galois array.
Field order	`.m`	Integer of type `double` that indicates that the Galois array is in GF(`2^m`).
Primitive polynomial	`.prim_poly`	Integer of type `uint32` that represents the primitive polynomial. The representation is similar to the description in "How Integers Correspond to Galois Field Elements" on page 4-110.

Note: If the output value is an integer data type and you want to convert it to `double` for later manipulation, use the `double` function.

The code below illustrates the use of these suffixes. The definition of `empr` uses a vector of binary coefficients of a polynomial to create a Galois array in an extension field.

Another part of the example retrieves the primitive polynomial for the field and converts it to a binary vector representation having the appropriate number of bits.

```
% Check that e solves its own minimal polynomial.
e = gf(6,4); % An element of GF(16)
emp = minpol(e); % The minimal polynomial, emp, is in GF(2).
empr = roots(gf(emp.x,e.m)); % Find roots of emp in GF(16).

% Check that the primitive element gf(2,m) is
% really a root of the primitive polynomial for the field.
primpoly_int = double(e.prim_poly);
mval = e.m;
primpoly_vect = gf(de2bi(primpoly_int,mval+1,'left-msb'),mval);
containstwo = roots(primpoly_vect); % Output vector includes 2.
```

Converting Galois Array to Doubles

```
a = gf([1,0])
b = double(a.x) %a.x is in uint16
```

MATLAB returns the following:

```
a = GF(2) array.

Array elements =

           1           0


b =

     1     0
```

Speed and Nondefault Primitive Polynomials

The section "Specifying the Primitive Polynomial" on page 4-112 described how to represent elements of a Galois field with respect to a primitive polynomial of your choice. This section describes how you can increase the speed of computations involving a Galois array that uses a primitive polynomial other than the default primitive polynomial. The technique is recommended if you perform many such computations.

The mechanism for increasing the speed is a data file, `userGftable.mat`, that some computational functions use to avoid performing certain computations repeatedly. To take advantage of this mechanism for your combination of field order (`m`) and primitive polynomial (`prim_poly`):

1. Navigate in the MATLAB application to a folder to which you have write permission. You can use either the `cd` function or the Current Folder feature to navigate.
2. Define `m` and `prim_poly` as workspace variables. For example:

   ```
   m = 3; prim_poly = 13; % Examples of valid values
   ```

3. Invoke the `gftable` function:

   ```
   gftable(m,prim_poly); % If you previously defined m and prim_poly
   ```

The function revises or creates `userGftable.mat` in your current working folder to include data relating to your combination of field order and primitive polynomial. After you initially invest the time to invoke `gftable`, subsequent computations using those values of `m` and `prim_poly` should be faster.

Note: If you change your current working directory after invoking `gftable`, you must place `userGftable.mat` on your MATLAB path to ensure that MATLAB can see it. Do this by using the `addpath` command to prefix the directory containing `userGftable.mat` to your MATLAB path. If you have multiple copies of `userGftable.mat` on your path, use `which('userGftable.mat','-all')` to find out where they are and which one MATLAB is using.

To see how much `gftable` improves the speed of your computations, you can surround your computations with the `tic` and `toc` functions. See the `gftable` reference page for an example.

Selected Bibliography for Galois Fields

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, MA, Addison-Wesley, 1993.

[3] Lin, Shu, and Daniel J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1983.

[4] van Lint, J. H., *Introduction to Coding Theory*, New York, Springer-Verlag, 1982.

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

Galois Fields of Odd Characteristic

A *Galois field* is an algebraic field having p^m elements, where p is prime and m is a positive integer. This chapter describes how to work with Galois fields in which p is *odd*. To work with Galois fields having an even number of elements, see Galois Field Computations. The sections in this chapter are as follows.

Galois Field Terminology

Throughout this section, p is an odd prime number and m is a positive integer.

Also, this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [5].

- A *primitive element* of $GF(p^m)$ is a cyclic generator of the group of nonzero elements of $GF(p^m)$. This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power. Primitive elements are called A throughout this section.
- A *primitive polynomial* for $GF(p^m)$ is the minimal polynomial of some primitive element of $GF(p^m)$. As a consequence, it has degree m and is irreducible.

Representing Elements of Galois Fields

Section Overview

This section discusses how to represent Galois field elements using this toolbox's exponential format and polynomial format. It also describes a way to list all elements of the Galois field, because some functions use such a list as an input argument. Finally, it discusses the nonuniqueness of representations of Galois field elements.

The elements of GF(p) can be represented using the integers from 0 to p-1.

When m is at least 2, GF(p^m) is called an extension field. Integers alone cannot represent the elements of GF(p^m) in a straightforward way. MATLAB technical computing software uses two main conventions for representing elements of GF(p^m): the exponential format and the polynomial format.

Note: Both the exponential format and the polynomial format are relative to your choice of a particular primitive element A of GF(p^m).

Exponential Format

This format uses the property that every nonzero element of GF(p^m) can be expressed as A^c for some integer c between 0 and p^m-2. Higher exponents are not needed, because the theory of Galois fields implies that every nonzero element of GF(p^m) satisfies the equation $x^{q-1} = 1$ where $q = p^m$.

The use of the exponential format is shown in the table below.

Element of GF(p^m)	MATLAB Representation of the Element
0	`-Inf`
$A^0 = 1$	`0`
A^1	`1`
...	...
A^{q-2} where `q` = p^m	`q-2`

Although `-Inf` is the standard exponential representation of the zero element, all negative integers are equivalent to `-Inf` when used as *input* arguments in exponential format. This equivalence can be useful; for example, see the concise line of code at the end of the section “Default Primitive Polynomials” on page 4-141.

Note: The equivalence of all negative integers and `-Inf` as exponential formats means that, for example, -1 does *not* represent A^{-1}, the multiplicative inverse of A. Instead, -1 represents the zero element of the field.

Polynomial Format

The polynomial format uses the property that every element of $GF(p^m)$ can be expressed as a polynomial in A with exponents between 0 and m-1, and coefficients in GF(p). In the polynomial format, the element

`A(1)` + `A(2)` A + `A(3)` A^2 + ... + `A(m)` A^{m-1}

is represented in MATLAB by the vector

```
[A(1) A(2) A(3) ... A(m)]
```

Note: The Galois field functions in this toolbox represent a polynomial as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

List of All Elements of a Galois Field

Some Galois field functions in this toolbox require an argument that lists all elements of an extension field $GF(p^m)$. This is again relative to a particular primitive element A of $GF(p^m)$. The proper format for the list of elements is that of a matrix having p^m rows, one for each element of the field. The matrix has m columns, one for each coefficient of a power of A in the polynomial format shown in "Polynomial Format" on page 4-139 above. The first row contains only zeros because it corresponds to the zero element in $GF(p^m)$. If k is between 2 and p^m, then the kth row specifies the polynomial format of the element A^{k-2}.

The minimal polynomial of A aids in the computation of this matrix, because it tells how to express A^m in terms of lower powers of A. For example, the table below lists the elements of $GF(3^2)$, where A is a root of the primitive polynomial $2 + 2x + x^2$. This polynomial allows repeated use of the substitution

$A^2 = -2 - 2A = 1 + A$

when performing the computations in the middle column of the table.

Elements of GF(9)

Exponential Format	Polynomial Format	Row of MATLAB Matrix of Elements
A^{-Inf}	0	0 0
A^0	1	1 0
A^1	A	0 1
A^2	1+A	1 1
A^3	$A + A^2 = A + 1 + A = 1 + 2A$	1 2
A^4	$A + 2A^2 = A + 2 + 2A = 2$	2 0
A^5	2A	0 2
A^6	$2A^2 = 2 + 2A$	2 2
A^7	$2A + 2A^2 = 2A + 2 + 2A = 2 + A$	2 1

Example

An automatic way to generate the matrix whose rows are in the third column of the table above is to use the code below.

```
p = 3; m = 2;
% Use the primitive polynomial 2 + 2x + x^2 for GF(9).
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

The `gftuple` function is discussed in more detail in “Converting and Simplifying Element Formats” on page 4-141.

Nonuniqueness of Representations

A given field has more than one primitive element. If two primitive elements have different minimal polynomials, then the corresponding matrices of elements will have their rows in a different order. If the two primitive elements share the same minimal polynomial, then the matrix of elements of the field is the same.

Note: You can use whatever primitive element you want, as long as you understand how the inputs and outputs of Galois field functions depend on the choice of *some* primitive

polynomial. It is usually best to use the same primitive polynomial throughout a given script or function.

Other ways in which representations of elements are not unique arise from the equations that Galois field elements satisfy. For example, an exponential format of 8 in GF(9) is really the same as an exponential format of 0, because $A^8 = 1 = A^0$ in GF(9). As another example, the substitution mentioned just before the table Elements of GF(9) shows that the polynomial format [0 0 1] is really the same as the polynomial format [1 1].

Default Primitive Polynomials

This toolbox provides a *default* primitive polynomial for each extension field. You can retrieve this polynomial using the `gfprimdf` function. The command

```
prim_poly = gfprimdf(m,p); % If m and p are already defined
```

produces the standard row-vector representation of the default minimal polynomial for $GF(p^m)$.

For example, the command below shows that the default primitive polynomial for GF(9) is $2 + x + x^2$, *not* the polynomial used in "List of All Elements of a Galois Field" on page 4-139.

```
poly1=gfprimdf(2,3);


poly1 =

     2     1     1
```

To generate a list of elements of $GF(p^m)$ using the default primitive polynomial, use the command

```
field = gftuple([-1:p^m-2]',m,p);
```

Converting and Simplifying Element Formats

Converting to Simplest Polynomial Format

The `gftuple` function produces the simplest polynomial representation of an element of $GF(p^m)$, given either an exponential representation or a polynomial representation of that element. This can be useful for generating the list of elements of $GF(p^m)$ that other functions require.

Using `gftuple` requires three arguments: one representing an element of $GF(p^m)$, one indicating the primitive polynomial that MATLAB technical computing software should use when computing the output, and the prime p. The table below indicates how `gftuple` behaves when given the first two arguments in various formats.

Behavior of gftuple Depending on Format of First Two Inputs

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Exponential format; c = any integer	Integer m > 1	Polynomial format of A^c, where A is a root of the *default* primitive polynomial for $GF(p^m)$
Example: `tp = gftuple(6,2,3); % c = 6 here`		
Exponential format; c = any integer	Vector of coefficients of primitive polynomial	Polynomial format of A^c, where A is a root of the *given* primitive polynomial
Example: `polynomial = gfprimdf(2,3); tp = gftuple(6,polynomial,3); % c = 6 here`		
Polynomial format of any degree	Integer m > 1	Polynomial format of degree < m, using *default* primitive polynomial for $GF(p^m)$ to simplify
Example: `tp = gftuple([0 0 0 0 0 0 1],2,3);`		
Polynomial format of any degree	Vector of coefficients of primitive polynomial	Polynomial format of degree < m, using the *given* primitive polynomial for $GF(p^m)$ to simplify
Example: `polynomial = gfprimdf(2,3); tp = gftuple([0 0 0 0 0 0 1],polynomial,3);`		

The four examples that appear in the table above all produce the same vector `tp = [2, 1]`, but their different inputs to `gftuple` correspond to the lines of the table. Each example expresses the fact that $A^6 = 2+A$, where A is a root of the (default) primitive polynomial $2 + x + x^2$ for $GF(3^2)$.

Example

This example shows how `gfconv` and `gftuple` combine to multiply two polynomial-format elements of $GF(3^4)$. Initially, `gfconv` multiplies the two polynomials, treating the primitive element as if it were a variable. This produces a high-order polynomial, which `gftuple` simplifies using the polynomial equation that the primitive element satisfies. The final result is the simplest polynomial format of the product.

```
p = 3; m = 4;
a = [1 2 0 1]; b = [2 2 1 2];
notsimple = gfconv(a,b,p) % a times b, using high powers of alpha
simple = gftuple(notsimple,m,p) %Highest exponent of alpha is m-1
```

The output is below.

```
notsimple =

     2     0     2     0     0     1     2


simple =

     2     1     0     1
```

Example: Generating a List of Galois Field Elements

This example applies the conversion functionality to the task of generating a matrix that lists all elements of a Galois field. A matrix that lists all field elements is an input argument in functions such as `gfadd` and `gfmul`. The variables `field1` and `field2` below have the format that such functions expect.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field1 = gftuple([-1:p^m-2]',m,p);

prim_poly = gfprimdf(m,p); % Or any primitive polynomial
% for GF(p^m)
field2 = gftuple([-1:p^m-2]',prim_poly,p);
```

Converting to Simplest Exponential Format

The same function `gftuple` also produces the simplest exponential representation of an element of GF(p^m), given either an exponential representation or a polynomial representation of that element. To retrieve this output, use the syntax

```
[polyformat, expformat] = gftuple(...)
```

The input format and the output `polyformat` are as in the table 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 p^m-2.

Example

To recover the exponential format of the element 2 + A that the previous section considered, use the commands below. In this case, `polyformat` contains redundant information, while `expformat` contains the desired result.

```
[polyformat, expformat] = gftuple([2 1],2,3)


polyformat =

     2     1

expformat =

     6
```

This output appears at first to contradict the information in the table 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

You can add, subtract, multiply, and divide elements of Galois fields using the functions `gfadd`, `gfsub`, `gfmul`, and `gfdiv`, respectively. Each of these functions has a mode for prime fields and a mode for extension fields.

Arithmetic in Prime Fields

Arithmetic in GF(p) is the same as arithmetic modulo p. The functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` accept two arguments that represent elements of GF(p) as integers between 0 and p-1. The third argument specifies p.

Example: Addition Table for GF(5)

The code below constructs an addition table for GF(5). If `a` and `b` are between 0 and 4, then the element `gfp_add(a+1,b+1)` represents the sum `a+b` in GF(5). For example, `gfp_add(3,5) = 1` because 2+4 is 1 modulo 5.

```
p = 5;
row = 0:p-1;
table = ones(p,1)*row;
gfp_add = gfadd(table,table',p)
```

The output for this example follows.

```
gfp_add =

     0     1     2     3     4
     1     2     3     4     0
     2     3     4     0     1
     3     4     0     1     2
     4     0     1     2     3
```

Other values of `p` produce tables for different prime fields GF(p). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(p).

Arithmetic in Extension Fields

The same arithmetic functions can add elements of $GF(p^m)$ when m > 1, but the format of the arguments is more complicated than in the case above. In general, arithmetic in extension fields is more complicated than arithmetic in prime fields; see the works listed in "Selected Bibliography for Galois Fields" on page 4-151 for details about how the arithmetic operations work.

When working in extension fields, the functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` use the first two arguments to represent elements of $GF(p^m)$ in exponential format. The third argument, which is required, lists all elements of $GF(p^m)$ as described in "List of All Elements of a Galois Field" on page 4-139. The result is in exponential format.

Example: Addition Table for GF(9)

The code below constructs an addition table for $GF(3^2)$, using exponential formats relative to a root of the default primitive polynomial for GF(9). If `a` and `b` are between -1 and 7, then the element `gfpm_add(a+2,b+2)` represents the sum of A^a and A^b in GF(9). For example, `gfpm_add(4,6) = 5` because

$A^2 + A^4 = A^5$

Using the fourth and sixth rows of the matrix `field`, you can verify that

$A^2 + A^4 = (1 + 2A) + (2 + 0A) = 3 + 2A = 0 + 2A = A^5$ modulo 3.

```
p = 3; m = 2; % Work in GF(3^2).
field = gftuple([-1:p^m-2]',m,p); % Construct list of elements.
row = -1:p^m-2;
table = ones(p^m,1)*row;
gfpm_add = gfadd(table,table',field)
```

The output is below.

```
gfpm_add =

  -Inf     0     1     2     3     4     5     6     7
     0     4     7     3     5  -Inf     2     1     6
     1     7     5     0     4     6  -Inf     3     2
     2     3     0     6     1     5     7  -Inf     4
     3     5     4     1     7     2     6     0  -Inf
     4  -Inf     6     5     2     0     3     7     1
     5     2  -Inf     7     6     3     1     4     0
```

```
     6     1     3  -Inf     0     7     4     2     5
     7     6     2     4  -Inf     1     0     5     3
```

Note: If you used a different primitive polynomial, then the tables would look different. This makes sense because the ordering of the rows and columns of the tables was based on that particular choice of primitive polynomial and not on any natural ordering of the elements of GF(9).

Other values of `p` and `m` produce tables for different extension fields GF(`p^m`). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(`p^m`).

Polynomials over Prime Fields

Section Overview

A polynomial over GF(p) is a polynomial whose coefficients are elements of GF(p). Communications System Toolbox software provides functions for

- Changing polynomials in cosmetic ways
- Performing polynomial arithmetic
- Characterizing polynomials as primitive or irreducible
- Finding roots of polynomials in a Galois field

Note: The Galois field functions in this toolbox represent a polynomial over GF(p) for odd values of p as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

Cosmetic Changes of Polynomials

To display the traditionally formatted polynomial that corresponds to a row vector containing coefficients, use `gfpretty`. To truncate a polynomial by removing all zero-

coefficient terms that have exponents *higher* than the degree of the polynomial, use `gftrunc`. For example,

```
polynom = gftrunc([1 20 394 10 0 0 29 3 0 0])
gfpretty(polynom)
```

The output is below.

```
polynom =

     1    20   394    10     0     0    29     3


                                 2       3       6       7
                1 + 20 X + 394 X  + 10 X  + 29 X  + 3 X
```

Note: If you do not use a fixed-width font, then the spacing in the display might not look correct.

Polynomial Arithmetic

The functions `gfadd` and `gfsub` add and subtract, respectively, polynomials over GF(p). The `gfconv` function multiplies polynomials over GF(p). The `gfdeconv` function divides polynomials in GF(p), producing a quotient polynomial and a remainder polynomial. For example, the commands below show that $2 + x + x^2$ times $1 + x$ over the field GF(3) is $2 + 2x^2 + x^3$.

```
a = gfconv([2 1 1],[1 1],3)
[quot, remd] = gfdeconv(a,[2 1 1],3)
```

The output is below.

```
a =

     2     0     2     1


quot =

     1     1

remd =
```

```
     0
```

The previously discussed functions `gfadd` and `gfsub` add and subtract, respectively, polynomials. Because it uses a vector of coefficients to represent a polynomial, MATLAB does not distinguish between adding two polynomials and adding two row vectors elementwise.

Characterization of Polynomials

Given a polynomial over GF(p), the `gfprimck` function determines whether it is irreducible and/or primitive. By definition, if it is primitive then it is irreducible; however, the reverse is not necessarily true. The `gfprimdf` and `gfprimfd` functions return primitive polynomials.

Given an element of $GF(p^m)$, the `gfminpol` function computes its minimal polynomial over GF(p).

Example

For example, the code below reflects the irreducibility of all minimal polynomials. However, the minimal polynomial of a nonprimitive element is not a primitive polynomial.

```
p = 3; m = 4;
% Use default primitive polynomial here.

prim_poly = gfminpol(1,m,p);
ckprim = gfprimck(prim_poly,p);
% ckprim = 1, since prim_poly represents a primitive polynomial.

notprimpoly = gfminpol(5,m,p);
cknotprim = gfprimck(notprimpoly,p);
% cknotprim = 0 (irreducible but not primitive)
% since alpha^5 is not a primitive element when p = 3.

ckreducible = gfprimck([0 1 1],p);
% ckreducible = -1 since the polynomial is reducible.
```

Roots of Polynomials

Given a polynomial over GF(p), the `gfroots` function finds the roots of the polynomial in a suitable extension field $GF(p^m)$. There are two ways to tell MATLAB the degree m of the extension field $GF(p^m)$, as shown in the following table.

Formats for Second Argument of gfroots

Second Argument	Represents
A positive integer	m as in GF(p^m). MATLAB uses the default primitive polynomial in its computations.
A row vector	A primitive polynomial for GF(p^m). Here m is the degree of this primitive polynomial.

Example: Roots of a Polynomial in GF(9)

The code below finds roots of the polynomial $1 + x^2 + x^3$ in GF(9) and then checks that they are indeed roots. The exponential format of elements of GF(9) is used throughout.

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
   root = rts(ii);
   rootsquared = gfmul(root,root,field);
   rootcubed = gfmul(root,rootsquared,field);
   answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
   % Recall that 1 is really alpha to the zero power.
   % If answer = -Inf, then the variable root represents
   % a root of the polynomial.
end
answer
```

The output shows that A^0 (which equals 1), A^5, and A^7 are roots.

```
roots =

     0
     5
     7


answer =

  -Inf  -Inf  -Inf
```

See the reference page for `gfroots` to see how `gfroots` can also provide you with the polynomial formats of the roots and the list of all elements of the field.

Other Galois Field Functions

See the online reference pages for information about these other Galois field functions in Communications System 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

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 explicitly as an input argument.
Algebraic interleaver	`algintrlv`	Derives a permutation table algebraically, using the Takeshita-Costello or Welch-Costas method. These methods are described in [4].
Helical scan interleaver	`helscanintrlv`	Fills a matrix with data row by row and then sends the matrix contents to the output in a helical fashion.
Matrix interleaver	`matintrlv`	Fills a matrix with data elements row by row and then sends the matrix contents to the output column by column.
Random interleaver	`randintrlv`	Chooses a permutation table randomly using the initial state input that you provide.

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 `[1 2 3 4 5 6]`, the block produces an output of `[1 4 2 5 3 6]`.

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];

% 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
```

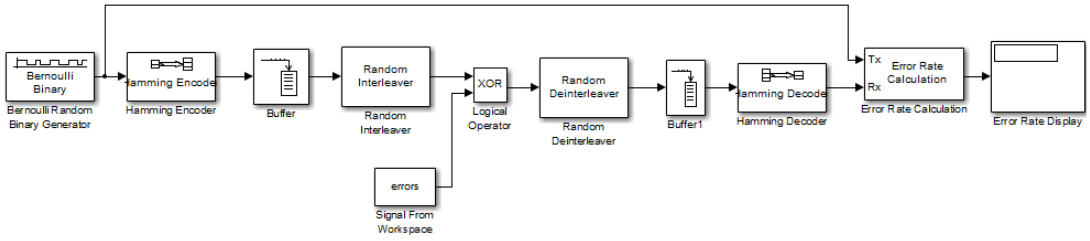
The output from the example follows.

```
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 4-155. 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 System 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 System 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.

Click the **Simulation** menu and select **Model Configuration parameters**. Set **Stop time** to `length(errors)`.

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 1s, representing bursts of errors, at random intervals. The distance between two consecutive blocks of 1s 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)=[1 1 1];
end
```

To determine the ratio of the number of 1s 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 1s, the expected distance to the next sequence of 1s is 40. Consequently, you expect to see three 1s in 43 terms of the sequence. If there were no error correction in the model, the bit error rate would be approximately .0698.

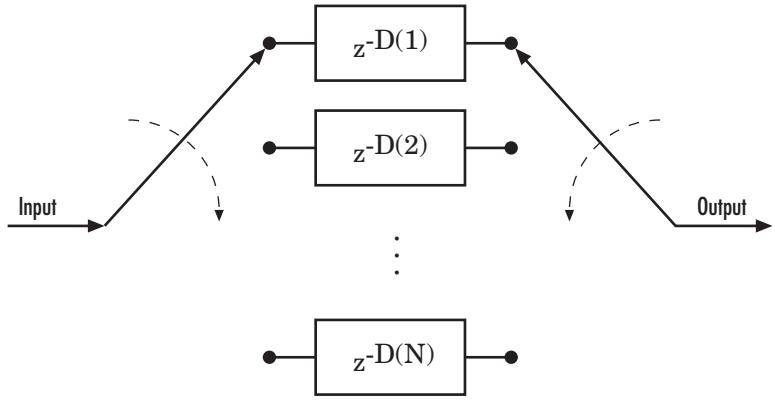
When you run a simulation with the model, the error rate is approximately .019, which shows the improvement due to error correction and interleaving. You can see the effect of interleaving by deleting the Random Interleaver and Random Deinterleaver blocks from the model, connecting the lines, and running 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

A convolutional interleaver consists of a set of shift registers, each with a fixed delay. In a typical convolutional interleaver, the delays are nonnegative integer multiples of a fixed integer (although a general multiplexed interleaver allows unrestricted delay values). Each new symbol from an input vector feeds into the next shift register and the oldest symbol in that register becomes part of the output vector. A convolutional interleaver has memory; that is, its operation depends not only on current symbols but also on previous symbols.

The schematic below depicts the structure of a general convolutional interleaver by showing the set of shift registers and their delay values D(1), D(2),..., D(N). The kth shift register holds D(k) symbols, where k = 1,2,...,N. The convolutional interleaving functions in this toolbox have input arguments that indicate the number of shift registers and the delay for each shift register.



Communications System 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 interleaver	`muxintrlv`	Allows unrestricted delay values for the set of shift registers.
Convolutional interleaver	`convintrlv`	The delay values for the set of shift registers are nonnegative integer multiples of a fixed integer that you specify.
Helical interleaver	`helintrlv`	Fills an array with input symbols in a helical fashion and empties the array row by row.

The `helscanintrlv` function and the `helintrlv` function both use a helical array for internal computations. However, the two functions have some important differences:

- `helintrlv` uses an unlimited-row array, arranges input symbols in the array along columns, outputs some symbols that are not from the current input, and leaves some input symbols in the array without placing them in the output.
- `helscanintrlv` uses a fixed-size matrix, arranges input symbols in the array across rows, and outputs all the input symbols without using any default values or values from a previous call.

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); ...; 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) * (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 * (N-1)

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

$$CN\left\lceil \frac{s(C-1)}{N} \right\rceil$$

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
servec1 = 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
     0
     0
     0
```

```
     0
     0
     0
     0
     0
    59
    42
     1
    28
    52
    54
    43
     8
    56
     5
    35
    37
    48
    17
    28
    62
    10
    31
    61
    39

ser1 =

     0

b2 =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

```
     0
     0
    59
    42
     1
    28
    52
    54
    43
     8


ser2 =

     0
```

Combining Interleaving Delays and Other Delays

If you use convolutional interleavers in a script that incurs an additional delay, d, between the interleaver output and the deinterleaver input (for example, a delay from a filter), then the restored sequence lags behind the original sequence by the sum of d and the amount from the table Delays of Interleaver/Deinterleaver Pairs. In this case, d must be an integer multiple of the number of shift registers, or else the convolutional deinterleaver cannot recover the original symbols properly. If d is not naturally an integer multiple of the number of shift registers, then you can adjust the delay manually by padding the vector that forms the input to the deinterleaver.

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	[ ] [0] [0 0]
2	2	0	[ ] [2] [0 0]
3	3	0	[ ] [2] [0 3]
4	1	4	[ ] [2] [0 3]
5	2	2	[ ] [5] [0 3]
6	3	0	[ ] [5] [3 6]
7	1	7	[ ] [5] [3 6]
8	2	5	[ ] [8] [3 6]
9	3	3	[ ] [8] [6 9]
10	1	10	[ ] [8] [6 9]

The output from the example is below.

```
y =
```

```
     1
     0
     0
     4
     2
     0
     7
     5
     3
    10


state_y =

    value: {3x1 cell}
    index: 2


z =

     0
     0
     0
     0
     0
     0
     1
     2
     3
     4
```

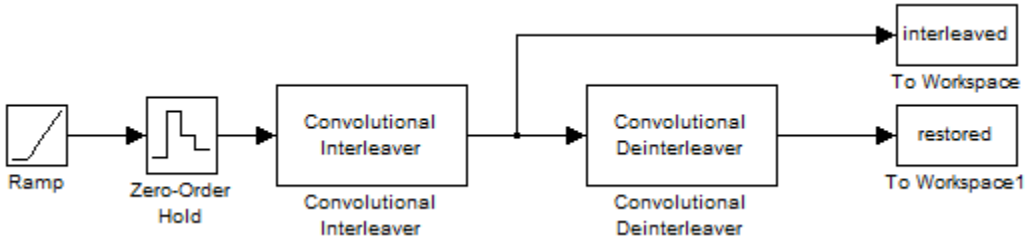
Notice that the "Current Output" column of the table above agrees with the values in the vector `y`. Also, the last row of the table above indicates that the last shift register processed for the given data set is the first shift register. This agrees with the value of `2` for `state_y.index`, which indicates that any additional input data would be directed to the second shift register. You can optionally check that the state values listed in `state_y.value` match the "Contents of Shift Registers" entry in the last row of the table by typing `state_y.value{:}` in the Command Window after executing the example.

Another feature to notice about the example output is that `z` contains six zeros at the beginning before containing any of the symbols from the original data set. The six

zeros illustrate that the delay of this convolutional interleaver/deinterleaver pair is `length(delay)*max(delay) = 3*2 = 6`. For more information about delays, see "Delays of Convolutional Interleavers" on page 4-160.

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.



To open the completed model, click here in the MATLAB Help browser. 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 `[-1 -2 -3]'`.
- Convolutional Deinterleaver
  - Set **Rows of shift registers** to `3`.
  - Set **Initial conditions** to `[-1 -2 -3]'`.
- 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 diagram. From the model window's **Simulation** menu, select **Model Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to 20. Run the simulation and execute the following command:

```
comparison = [[0:20]', interleaved, restored]

comparison =

     0     0    -1
     1    -2    -2
     2    -3    -3
     3     3    -1
     4    -2    -2
     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" in the Communications System Toolbox documentation set.

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

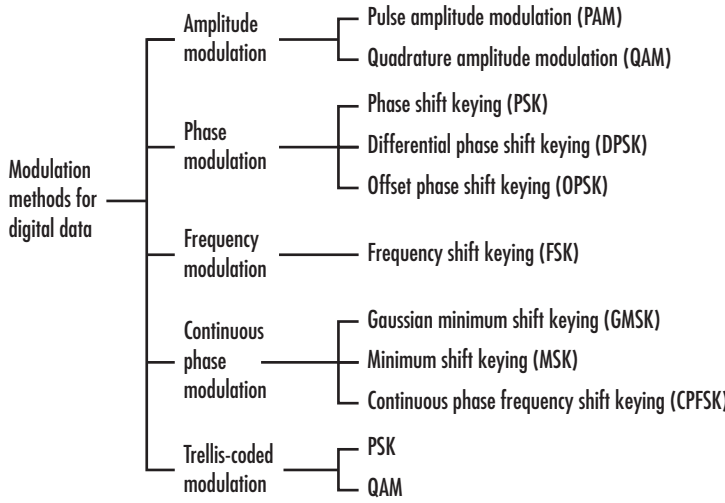
In most media for communication, only a fixed range of frequencies is available for transmission. One way to communicate a message signal whose frequency spectrum does not fall within that fixed frequency range, or one that is otherwise unsuitable for the channel, is to alter a transmittable signal according 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*.

The sections of this chapter are as follows.

Digital Modulation Features

Modulation Techniques

The figure below shows the modulation techniques that Communications System Toolbox supports 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, in this case, the message signal is restricted to a finite set. Using this product, you can modulate or demodulate signals using various digital modulation techniques. You can also plot signal constellations. Modulation functions output the complex envelope of the modulated signal.

Note: The modulation and demodulation functions do not perform pulse shaping or filtering. See "Combine Pulse Shaping and Filtering with Modulation" on page 4-191 for more information about filtering.

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

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

Analog Modulation Method	Acronym	Function or Method
Single sideband amplitude modulation	SSB	`ssbmod`, `ssbdemod`

Digital Modulation Method	Acronym	System object
Differential phase shift keying modulation	DPSK	`comm.DPSKDemodulator`System object, `comm.DPSKModulator` System object
Frequency shift keying modulation	FSK	comm.FSKDemodulator System object, comm.FSKModulator System object,
General Quadrature amplitude modulation	General QAM	`comm.GeneralQAMDemodulator` System object, `comm.GeneralQAMModulator` System object
Minimum shift keying modulation	MSK	`comm.MSKDemodulator` System object, `comm.MSKModulator` System object
Offset quadrature phase shift keying modulation	OQPSK	`comm.OQPSKDemodulator` System object, `comm.OQPSKModulator` System object
Phase shift keying modulation	PSK	`comm.PSKDemodulator` System object, `comm.PSKModulator` System object
Pulse amplitude modulation	PAM	`comm.PAMDemodulator` System object, `comm.PAMModulator` System object

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 double-clicking 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 special cases of a more general technique and are, in fact, special cases of a more general block. These special-case blocks use the same computational code that their general counterparts use, but provide an interface that is either simpler or more suitable for the special case. The following table lists special-case modulators, their general counterparts, and the conditions under which the two are equivalent. The situation is analogous for demodulators.

General and Specific Blocks

General Modulator	Specific Modulator	Specific Conditions
General QAM Modulator Baseband	Rectangular QAM Modulator Baseband	Predefined constellation containing 2^K points on a rectangular lattice
M-PSK Modulator Baseband	BPSK Modulator Baseband	**M-ary number** parameter is 2.
	QPSK Modulator Baseband	**M-ary number** parameter is 4.
M-DPSK Modulator Baseband	DBPSK Modulator Baseband	**M-ary number** parameter is 2.
	DQPSK Modulator Baseband	**M-ary number** parameter is 4.
CPM Modulator Baseband	GMSK Modulator Baseband	**M-ary number** parameter is 2, **Frequency pulse shape** parameter is `Gaussian`.

General Modulator	Specific Modulator	Specific Conditions
	MSK Modulator Baseband	**M-ary number** parameter is 2, **Frequency pulse shape** parameter is `Rectangular`, **Pulse length** parameter is 1.
	CPFSK Modulator Baseband	**Frequency pulse shape** parameter is `Rectangular`, **Pulse length** parameter is 1.
General TCM Encoder	Rectangular QAM TCM Encoder	Predefined signal constellation containing 2^K points on a rectangular lattice
	M-PSK TCM Encoder	Predefined signal constellation containing 2^K points on a circle

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. This product 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

$$Y_1(t)\cos(2\pi f_c t + \theta) - Y_2(t)\sin(2\pi f_c t + \theta)$$

where f_c is the carrier frequency and θ is the carrier signal's initial phase, then a baseband simulation recognizes that this equals the real part of

$$[(Y_1(t) + jY_2(t))e^{j\theta}]\exp(j2\pi f_c t)$$

and models only the part inside the square brackets. Here *j* is the square root of -1. The complex vector `y` is a sampling of the complex signal

$$(Y_1(t) + jY_2(t))e^{j\theta}$$

If you prefer to work with passband signals instead of baseband signals, then you can build functions that convert between the two. Be aware that passband modulation tends to be more computationally intensive than baseband modulation because the carrier signal typically needs to be sampled at a high rate.

Modulation Terminology

Modulation is a process by which a *carrier signal* is altered according to information in a *message signal*. The *carrier frequency*, denoted `Fc`, is the frequency of the carrier signal. The *sampling rate* is the rate at which the message signal is sampled during the simulation.

The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. The Nyquist sampling theorem requires that the simulation sampling rate `Fs` be greater than two times the sum of the carrier frequency and the highest frequency of the modulated signal in order for the demodulator to recover the message correctly.

Representing Digital Signals

To modulate a signal using digital modulation with an alphabet having M symbols, start with a real message signal whose values are integers from 0 to M-1. Represent the signal by listing its values in a vector, `x`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

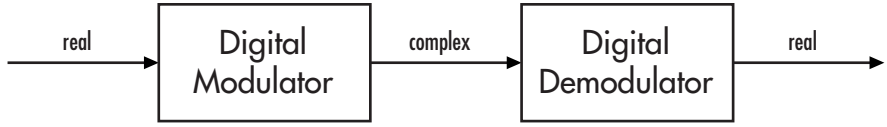
For example, if the modulation uses an alphabet with eight symbols, then the vector `[2 3 7 1 0 5 5 2 6]'` is a valid single-channel input to the modulator. As a multichannel example, the two-column matrix

```
[2 3;
 3 3;
 7 3;
 0 3;]
```

defines a two-channel signal in which the second channel has a constant value of `3`.

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 between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of *K* bits.

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) ... u(K)] to the integer

$$\sum_{i=1}^{K} u(i)2^{K-i}$$

and assumes that this integer is the input value. u(1) is the most significant bit.

If you set M = 8, **Constellation ordering** (or **Symbol set ordering**) to `Binary`, and the binary input word is [1 1 0], the block converts [1 1 0] to the integer 6. The block produces the same output when the input is 6 and the **Input type** parameter is `Integer`.

When you set **Constellation ordering** (or **Symbol set ordering**) to `Gray`, the block uses a Gray-coded arrangement and assigns binary inputs to points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
```

to the P^{th} integer.

The following tables show the typical Binary to Gray mapping for M = 8.

Binary to Gray Mapping for Bits

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111

Binary Code	Gray Code
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

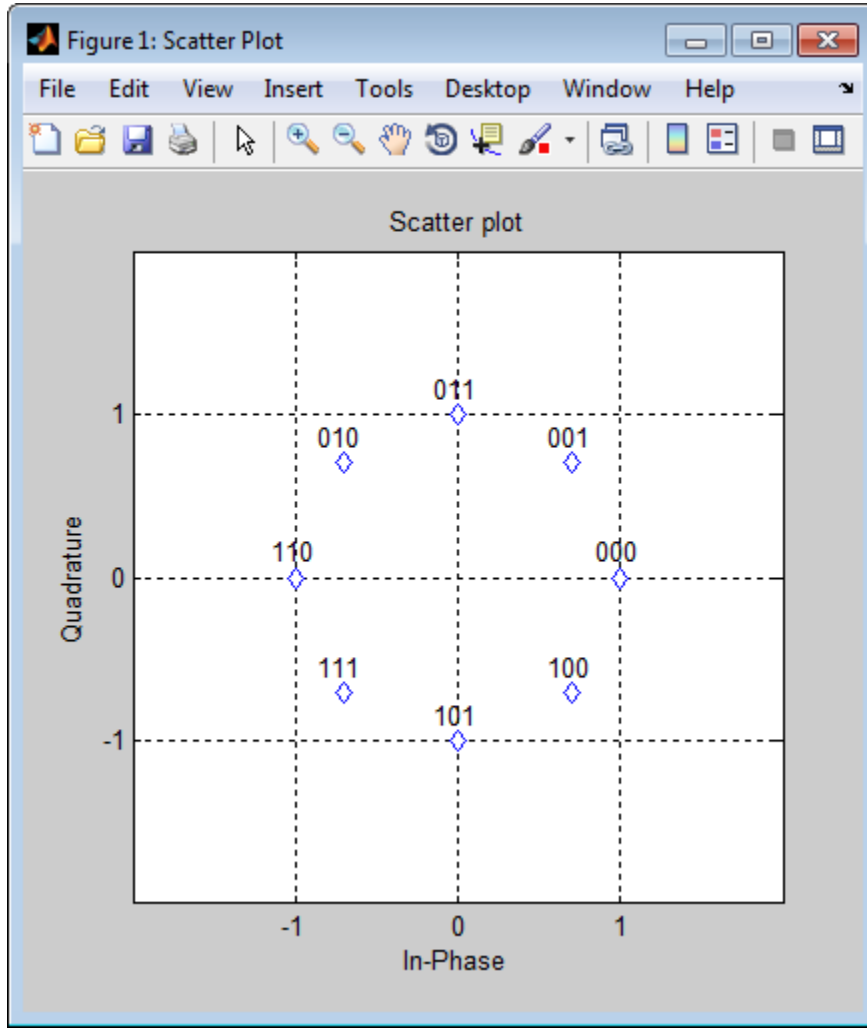
Gray Encoding a Modulated Signal

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

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

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

```
% Create 8-PSK Gray encoded modulator
hMod = comm.PSKModulator('ModulationOrder',8, ...
    'SymbolMapping','Gray','PhaseOffset',0);
% Create a scatter plot
constellation(hMod)
```



For modulation functions, set the symbol order argument to Gray.

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

Delays From 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 Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
FM demodulator	Sample-based processing	One output period
All demodulators in CPM sublibrary	Multirate processing, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking` D = **Traceback length** parameter	D+1 output periods
	Single-rate processing, D = **Traceback depth** parameter	D output periods
OQPSK demodulator	Single-rate processing	One output period
	Multirate processing, and the model uses a fixed-step solver with **Tasking Mode** parameter set to `Auto` or `MultiTasking`.	Two output periods
	Multirate processing processing, and the model uses a variable-step solver or the **Tasking Mode** parameter is set to Single`Tasking`.	One output period
All decoders in TCM sublibrary	**Operation mode** set to `Continuous`, Tr = **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.

Delays from Demodulation

For an example that illustrates delays from demodulation, see the “Delays from Demodulation” example.

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 2S in the scaling factors.

Process Upsampled Modulated Data (Except OQPSK Method)

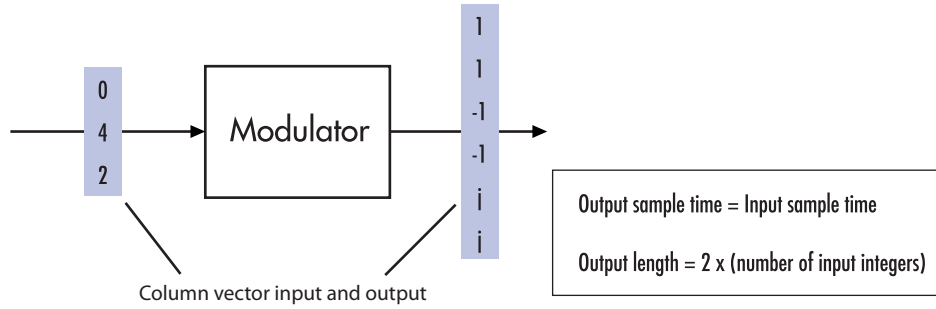
Computation Type	Input Status	Result
Modulation	Single-rate processing	Output vector length is S times the number of integers or binary words in the input vector. Output sample time equals the input sample time.
Modulation	Multirate processing	Output vector is a scalar. Output sample time is 1/S times the input sample time.
Demodulation	Single-rate processing	Number of integers or binary words in the output vector is 1/S times the number of samples in the input vector. Output sample time equals the input sample time.
Demodulation	Multirate processing	Output signal contains one integer or one binary word. Output sample time is S times the input sample time.

Computation Type	Input Status	Result
		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 = 1 or if the demodulator is from the CPM sublibrary.

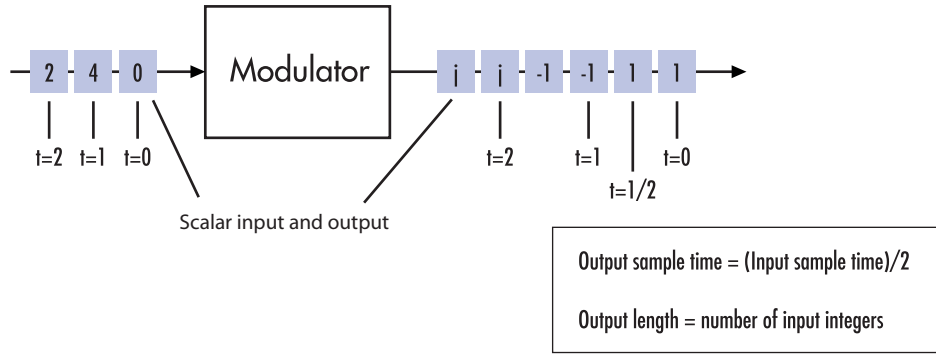
Illustrations of Size or Rate Changes

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

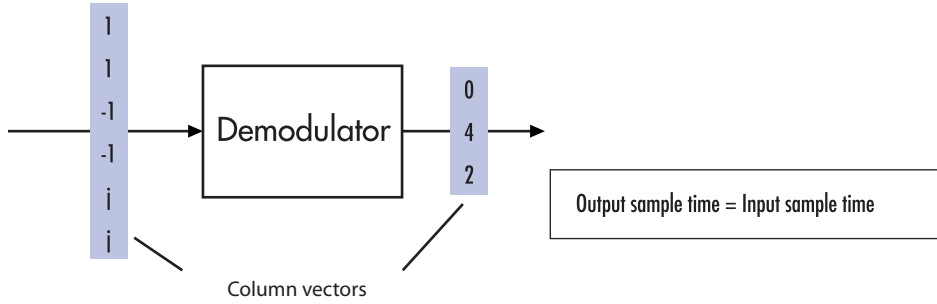


Upsample Output: Multirate Processing

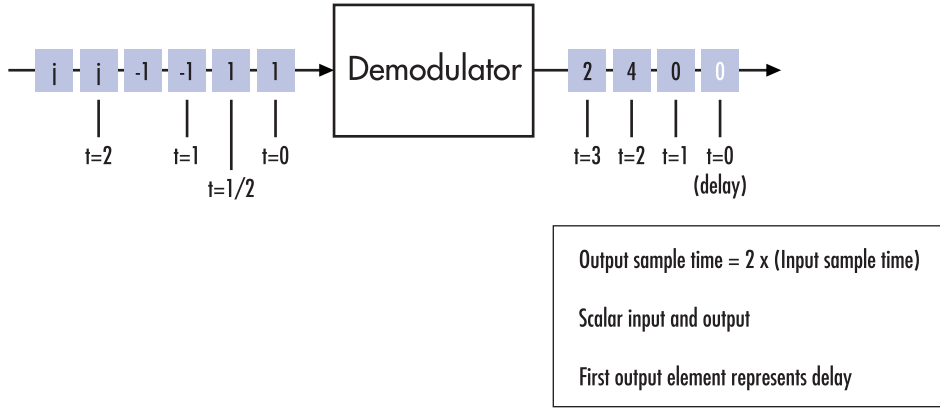


The following schematics illustrate how a demodulator (other than OQPSK or one from the CPM sublibrary) processes three doubly sampled symbols using both frame-based and sample-based inputs. In both cases, the **Samples per symbol** parameter is 2. The sample-based schematic includes an output delay of one sample period.

Upsampled Input: Single Rate Processing



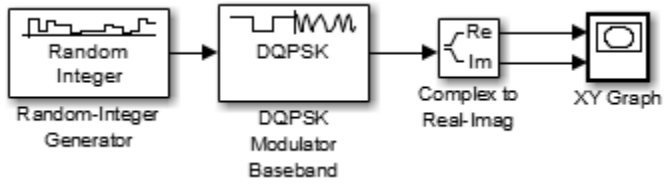
Upsampled Input: Multirate Processing



PM Modulation

DQPSK Signal Constellation Points and Transitions

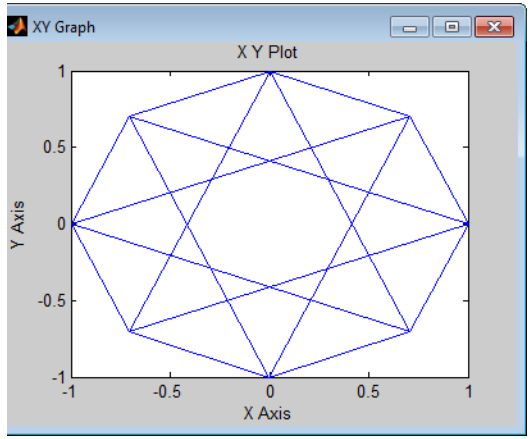
The model below plots the output of the DQPSK Modulator Baseband block. The image shows the possible transitions from each symbol in the DQPSK signal constellation to the next symbol.



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 `randseed` 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.

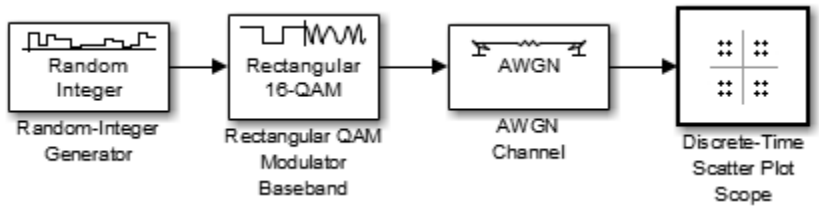


This plot illustrates π/4-DQPSK modulation, because the default **Phase offset** parameter in the DQPSK Modulator Baseband block is `pi/4`. To see how the phase offset influences the signal constellation, change the **Phase offset** parameter in the DQPSK Modulator Baseband block to `pi/8` or another value. Run the model again and observe how the plot changes.

AM Modulation

Rectangular QAM Modulation and Scatter Diagram

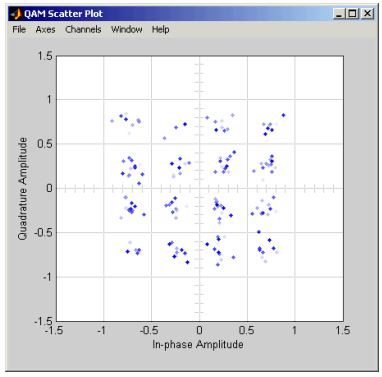
The model below uses the M-QAM Modulator Baseband block to modulate random data. After passing the symbols through a noisy channel, the model produces a scatter diagram of the noisy data. The diagram suggests what the underlying signal constellation looks like and shows that the noise distorts the modulated 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 `randseed` 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 in the figure. From the model window's **Simulation** menu, select **Model Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `250`. 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 the Symbol Error Rate

The example generates a random digital signal, modulates it, and adds noise. Then it creates a scatter plot, demodulates the noisy signal, and computes the symbol error rate.

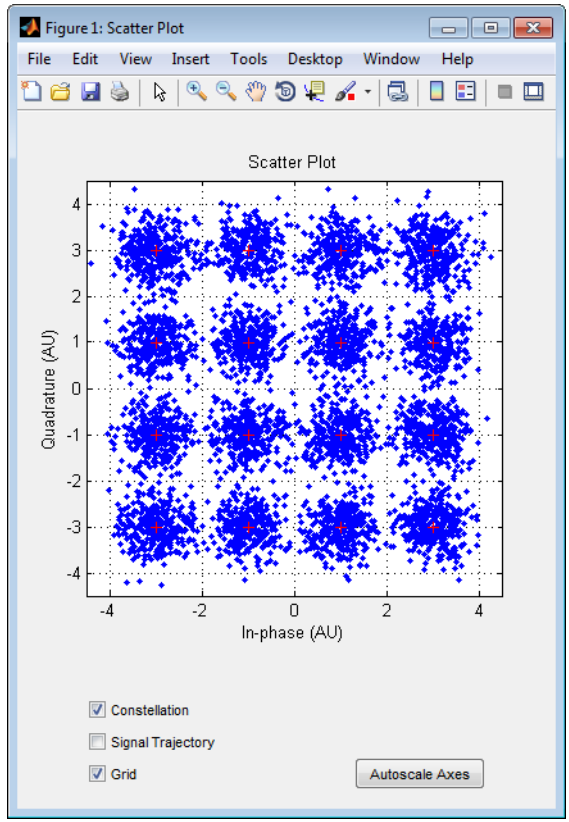
```
% Create a random digital message
M = 16;                      % Alphabet size
x = randi([0 M-1],5000,1);  % Random symbols

% Use 16-QAM modulation.
hMod = comm.RectangularQAMModulator('ModulationOrder',M);
hDemod = comm.RectangularQAMDemodulator('ModulationOrder',M);

% Create a constellation diagram object.
cpts = constellation(hMod);
hConst = comm.ConstellationDiagram('ReferenceConstellation',cpts, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);

% Apply 16-QAM modulation.
y = step(hMod,x);

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

% Create constellation diagram from noisy data.
step(hConst,ynoisy)

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

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

%%
% ================================================
% Documentation example from
% "Constellation for 16-PSK"
% in modulation.xml

% begindocexample 16psk_const
% Use 16-PSK modulation with no phase offset and binary symbol mapping.
hMod = comm.PSKModulator(16,0,'SymbolMapping','binary');

% Create a scatter plot
```

```
constellation(hMod)
% enddocexample 16psk_const

%%
% ================================================
% Documentation example from
% "Constellation for 32-QAM"
% in modulation.xml

% Example: Plotting Signal Constellations
% Constellation for 32-QAM

% Copyright 2003 The MathWorks, Inc.

close all;

% begindocexample 32qam_const
% Create 32-QAM modulator with binary symbol mapping
hMod = comm.RectangularQAMModulator(32,'SymbolMapping','binary');
% Create a scatter plot
constellation(hMod)
% enddocexample 32qam_const

doctouchupfigure(gcf,1);

%%
% ================================================
% Documentation example from
% "Gray-Coded Signal Constellation"
% in modulation.xml

% begindocexample graycoded_const
% Create 8-QAM Gray encoded modulator
hMod = comm.RectangularQAMModulator(8);
% Create a scatter plot
constellation(hMod)
% enddocexample graycoded_const
```

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

```
num =
```

```
    83

rt =

    0.0166
```



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

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".

Rectangular Pulse Shaping

Rectangular pulse shaping repeats each output from the modulator a fixed number of times to create an upsampled signal. Rectangular pulse shaping can be a first step or an exploratory step in algorithm development, though it is less realistic than other kinds of pulse shaping. If the transmitter upsamples the modulated signal, then the receiver should downsample the received signal before demodulating. The "integrate and dump" operation is one way to downsample the received signal.

The code below uses the `rectpulse` function for rectangular pulse shaping at the transmitter and the `intdump` function for downsampling at the receiver.

```
M = 16;                       % Alphabet size
x = randi([0 M-1],5000,1);  % Message signal
Nsamp = 4;                    % Oversampling rate

% Use 16-QAM modulation.
hMod = comm.RectangularQAMModulator;
hDemod = comm.RectangularQAMDemodulator;

% Modulate
y = step(hMod,x);

% Follow with rectangular pulse shaping.
ypulse = rectpulse(y,Nsamp);

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

% Downsample at the receiver.
ydownsamp = intdump(ynoisy,Nsamp);

% Demodulate to recover the message.
z = step(hDemod,ydownsamp);
```

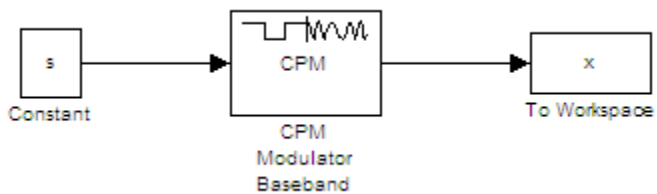
CPM Modulation

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 so that it uses 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.



The first step of this example is to build the model. To open the completed model, click here in the MATLAB Help browser. 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`.

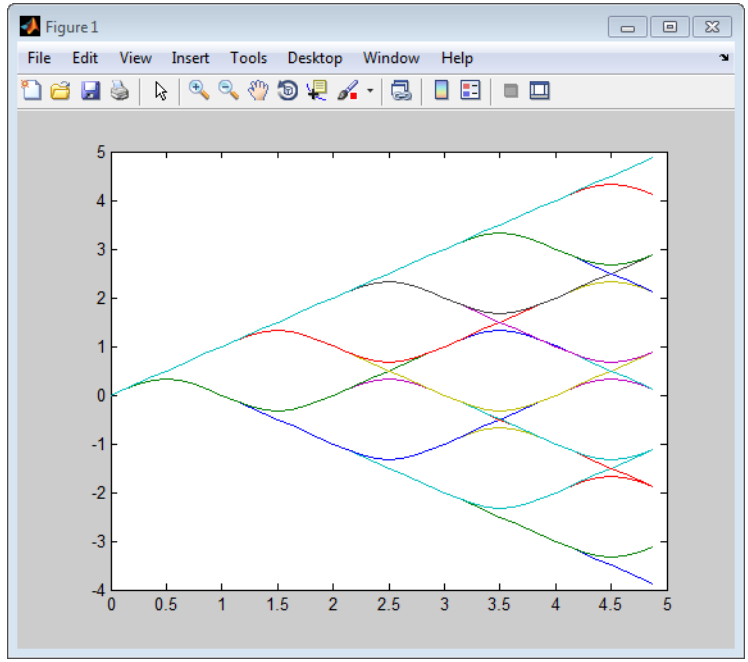
  - Set **Modulation index** to `2/3`.
  - Set **Frequency pulse shape** to `Raised Cosine`.
  - Set **Pulse length** to `2`.
- To Workspace, in the Simulink Sinks library
  - Set **Variable name** to `x`.
  - Set **Save format** to `Array`.

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;
modulation_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_phasetree', .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:

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

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

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

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

Variable	What the Variable Represents
r	Received signal with coordinates (x, y).
b	Transmitted bit (one of the K bits in an M-ary symbol, assuming all M symbols are equally probable.
S_0	Ideal symbols or constellation points with bit 0, at the given bit position.
S_1	Ideal symbols or constellation points with bit 1, at the given bit position.
s_x	In-phase coordinate of ideal symbol or constellation point.
s_y	Quadrature coordinate of ideal symbol or constellation point.
σ^2	Noise variance of baseband signal.
σ_x^2	Noise variance along in-phase axis.
σ_y^2	Noise variance along quadrature axis.

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

Approximate LLR Algorithm

Approximate LLR 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]:

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

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 Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
FM demodulator	Sample-based processing	One output period
All demodulators in CPM sublibrary	Multirate processing, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking` D = **Traceback length** parameter	D+1 output periods
	Single-rate processing, D = **Traceback depth** parameter	D output periods
OQPSK demodulator	Single-rate processing	One output period
	Multirate processing, and the model uses a fixed-step solver with **Tasking Mode** parameter set to `Auto` or `MultiTasking`.	Two output periods
	Multirate processing processing, and the model uses a variable-step solver or the **Tasking Mode** parameter is set to Single`Tasking`.	One output period

Modulation or Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
All decoders in TCM sublibrary	**Operation mode** set to `Continuous`, Tr = **Traceback depth** parameter, and code rate k/n	Tr*k output bits

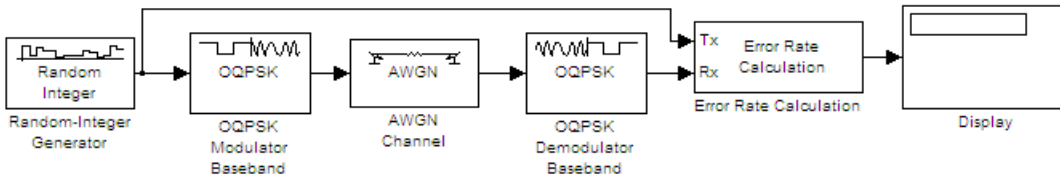
As a result of delays, data that enters a modulation or demodulation block at time T appears in the output at time T+delay. In particular, if your simulation computes error statistics or compares transmitted with received data, it must take the delay into account when performing such computations or 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 above. From the model window's **Simulation**, select **Model Configuration parameters**. In the **Configuration Parameters** dialog box, set **Stop time** to `1000`. 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, Prentice-Hall, 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...

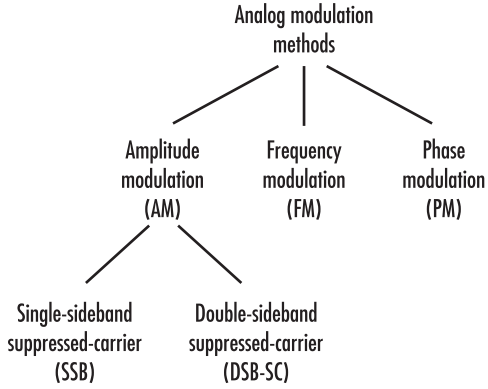
Analog Modulation

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 System 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 System 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.

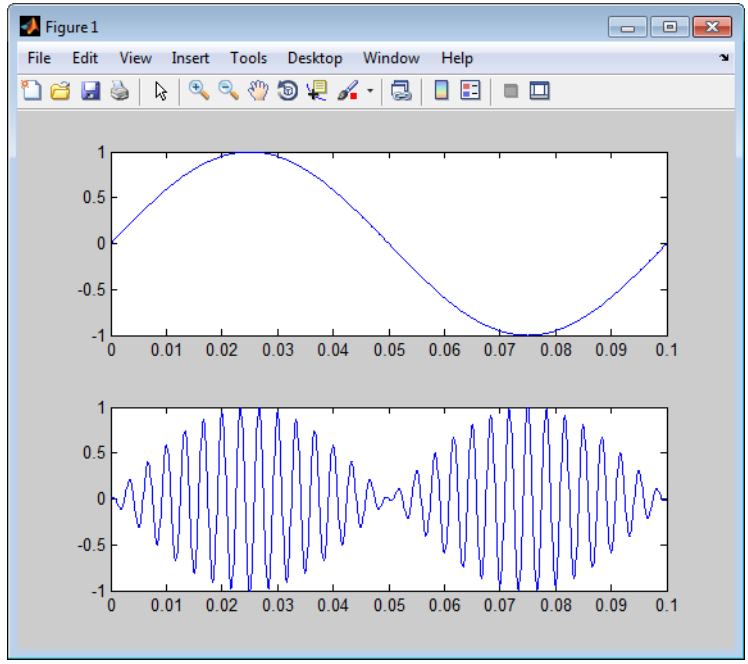
Representing Analog Signals Using MATLAB

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/`Fs`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

For example, if `t` measures time in seconds, then the vector `x` below is the result of sampling a sine wave 8000 times per second for 0.1 seconds. The vector `y` represents the modulated signal.

```
Fs = 8000; % Sampling rate is 8000 samples per second.
```

```
Fc = 300; % Carrier frequency in Hz
t = [0:.1*Fs]'/Fs; % Sampling times for .1 second
x = sin(20*pi*t); % Representation of the signal
y = ammod(x,Fc,Fs); % Modulate x to produce y.
figure;
subplot(2,1,1); plot(t,x); % Plot x on top.
subplot(2,1,2); plot(t,y)% Plot y below.
```



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

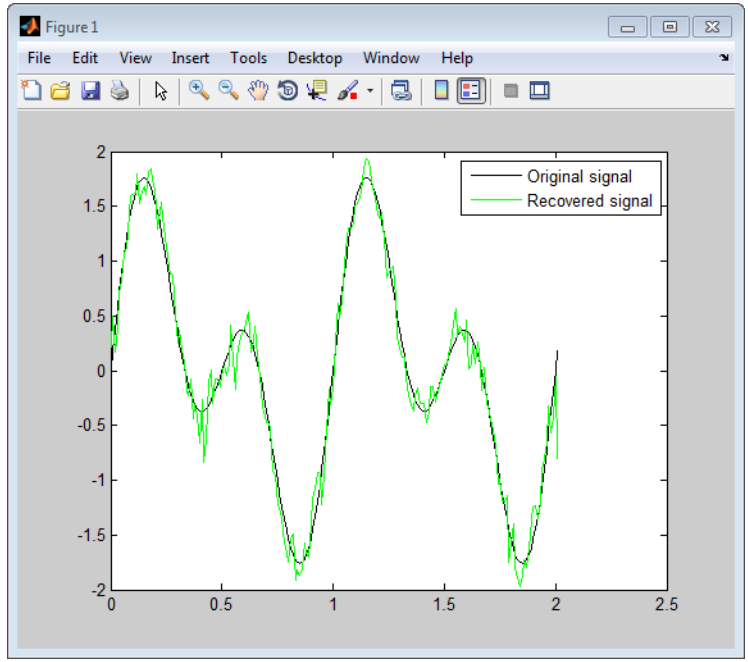
```
Fs = 8000;
t = [0:.1*Fs]'/Fs;
x = [sin(20*pi*t), sin(20*pi*t+pi/8)];
```

Analog Modulation 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+1]'/Fs; % Time points for sampling

% Create the signal, a sum of sinusoids.
x = sin(2*pi*t) + sin(4*pi*t);

Fc = 10; % Carrier frequency in modulation
phasedev = pi/2; % Phase deviation for phase modulation

y = pmmod(x,Fc,Fs,phasedev); % Modulate.
y = awgn(y,10,'measured',103); % Add noise.
z = pmdemod(y,Fc,Fs,phasedev); % Demodulate.

% Plot the original and recovered signals.
figure; plot(t,x,'k-',t,z,'g-');
legend('Original signal','Recovered signal');
```



Other examples using analog modulation functions appear in the reference pages for `ammod`, `amdemod`, `ssbdemod`, and `fmmod`.

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.

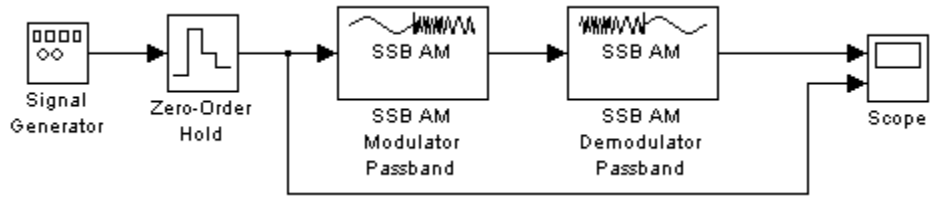
Filter Design Issues

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

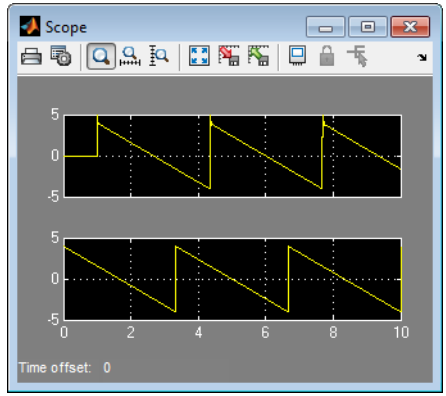
In many situations, a suitable cutoff frequency is half the carrier frequency. Because the carrier frequency must be higher than the bandwidth of the message signal, a cutoff frequency chosen in this way properly filters out unwanted frequency components. If the cutoff frequency is too high, the unwanted 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.

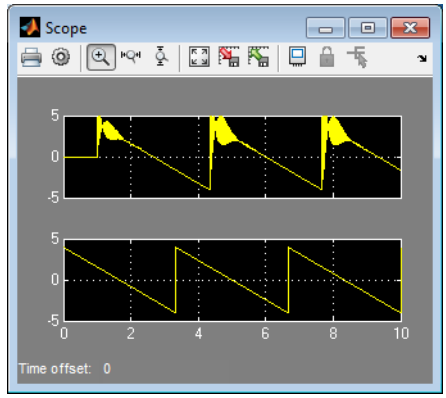
This example generates the following output:



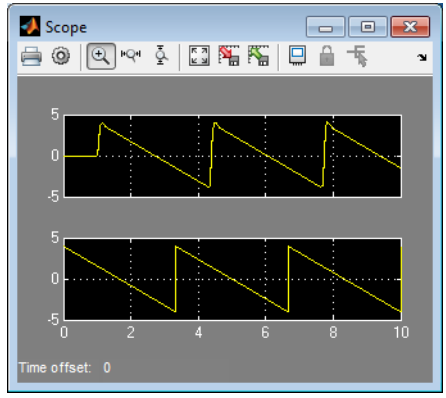
There is invariably a delay between a demodulated signal and the original received signal. Both the filter order and the filter parameters directly affect the length of this delay.

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.



Synchronization

In this section...

Synchronization Features

In order to interpret information correctly, a communication receiver must be synchronized with the corresponding transmitter. This can be achieved in both analog and digital domains. A digital receiver must sample the signal at an appropriate instant within the symbol period, and must estimate the carrier phase. Alternatively, analog components such as voltage-controlled oscillators (VCOs) and phase-locked loops (PLLs) can enable a receiver to adjust its behavior based on the parameters of the incoming signals or the desired signals.

This product implements several algorithms for timing phase recovery and carrier phase recovery. It also includes some lower-level components that you can use to build your own PLLs. This section describes the capabilities of the Synchronization library's blocks, in these key sections:

Open the Synchronization library by double-clicking its icon in the main Communications System Toolbox block library. Then open the sublibraries by double-clicking their icons in the Synchronization library.

Timing Phase Recovery

The Timing Phase Recovery library contains blocks that implement various algorithms for determining the best instant within a symbol period to sample a signal at the receiver. For example, the best instant for a PSK-modulated signal is at the peak of the pulse shape. Sampling at the best instant improves the receiver's performance on a noisy signal. Typically, you would place a timing phase recovery block after a receive filter that is matched to the transmitting pulse shape, and before a demodulator.

This section about timing phase recovery covers these topics:

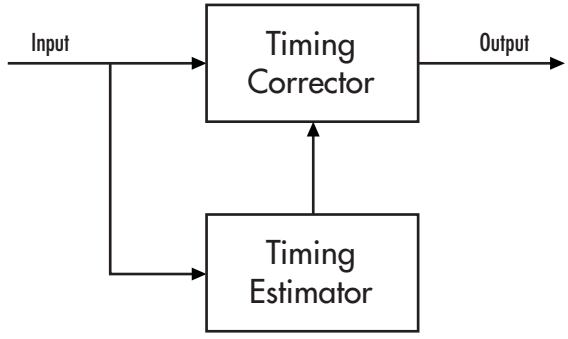
Supported Algorithms for Timing Phase Recovery

This library supports the algorithms listed below, which are all digital recovery methods rather than conventional analog phase-locked loops. For more information about each algorithm, see the reference works cited on each block's reference entry.

Algorithm	Block
Squaring method (feedforward)	Squaring Timing Recovery
Early-late gate method (feedback)	Early-Late Gate Timing Recovery
Gardner's method (feedback)	Gardner Timing Recovery
Fourth-order nonlinearity method (feedback)	MSK-Type Signal Timing Recovery
Mueller-Muller method (feedback)	Mueller-Muller Timing Recovery

Feedforward Method for Timing Phase Recovery

A feedforward method for timing phase recovery is structured as in the following figure.



In the figure,

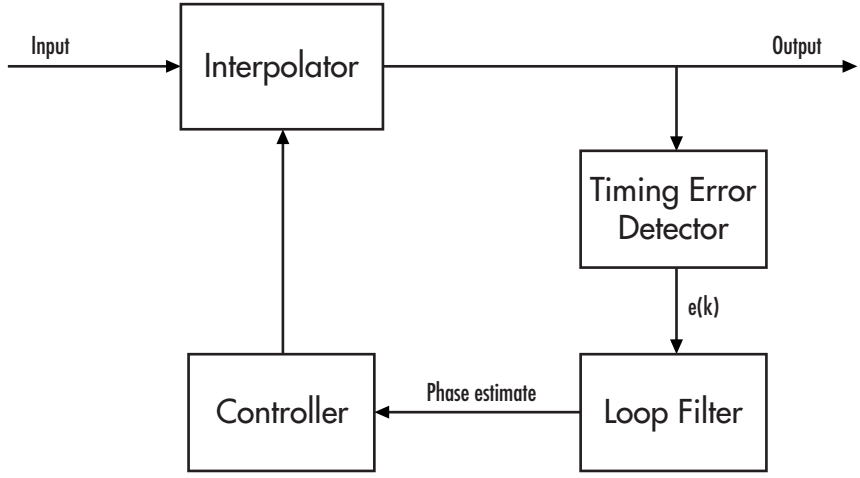
- The input signal is typically the output of a receive filter that is matched to the transmit pulse shape.
- The timing estimator gives an estimate of the input signal's sampling phase.
- The timing corrector is a sampler that outputs the value of the input signal corresponding to the phase estimate. The timing corrector interpolates between input signal values if necessary.

Squaring Timing Recovery block

The Squaring Timing Recovery block implements a feedforward method for timing phase recovery. In this method, the timing estimator uses a complex Fourier coefficient to determine the spectral component of the squared input signal at frequency 1/T, where T is the symbol period. For the specific equation, see the reference page for the Squaring Timing Recovery block.

Feedback Methods for Timing Phase Recovery

The Timing Phase Recovery library implements several feedback methods for timing phase recovery. A feedback method for timing phase recovery is structured as in the following figure.



In the figure,

- The input signal is typically the output of a receive filter that is matched to the transmit pulse shape.

- The interpolator generates additional samples based on the needs of the timing error detector. As implemented here, the interpolator uses linear interpolation between pairs of points.
- The timing error detector generates a timing error signal for each symbol. The algorithm used for timing error detection depends on the library block.
- The loop filter updates the phase estimate for the current symbol using the timing error signal and the previous symbol's phase estimate. The phase estimate for the (k +1)st symbol is $[[TAU]]_{k+1} = [[TAU]]_k+g*e(k)$, where g is the step size (also the **Error update gain** parameter in the feedback-method blocks in this library) and e(k) is the timing error for the kth symbol.
- The controller uses the phase estimates to determine the interpolating instants that the interpolator uses in the next cycle.

Restarting the Phase Estimating Process During the Simulation

When using a feedback method for timing phase recovery in Simulink, you can restart the phase-estimation process at different points during the simulation. Restarting the process means resetting the data buffer and phase-estimate buffer to the all-zeros state. The table below lists the supported options.

Value of Reset Parameter	When Estimation Process Restarts
`None`	At beginning of simulation only. During the simulation, the block operates continuously, retaining information from one symbol to the next.
`Every frame`	Regularly, at the start of each frame of data. During the simulation, each frame of data is processed independently. This option is valid only with frame-based data.
`On nonzero input via port`	Whenever the second input (`Rst`) is nonzero. When the first input is sample-based, its symbol period must equal the sample time of `Rst`. When the first input is frame-based, its frame period must equal the sample time of `Rst`, and the reset occurs at the start of the frame.

Using the Restarting Options Effectively

If you restart the phase-estimation process during the simulation, be sure to include enough symbols between successive resets for the algorithm to converge to a stable value.

Check the phase (`Ph`) output from the block to see whether its values stabilize before the reset occurs. To include more symbols between successive resets, either increase the frame size by buffering frames together (when using the `Every frame` option) or change the `Rst` input so that nonzero values occur less frequently.

Choosing a Method for Timing Phase Recovery

Depending on your system, one or more recovery methods implemented in this library might be suitable. If you use a method that is not suitable for your system, the results might not be accurate. This section discusses the assumptions and suitability of the various methods, covering these topics:

Squaring Timing Recovery Block

The Squaring Timing Recovery block recovers the symbol-timing phase of the input signal using a squaring method. This frame-based, feedforward, nondata-aided method is similar to a conventional squaring loop.

This block is suitable for systems that use linear baseband modulation types such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, and quadrature amplitude modulation (QAM).

The block assumes that the phase offset is constant for all symbols in the entire input frame. If necessary, you can use the Buffer block to reorganize your data into frames over which the phase offset can be assumed constant.

Assumptions Common to All Feedback Method Blocks

The feedback method, as implemented in this library, makes some assumptions about the data it receives:

- The phase varies slowly over time. Although the blocks compute a phase estimate for each symbol, the estimate should remain approximately constant for several symbols or else the algorithm does not converge.

- The symbol frequency is constant and known. Small variations in phase correspond to a frequency offset, but the blocks do not compensate for it. The blocks estimate and correct only the phase, not the frequency.

Although the blocks that implement feedback methods share a common structure and the common assumptions above, the blocks use different algorithms in the timing error detector and incur different delays. See each block's reference entry for details.

Early-Late Gate Timing Recovery Block

The Early-Late Gate Timing Recovery block implements a nondata-aided feedback method.

This block is suitable for systems that use a linear modulation type, such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), with Nyquist pulses (for example, using a raised cosine filter). In the presence of noise, the performance of this timing recovery method improves as the pulse's excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

The early-late gate method is similar to Gardner's method, which is implemented in the Gardner Timing Recovery block. Some differences between the two methods are as follows:

- In the ideal case (that is, when the phase estimate is zero and the input signal has symmetric Nyquist pulses), the timing error detector for the early-late gate method requires samples that span one symbol interval, rather than two symbol intervals as in Gardner's method.
- Compared to Gardner's method, the early-late gate method has higher self noise and thus does not perform as well as Gardner's method in systems with high SNR values.

Gardner Timing Recovery Block

The Gardner Timing Recovery block implements a nondata-aided feedback method that is independent of carrier phase recovery.

This block is suitable for both baseband systems and modulated carrier systems. More specifically, this block is suitable for systems that use a linear modulation type with Nyquist pulses that have an excess bandwidth between approximately 40% and 100%. Examples of suitable systems are those that use pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), and that shape the signal using raised cosine filters whose rolloff factor is between 0.4 and 1.

In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

Gardner's method is similar to the early-late gate method, which is implemented in the Early-Late Gate Timing Recovery block.

MSK-Type Signal Timing Recovery Block

The MSK-Type Signal Timing Recovery block recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general nondata-aided feedback method that is independent of carrier phase recovery but that requires prior compensation for the carrier frequency offset.

This block is suitable for systems that use baseband minimum shift keying (MSK) modulation or Gaussian minimum shift keying (GMSK) modulation. Unlike the other blocks in this library, this block does not require the input signal to have been filtered beforehand.

Mueller-Muller Timing Recovery Block

The Mueller-Muller Timing Recovery block implements a decision-directed, data-aided feedback method that requires prior recovery of the carrier phase.

This block is suitable for systems that use a *binary* linear modulation type, such as binary phase shift keying (BPSK) modulation, or binary phase amplitude modulation (BPAM). The binary requirement arises because the algorithm uses a sign detector (that is, a 1-bit quantizer) to arrive at decisions. When the input signal has Nyquist pulses (for example, using a raised cosine filter), this timing recovery method has no self noise. In the presence of noise, the performance of this timing recovery method improves as the pulse's excess bandwidth factor decreases, making the method a good candidate for narrowband signaling.

Examples of Timing Phase Recovery

One way to illustrate the usage and behavior of the timing phase recovery blocks is to introduce a fractional delay in a communications link and then see how well the block estimates the delay value and samples the received signal. In this context, a “fractional delay” is a delay that is not a multiple of the signal's symbol period. The examples discussed here are

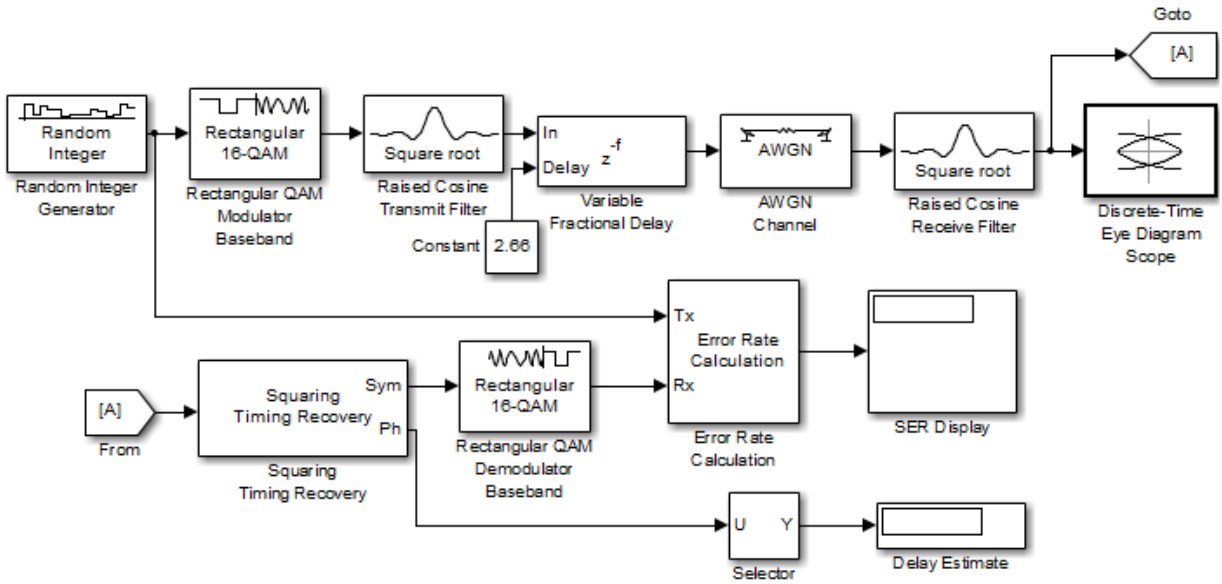
- “Squaring Timing Phase Recovery Example” on page 4-214, described below. This model introduces a fixed fractional delay and uses a feedforward method for timing phase recovery.

- Gardner timing phase recovery example, which you can open by entering `doc_gardner_phase_recovery` at the MATLAB command line. This model introduces a fractional delay that varies from frame to frame and uses a feedback method for timing phase recovery.

Squaring Timing Phase Recovery Example

This example modifies the one in "Design Raised Cosine Filters in Simulink" by introducing and then correcting for a fixed fractional delay. The model uses the Squaring Timing Recovery block to estimate that delay and determine the best instant within the symbol to sample its input signal. The model then demodulates the downsampled signal and computes a symbol error rate.

To open the completed model, click here in the MATLAB Help browser.



To build the model, first open the raised cosine filter model by clicking here in the MATLAB Help browser. Then, gather and configure these blocks:

- Variable Fractional Delay, in the DSP System Toolbox Signal Operations library. Use default parameters.
- Constant, in the Simulink Sources library

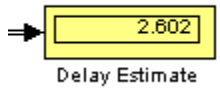
    - Set **Constant value** to `2.66`. This is the number of samples of delay introduced in the system.
- Goto and From, in the Simulink Signal Routing library. Use default parameters.
- Selector, in the Simulink Signal Routing library
    - Set **Elements** to `1`. This causes the block to select the first value in the frame, all of whose entries are actually the same.
    - Set **Input port width** to `100`.
- Squaring Timing Recovery
    - Set **Samples per symbol** to `8`.
- Rectangular QAM Demodulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
    - Set **Normalization method** to `Peak Power`.
    - Set **Peak power** to `1`.
- Error Rate Calculation, in the Comm Sinks library
    - Set **Receive delay** to `8`. This accounts for the delay of the pair of square root raised cosine filters.
    - Set **Output data** to `Port`.
- Two copies of Display, in the Simulink Sinks library. Make one tall enough to accommodate three values.

Connect the blocks as in the figure, and then run the simulation.

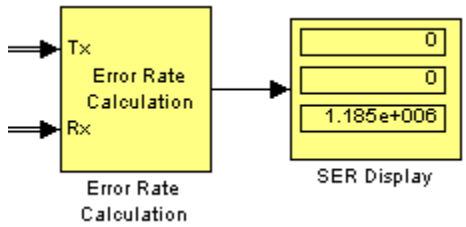
Results of the Simulation

When you run the simulation, look for these results:

- A delay estimate that varies during the simulation but is near the fixed value of 2.66. The Squaring Timing Recovery block computes this delay estimate for each frame and then uses it to choose a sampling instant for the symbols in that frame.



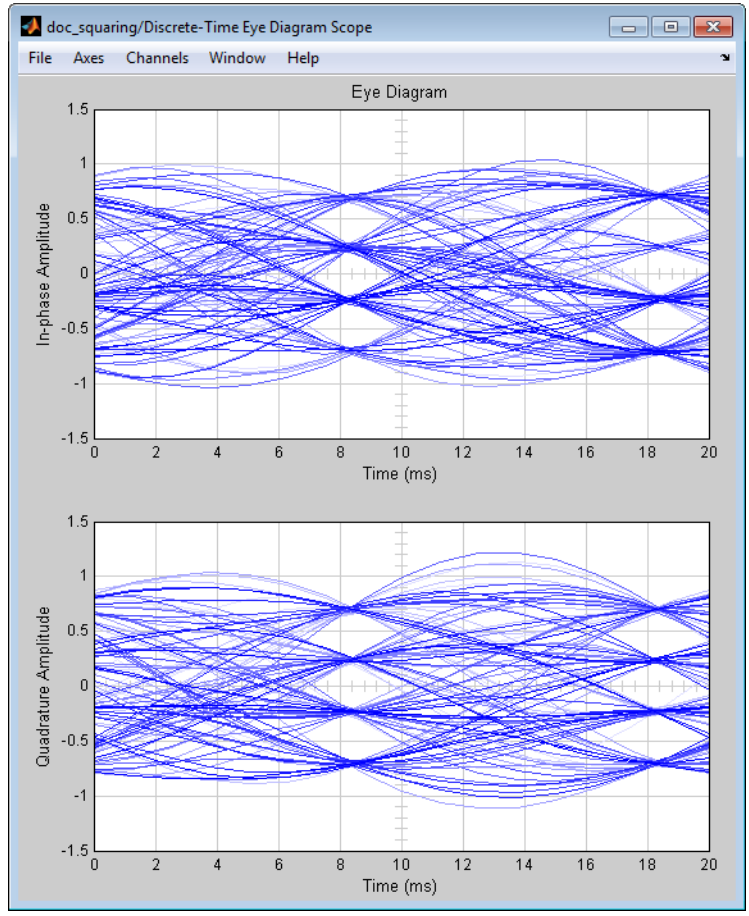
- A symbol error rate that is small or zero, depending on how long you run the simulation. For most or all symbols, the Squaring Timing Recovery block determines a sampling instant that enables the demodulator to recover data correctly.



- An eye diagram that has two widely opened “eyes” near 8.325 ms and 18.325 ms. These wide openings indicate appropriate instants at which to sample the filtered signal before demodulating, and reflect the introduced delay of 2.66 samples.

  To arrive at the numbers 8.325 and 18.325, reason as follows: The eye diagram displays two symbols per trace, and each symbol has a period of 10 ms. Without the introduced delay, the centers of the trace's two symbols are at 5 ms and 15 ms. The delay value in each symbol is
  (2.66 samples) / (8 samples/symbol) * (10 ms/symbol) = 3.325 ms

  Therefore, the traces from the delayed signal have their widest openings at (5+3.325) ms and (15+3.325) ms.



While this example uses a fixed delay throughout the simulation, the blocks in the timing recovery library can also correct for delays that vary (slowly) from symbol to symbol. For an example that uses a varying delay, see the Gardner timing phase recovery example.

Carrier Phase Recovery

The Carrier Phase Recovery library contains blocks that implement digital algorithms for determining the carrier phase of a baseband digital signal. The blocks assume that the carrier frequency is known and fixed. The blocks output the estimated carrier phase

as well as a corrected (that is, rotated) version of the input signal. Typically, you place a carrier phase recovery block before a demodulator, and after a timing phase recovery block or another block that produces symbols rather than an upsampled signal.

This section about carrier phase recovery covers these topics:

Supported Algorithms for Carrier Phase Recovery

This library supports the algorithms listed below, which are all digital recovery methods rather than conventional analog methods. For more information about each algorithm, see the reference works cited on each block's reference entry.

Algorithm	Block
2P-power method, suitable for full-response CPM, MSK, CPFSK, or GMSK signals.	CPM Phase Recovery
M-power method, suitable for M-PSK signals. (Also, the 4-power method is suitable for QAM signals using any alphabet size.)	M-PSK Phase Recovery

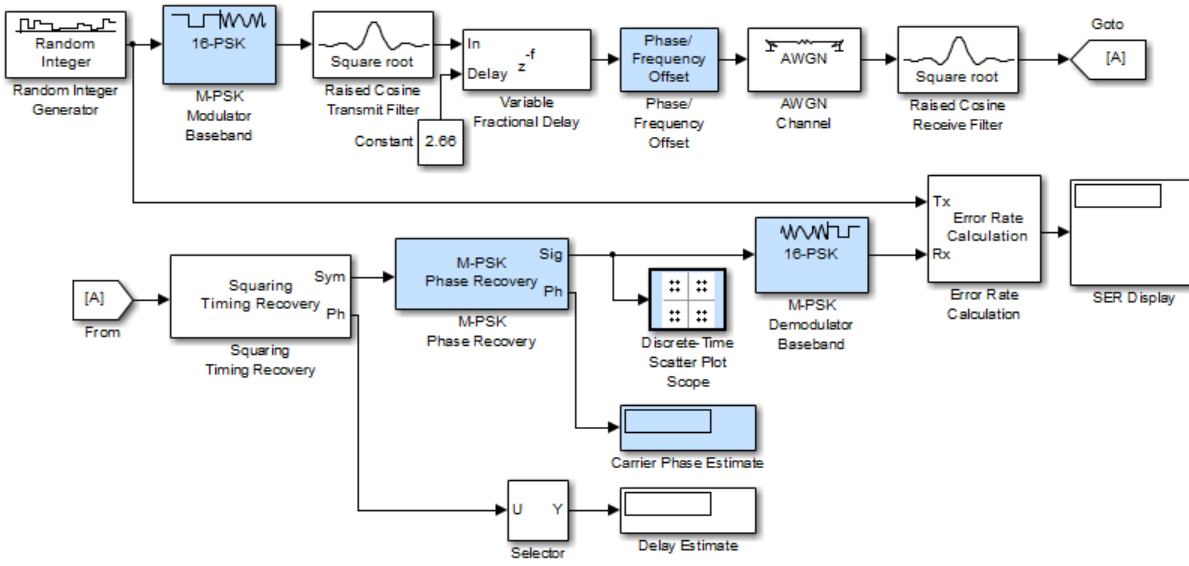
The methods described in the table are nondata-aided, clock-aided, feedforward methods. They assume that timing and carrier frequency are already known and any matched filtering has already been performed.

The methods also assume that the carrier phase to be estimated is constant over a series of consecutive symbols. When you use the blocks in this library, you specify the number of symbols over which the carrier phase is assumed constant.

Carrier Phase Recovery Example

This example modifies the "Squaring Timing Phase Recovery Example" on page 4-214 by introducing and then correcting for a fixed phase offset. The model uses the M-PSK Phase Recovery block to estimate the offset and correct the received baseband signal by rotating it. The model then demodulates the corrected signal and computes a symbol error rate.

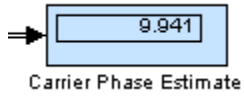
To open the model, enter `doc_carrier` at the MATLAB command line.



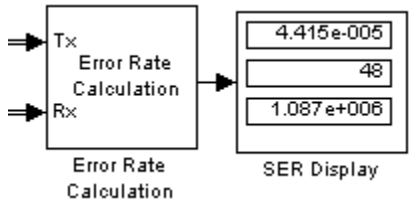
Results of the Simulation

When you run the simulation, look for these results:

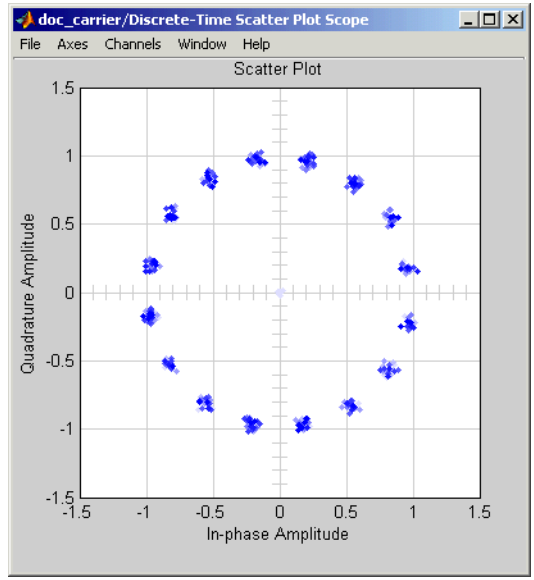
- A carrier phase estimate that varies during the simulation but is near the fixed value of 10 degrees. The M-PSK Phase Recovery block computes this carrier phase estimate for each frame and then uses it to correct the phase of the symbols in that frame.



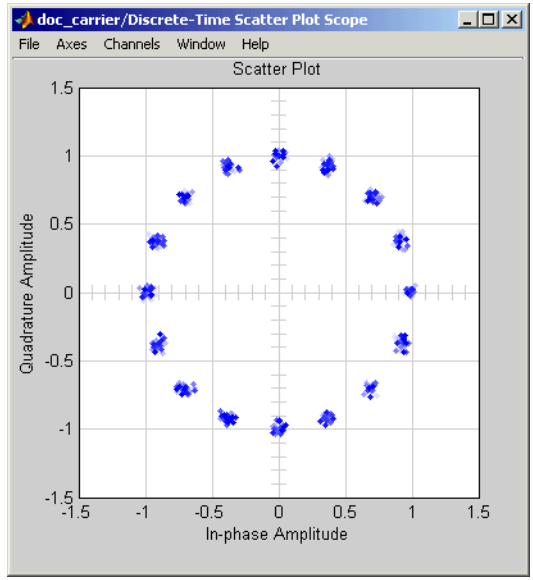
- A symbol error rate that is small or zero, depending on how long you run the simulation. For most or all symbols, the M-PSK Phase Recovery block enables the demodulator to recover data correctly.



- A signal constellation that reflects the signal whose phase the M-PSK Phase Recovery block has corrected. When you first begin the simulation and the block is in an initial latency period, the constellation reflects the phase offset of 10 degrees, with no correction. After the latency period is over, the constellation shows no phase offset because the M-PSK Phase Recovery block has corrected for it. The constellations before and after the end of the latency period appear below. The easiest way to see the 10-degree rotation between the two constellations is to look at the axes.



Before End of Latency Period

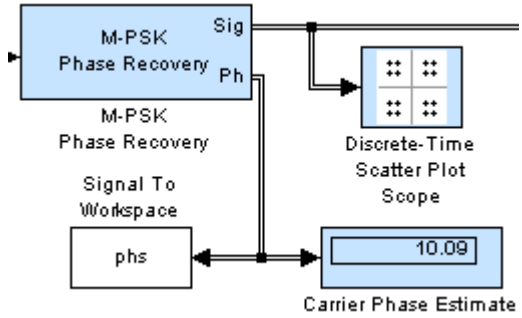


After End of Latency Period

Exploring the Simulation Further

Another way to examine the performance of the carrier phase recovery is to check how much the phase estimates from successive observation intervals differ from each other. You do this using the plotting capabilities of MATLAB along with the simulation capabilities of Simulink:

1. Add a To Workspace block, from the Sinks library in DSP System Toolbox, to the carrier phase recovery example model.
2. In the To Workspace block, set **Variable name** to `phs` and set **Limit data points to last** to `200`.
3. Connect the To Workspace block to the `Ph` output of the M-PSK Phase Recovery block, as shown in the following figure.



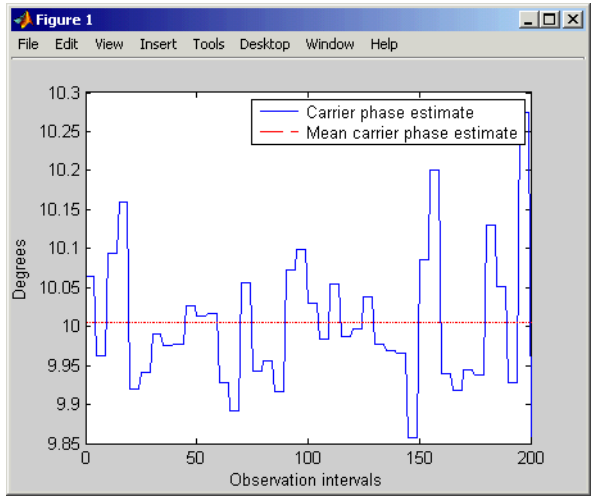
4 In the MATLAB Command Window, enter this command to run the simulation for a finite period of time:

```
sim('doc_carrier',205);
```

You make the simulation run faster by closing the window containing the signal constellation plot. When the simulation ends, the MATLAB workspace contains a variable called `phs` that contains the last 200 phase estimates from the M-PSK Phase Recovery block. Initial zeros from the delay period are omitted.

5 Create a plot showing the phase estimate values as well as their mean value by entering the following in the MATLAB Command Window:

```
plot(1:200,phs,'b-',1:200,mean(phs),'r--')
legend('Carrier phase estimate','Mean carrier phase estimate')
xlabel('Observation intervals'); ylabel('Degrees')
```



The plot shows that the mean is very close to the expected value of 10 degrees, while the individual phase estimates vary within an interval that includes 10 degrees.

Components

The Components sublibrary contains voltage-controlled oscillator (VCO) models as well as phase-locked loop (PLL) models.

This section discusses these topics:

For details about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" on page 4-225.

Voltage-Controlled Oscillator Blocks

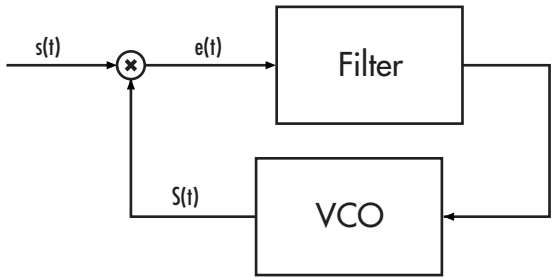
A voltage-controlled oscillator is one part of a phase-locked loop. The Continuous-Time VCO and Discrete-Time VCO blocks implement voltage-controlled oscillators. These blocks produce continuous-time and discrete-time output signals, respectively.

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 e(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

$$\sin\left(\Delta\theta(t)\right) \cong \Delta\theta(t)$$

to simplify the computations. This approximation is close when $\Delta\theta(t)$ 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.

Equalization

In this section...

Equalization Features

Time-dispersive channels can cause intersymbol interference (ISI), a form of distortion that causes symbols to overlap and become indistinguishable by the receiver. For example, in a multipath scattering environment, the receiver sees delayed versions of a symbol transmission, which can interfere with other symbol transmissions. An equalizer attempts to mitigate ISI and improve receiver performance. Communications System Toolbox provides equalization capabilities using one or more Simulink blocks, System objects, or MATLAB functions.

This product supports the following distinct classes of equalizers, each of which have a different overall structure:

- Linear equalizers, a class that is further divided into these categories:
  - Symbol-spaced equalizers
  - Fractionally spaced equalizers (FSEs)
- Decision-feedback equalizers (DFEs)
- MLSE (Maximum-Likelihood Sequence Estimation) equalizers that uses the Viterbi algorithm. To learn how to use the MLSE equalizer capabilities, see “MLSE Equalizers” on page 4-254.

Linear and decision-feedback equalizers are adaptive equalizers that use an adaptive algorithm when operating. For each of the adaptive equalizer classes listed above, this toolbox supports these adaptive algorithms:

- Least mean square (LMS)
- Signed LMS, including these types: sign LMS, signed regressor LMS, and sign-sign LMS
- Normalized LMS
- Variable-step-size LMS
- Recursive least squares (RLS)
- Constant modulus algorithm (CMA)

Several blocks from the Equalizers library implement adaptive equalizers, differing in the equalizer structure and the type of adaptive algorithm that they use. In all cases, you specify information about the equalizer structure (such as the number of taps), the adaptive algorithm (such as the step size), and the signal constellation used by the modulator in your model. You also specify an initial set of weights for the taps of the equalizer; the block adaptively updates the weights throughout the simulation. For adaptive algorithms other than CMA, the equalizer can adapt the weights in two modes: training mode and decision-directed mode.

To learn how to use the adaptive equalizer capabilities, start with "Adaptive Algorithms" on page 4-237. For more detailed background material, see the works listed in "Selected Bibliography for Equalizers".

Equalize A Signal

Equalizing a signal using Communications System Toolbox software involves these steps:

1. Create an equalizer object that describes the equalizer class and the adaptive algorithm that you want to use. An equalizer object is a type of MATLAB variable that contains information about the equalizer, such as the name of the equalizer class, the name of the adaptive algorithm, and the values of the weights.
2. Adjust properties of the equalizer object, if necessary, to tailor it to your needs. For example, you can change the number of weights or the values of the weights.
3. Apply the equalizer object to the signal you want to equalize, using the `equalize` method of the equalizer object.

Equalize a Signal Using MATLAB

This code briefly illustrates the steps in the basic procedure above.

```
% Build a set of test data.
hMod = comm.BPSKModulator; % BPSKModulator System object
x = step(hMod,randi([0 1],1000,1)); % BPSK symbols
rxsig = conv(x,[1 0.8 0.3]);  % Received signal
% Create an equalizer object.
eqlms = lineareq(8,lms(0.03));
% Change the reference tap index in the equalizer.
eqlms.RefTap = 4;
% Apply the equalizer object to a signal.
y = equalize(eqlms,rxsig,x(1:200));
```

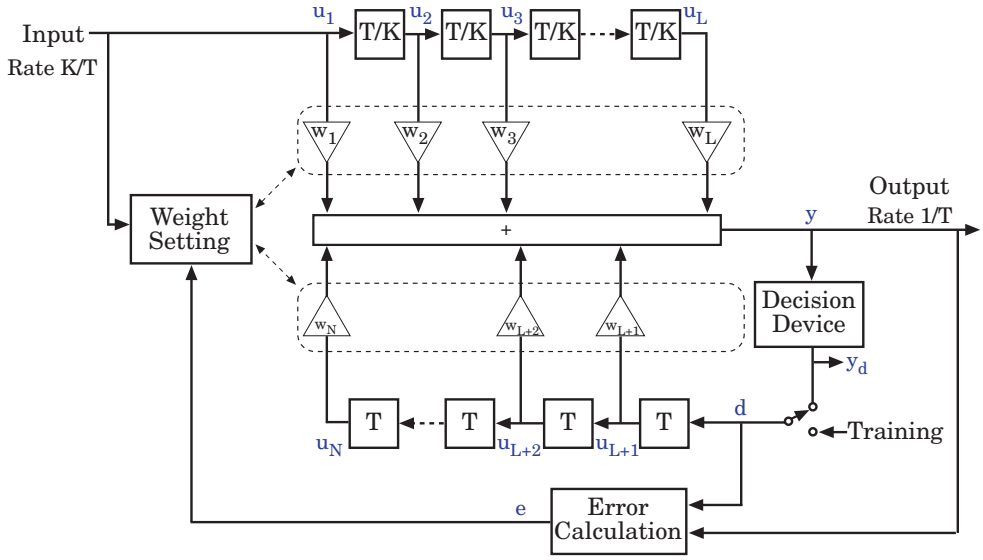
In this example, `eqlms` is an equalizer object that describes a linear LMS equalizer having eight weights and a step size of 0.03. At first, the reference tap index in the equalizer has a default value, but assigning a new value to the property `eqlms.RefTap` changes this index. Finally, the `equalize` command uses the `eqlms` object to equalize the signal `rxsig` using the training sequence `x(1:200)`.

Equalizer Structure

Decision-Feedback Equalizers

A decision-feedback equalizer is a nonlinear equalizer that contains a forward filter and a feedback filter. The forward filter is similar to the linear equalizer described in "Symbol-Spaced Equalizers" on page 4-230, while the feedback filter contains a tapped delay line whose inputs are the decisions made on the equalized signal. The purpose of a DFE is to cancel intersymbol interference while minimizing noise enhancement. By contrast, noise enhancement is a typical problem with the linear equalizers described earlier.

The following schematic contains a fractionally spaced DFE with L forward weights and N-L feedback weights. The forward filter is at the top and the feedback filter is at the bottom. If K is 1, the result is a symbol-spaced DFE instead of a fractionally spaced DFE.



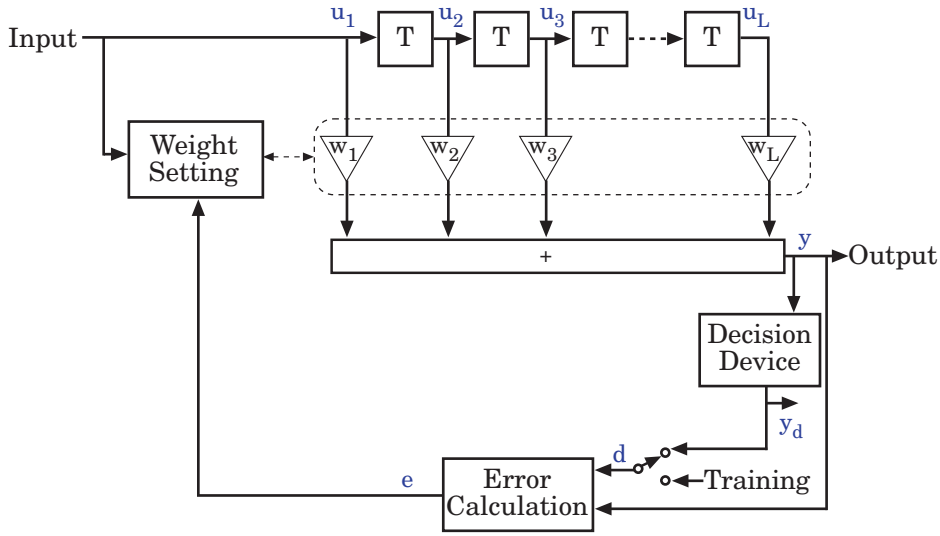
In each symbol period, the equalizer receives K input samples at the forward filter, as well as one decision or training sample at the feedback filter. The equalizer then outputs a weighted sum of the values in the forward and feedback delay lines, and updates the weights to prepare for the next symbol period.

Note: The algorithm for the Weight Setting block in the schematic *jointly* optimizes the forward and feedback weights. Joint optimization is especially important for the RLS algorithm.

Symbol-Spaced Equalizers

A symbol-spaced linear equalizer consists of a tapped delay line that stores samples from the input signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period. This class of equalizer is called *symbol-spaced* because the sample rates of the input and output are equal.

Below is a schematic of a symbol-spaced linear equalizer with N weights, where the symbol period is T.



Updating the Set of Weights

The algorithms for the Weight Setting and Error Calculation blocks in the schematic are determined by the adaptive algorithm chosen from the list in "Equalization Features" on page 4-227. The new set of weights depends on these quantities:

- The current set of weights
- The input signal
- The output signal
- For adaptive algorithms other than CMA, a reference signal, d, whose characteristics depend on the operation mode of the equalizer

Reference Signal and Operation Modes

The table below briefly describes the nature of the reference signal for each of the two operation modes.

Operation Mode of Equalizer	Reference Signal
Training mode	Preset known transmitted sequence
Decision-directed mode	Detected version of the output signal, denoted by y_d in the schematic

In typical applications, the equalizer begins in training mode to gather information about the channel, and later switches to decision-directed mode.

Error Calculation

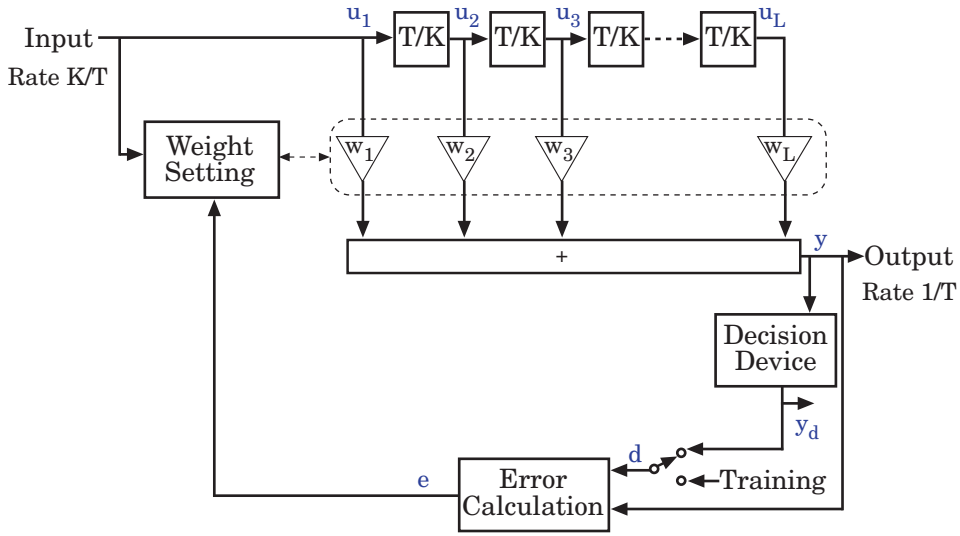
The error calculation operation produces a signal given by the expression below, where R is a constant related to the signal constellation.

$$e = \begin{cases} d - y & \text{Algorithms other than CMA} \\ y(R - |y|^2) & \text{CMA} \end{cases}$$

Fractionally Spaced Equalizers

A fractionally spaced equalizer is a linear equalizer that is similar to a symbol-spaced linear equalizer, as described in "Symbol-Spaced Equalizers" on page 4-230. By contrast, however, a fractionally spaced equalizer receives K input samples before it produces one output sample and updates the weights, where K is an integer. In many applications, K is 2. The output sample rate is 1/T, while the input sample rate is K/T. The weight-updating occurs at the output rate, which is the slower rate.

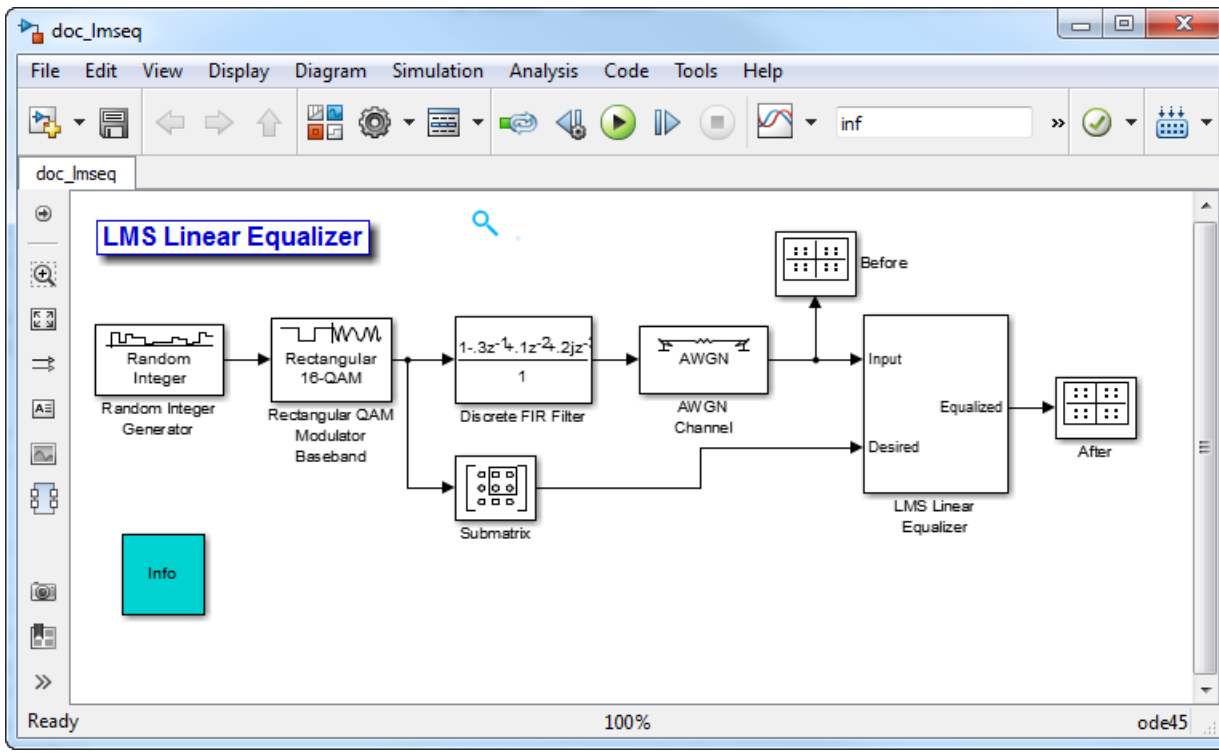
The following schematic illustrates a fractionally spaced equalizer.



Implement LMS Linear Equalizer Using Simulink

This example illustrates the use of an LMS linear equalizer. The simulation transmits a 16-QAM signal, modeling the channel using an FIR filter followed by additive white Gaussian noise. The equalizer receives the signal from the channel and, as training symbols, a subset of the modulator's output. The equalizer operates in training mode at the beginning of each frame and switches to decision-directed mode when it runs out of training symbols. The example contrasts the signals before and after equalization to illustrate the effect of the equalizer.

To open this model, enter `doc_lmseq` at the MATLAB command line.



To build the model, gather and configure these blocks:

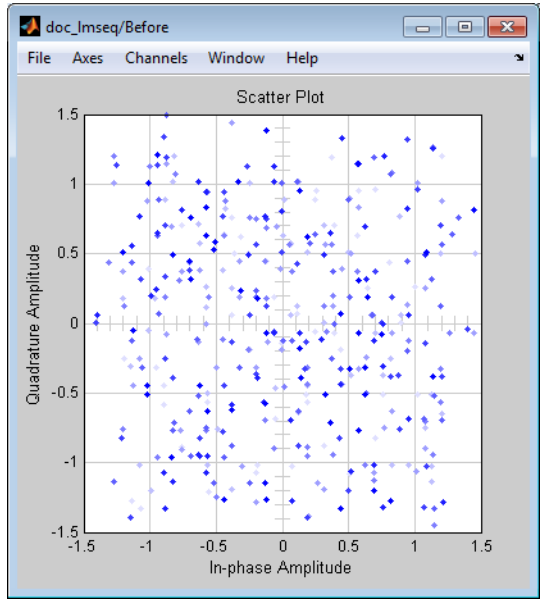
- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to 16.

  - Set **Sample time** to `1/1000`.
  - Select **Frame-based outputs**.
  - Set **Samples per frame** to `1000`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Average Power`.
  - Set **Average power** to `1`.
- Discrete FIR Filter, in the DSP System Toolbox Filter Implementations sublibrary of Filtering
  - Set **Filter structure** to `Direct form transposed`.
  - Set **Coefficients** to `[1 -.3 .1 .2j]`.
- Submatrix, in the DSP System Toolbox Indexing sublibrary of Signal Management
  - Set **Ending row** to `Index`.
  - Set **Ending row index** to `100`.
- AWGN Channel, in the Channels library
  - Set **Mode** to `Signal to noise ratio (SNR)`.
  - Set **SNR** to `40`.
- LMS Linear Equalizer
  - Set **Number of taps** to `6`.
  - Clear the **Mode input port**, **Output error**, and **Output weights** check boxes.
- Two copies of Constellation Diagram, in the Comm Sinks library
  - Set **Symbols to display** to `400` in each of the two copies.

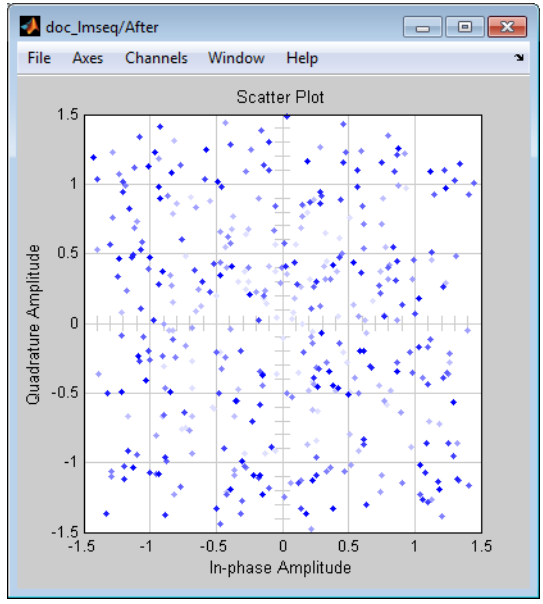
Connect the blocks as in the figure. Running the simulation produces two scatter plots that display the signal before and after equalization, respectively.

Scatter Plots in the Example

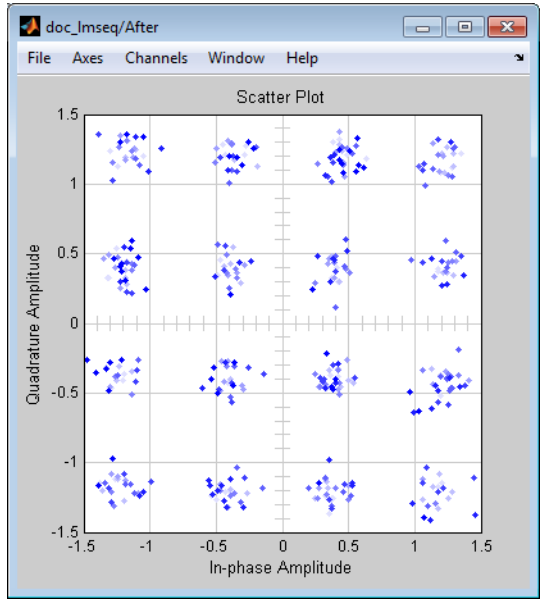
Throughout the simulation, the signal before equalization deviates noticeably from a 16-QAM signal constellation, as below.



Early in the simulation, the equalizer does not appear to improve the scatter plot. In fact, the equalizer is busy trying to adapt its weights appropriately. The following figure shows the equalized signal very early in the simulation.



After some simulation time passes, the equalizer's weights work well on the received signal. As a result, the equalized signal looks far more like a 16-QAM signal constellation than the received signal does. The figure below shows the equalized signal in its steady state.



Adaptive Algorithms

This section provides an overview of the process you typically use in the MATLAB environment to take advantage of the adaptive equalizer capabilities. The MLSE equalizer has a different interface, described in "MLSE Equalizers" on page 4-254.

Adaptive Equalizer Functions

Keeping the basic procedure in mind, read other portions of this chapter to learn more details about

- How to create objects that represent different classes of adaptive equalizers and different adaptive algorithms
- How to adjust properties of an adaptive equalizer or properties of an adaptive algorithm

- How to equalize signals using an adaptive equalizer object

Specify an Adaptive Algorithm

Choose an Adaptive Algorithm

Configuring an equalizer involves choosing an adaptive algorithm and indicating your choice when creating an equalizer object in the MATLAB environment.

Although the best choice of adaptive algorithm might depend on your individual situation, here are some generalizations that might influence your choice:

- The LMS algorithm executes quickly but converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, but its complexity grows with the square of the number of weights, roughly speaking. This algorithm can also be unstable when the number of weights is large.
- The various types of signed LMS algorithms simplify hardware implementation.
- The normalized LMS and variable-step-size LMS algorithms are more robust to variability of the input signal's statistics (such as power).
- The constant modulus algorithm is useful when no training signal is available, and works best for constant modulus modulations such as PSK.

  However, if CMA has no additional side information, it can introduce phase ambiguity. For example, CMA might find weights that produce a perfect QPSK constellation but might introduce a phase rotation of 90, 180, or 270 degrees. Alternatively, differential modulation can be used to avoid phase ambiguity.

Details about the adaptive algorithms are in the references listed in "Selected Bibliography for Equalizers".

Indicating a Choice of Adaptive Algorithm

After you have chosen the adaptive algorithm you want to use, indicate your choice when creating the equalizer object mentioned in "Equalize A Signal". The functions listed in the table below provide a way to indicate your choice of adaptive algorithm.

Adaptive Algorithm Function	Type of Adaptive Algorithm
`lms`	Least mean square (LMS)
`signlms`	Signed LMS, signed regressor LMS, sign-sign LMS
`normlms`	Normalized LMS
`varlms`	Variable-step-size LMS
`rls`	Recursive least squares (RLS)
`cma`	Constant modulus algorithm (CMA)

Two typical ways to use a function from the table are as follows:

- Use the function in an inline expression when creating the equalizer object.

  For example, the code below uses the `lms` function inline when creating an equalizer object.

  ```
  eqlms = lineareq(10,lms(0.003));
  ```

- Use the function to create a variable in the MATLAB workspace and then use that variable when creating the equalizer object. The variable is called an *adaptive algorithm object*.

  For example, the code below creates an adaptive algorithm object named `alg` that represents the adaptive algorithm, and then uses `alg` when creating an equalizer object.

  ```
  alg = lms(0.003);
  eqlms = lineareq(10,alg);
  ```

  Note: To create an adaptive algorithm object by duplicating an existing one and then changing its properties, see the important note in "Duplicating and Copying Objects" on page 4-241 about the use of `copy` versus the `=` operator.

In practice, the two ways are equivalent when your goal is to create an equalizer object or to equalize a signal.

Access Properties of an Adaptive Algorithm

The adaptive algorithm functions not only provide a way to indicate your choice of adaptive algorithm, but they also let you specify certain properties of the algorithm. For

information about what each property of an adaptive algorithm object means, see the reference pages for the `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma` functions.

To view or change any properties of an adaptive algorithm, use the syntax described for channel objects in "Display Object Properties" and "Change Object Properties".

Specify an Adaptive Equalizer

Defining an Equalizer Object

To create an equalizer object, use one of the functions listed in the table below.

Function	Type of Equalizer
`lineareq`	Linear equalizer (symbol-spaced or fractionally spaced)
`dfe`	Decision-feedback equalizer

For example, the code below creates three equalizer objects: one representing a symbol-spaced linear RLS equalizer having 10 weights, one representing a fractionally spaced linear RLS equalizer having 10 weights and two samples per symbol, and one representing a decision-feedback RLS equalizer having three weights in the feedforward filter and two weights in the feedback filter.

```
% Create equalizer objects of different types.
eqlin = lineareq(10,rls(0.3)); % Symbol-spaced linear
eqfrac = lineareq(10,rls(0.3),[-1 1],2); % Fractionally spaced linear
eqdfe = dfe(3,2,rls(0.3)); % DFE
```

Although the `lineareq` and `dfe` functions have different syntaxes, they both require an input argument that represents an adaptive algorithm. To learn how to represent an adaptive algorithm or how to vary properties of the adaptive algorithm, see "Specify an Adaptive Algorithm" on page 4-238.

Each of the equalizer objects created above is a valid input argument for the `equalize` function. To learn how to use the `equalize` function to equalize a signal, see "Using Adaptive Equalizers" on page 4-243.

Duplicating and Copying Objects

Another way to create an object is to duplicate an existing object and then adjust the properties of the new object, if necessary. If you do this, it is important that you use a `copy` command such as

```
c2 = copy(c1); % Copy c1 to create an independent c2.
```

instead of `c2 = c1`. The `copy` command creates a copy of `c1` that is independent of `c1`. By contrast, the command `c2 = c1` creates `c2` as merely a reference to `c1`, so that `c1` and `c2` always have indistinguishable content.

Accessing Properties of an Equalizer

An equalizer object has numerous properties that record information about the equalizer. Properties can be related to

- The structure of the equalizer (for example, the number of weights).
- The adaptive algorithm that the equalizer uses (for example, the step size in the LMS algorithm). When you create the equalizer object using `lineareq` or `dfe`, the function copies certain properties from the algorithm object to the equalizer object. However, the equalizer object does not retain a connection to the algorithm object.
- Information about the equalizer's current state (for example, the values of the weights). The `equalize` function automatically updates these properties when it operates on a signal.
- Instructions for operating on a signal (for example, whether the equalizer should reset itself before starting the equalization process).

For information about what each equalizer property means, see the reference page for the `lineareq` or `dfe` function.

To view or change any properties of an equalizer object, use the syntax described for channel objects in “Display Object Properties” and “Change Object Properties”.

Linked Properties of an Equalizer Object

Some properties of an equalizer object are related to each other such that when one property's value changes, another property's value must adjust, or else the equalizer object fails to describe a valid equalizer. For example, in a linear equalizer, the `nWeights` property is the number of weights, while the `Weights` property is the value of the weights. If you change the value of `nWeights`, the value of `Weights` must adjust so that its vector length is the new value of `nWeights`.

To find out which properties are related and how MATLAB compensates automatically when you make certain changes in property values, see the reference page for `lineareq` or `dfe`.

The example below illustrates that when you change the value of `nWeights`, MATLAB automatically changes the values of `Weights` and `WeightInputs` to make their vector lengths consistent with the new value of `nWeights`. Because the example uses the variable-step-size LMS algorithm, `StepSize` is a vector (not a scalar) and MATLAB changes its vector length to maintain consistency with the new value of `nWeights`.

```
eqlvar = lineareq(10,varlms(0.01,0.01,0,1)) % Create equalizer object.
eqlvar.nWeights = 8 % Change the number of weights from 10 to 8.
% MATLAB automatically changes the sizes of eqlvar.Weights and
% eqlvar.WeightInputs.
```

The output below displays all the properties of the equalizer object before and after the change in the value of the `nWeights` property. In the second listing of properties, the `nWeights`, `Weights`, `WeightInputs`, and `StepSize` properties all have different values compared to the first listing of properties.

```
eqlvar =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Variable Step Size LMS'
                nWeights: 10
              nSampPerSym: 1
                   RefTap: 1
                SigConst: [-1 1]
                InitStep: 0.0100
                 IncStep: 0.0100
                 MinStep: 0
                 MaxStep: 1
           LeakageFactor: 1
                StepSize: [1x10 double]
                 Weights: [0 0 0 0 0 0 0 0 0 0]
            WeightInputs: [0 0 0 0 0 0 0 0 0 0]
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0


eqlvar =

                  EqType: 'Linear Equalizer'
                 AlgType: 'Variable Step Size LMS'
```

```
              nWeights: 8
            nSampPerSym: 1
                 RefTap: 1
              SigConst: [-1 1]
              InitStep: 0.0100
               IncStep: 0.0100
               MinStep: 0
               MaxStep: 1
         LeakageFactor: 1
              StepSize: [1x8 double]
               Weights: [0 0 0 0 0 0 0 0]
          WeightInputs: [0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
   NumSamplesProcessed: 0
```

Using Adaptive Equalizers

Equalize Using a Training Sequence in MATLAB

This section describes how to equalize a signal by using the `equalize` function to apply an adaptive equalizer object to the signal. The `equalize` function also updates the equalizer. This section assumes that you have already created an adaptive equalizer object, as described in "Specify an Adaptive Equalizer" on page 4-240.

For an example that complements this section, see the Adaptive Equalization Simulation.

In typical applications, an equalizer begins by using a known sequence of transmitted symbols when adapting the equalizer weights. The known sequence, called a *training sequence*, enables the equalizer to gather information about the channel characteristics. After the equalizer finishes processing the training sequence, it adapts the equalizer weights in decision-directed mode using a detected version of the output signal. To use a training sequence when invoking the `equalize` function, include the symbols of the training sequence as an input vector.

Note: As an exception, CMA equalizers do not use a training sequence. If an equalizer object is based on CMA, you should not include a training sequence as an input vector.

The following code illustrates how to use `equalize` with a training sequence. The training sequence in this case is just the beginning of the transmitted message.

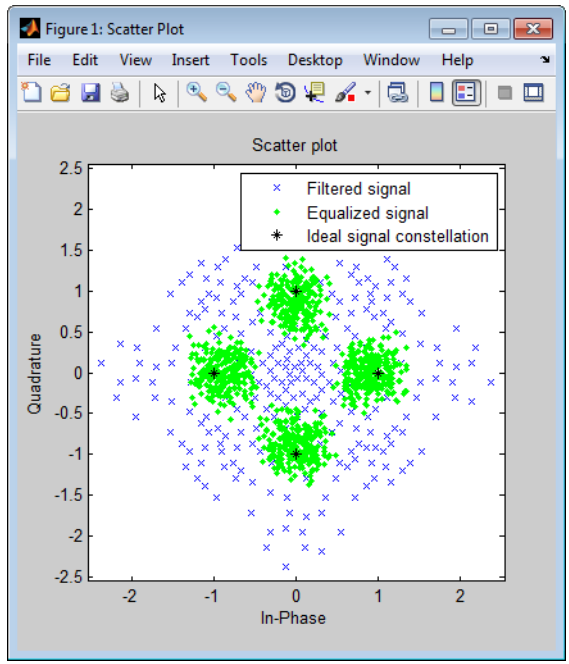
```
% Set up parameters and signals.
M = 4; % Alphabet size for modulation
msg = randi([0 M-1],1500,1); % Random message
hMod = comm.QPSKModulator('PhaseOffset',0);
modmsg = step(hMod,msg); % Modulate using QPSK.
trainlen = 500; % Length of training sequence
chan = [.986; .845; .237; .123+.31i]; % Channel coefficients
filtmsg = filter(chan,1,modmsg); % Introduce channel distortion.

% Equalize the received signal.
eq1 = lineareq(8, lms(0.01)); % Create an equalizer object.
eq1.SigConst = step(hMod,(0:M-1)')'; % Set signal constellation.
[symbolest,yd] = equalize(eq1,filtmsg,modmsg(1:trainlen)); % Equalize.

% Plot signals.
h = scatterplot(filtmsg,1,trainlen,'bx'); hold on;
scatterplot(symbolest,1,trainlen,'g.',h);
scatterplot(eq1.SigConst,1,0,'k*',h);
legend('Filtered signal','Equalized signal',...
   'Ideal signal constellation');
hold off;

% Compute error rates with and without equalization.
hDemod = comm.QPSKDemodulator('PhaseOffset',0);
demodmsg_noeq = step(hDemod,filtmsg); % Demodulate unequalized signal.
demodmsg = step(hDemod,yd); % Demodulate detected signal from equalizer.
hErrorCalc = comm.ErrorRate; % ErrorRate calculator
ser_noEq = step(hErrorCalc, ...
    msg(trainlen+1:end), demodmsg_noeq(trainlen+1:end));
reset(hErrorCalc)
ser_Eq = step(hErrorCalc, msg(trainlen+1:end),demodmsg(trainlen+1:end));
disp('Symbol error rates with and without equalizer:')
disp([ser_Eq(1) ser_noEq(1)])
```

The example goes on to determine how many errors occur in trying to recover the modulated message with and without the equalizer. The symbol error rates, below, show that the equalizer improves the performance significantly.

```
Symbol error rates with and without equalizer:
         0    0.3410
```

The example also creates a scatter plot that shows the signal before and after equalization, as well as the signal constellation for QPSK modulation. Notice on the plot

that the points of the equalized signal are clustered more closely around the points of the signal constellation.



Equalizing Using a Training Sequence in Simulink

To train a non-CMA equalizer block at the beginning of each frame throughout the simulation, follow these steps:

1. Clear the **Mode input port** check box.
2. Provide the training sequence at the input port labeled `Desired`. Valid training symbols are those listed in the **Signal constellation** vector. The block operates in training mode at the beginning of each frame and switches to decision-directed mode when it runs out of training symbols.

Typically, the symbol periods of the `Input` and `Desired` inputs match; that is, the sample time of the `Desired` signal is k times the sample time of the `Input` signal, where k is the **Number of samples per symbol** parameter in the equalizer block. If your training sequence is constant throughout the simulation, the Simulink Constant block is a convenient way to specify the sequence without having to specify a sample time explicitly.

To train a non-CMA equalizer block only on selected frames during the simulation, see "Equalize in Decision-Directed Mode Using MATLAB" on page 4-246.

Equalize in Decision-Directed Mode Using MATLAB

Decision-directed mode means the equalizer uses a detected version of its output signal when adapting the weights. Adaptive equalizers typically start with a training sequence (as mentioned in "Equalize Using a Training Sequence in MATLAB" on page 4-243) and switch to decision-directed mode after exhausting all symbols in the training sequence. CMA equalizers are an exception, using neither training mode nor decision-directed mode.

For non-CMA equalizers, the `equalize` function operates in decision-directed mode when one of these conditions is true:

- The syntax does not include a training sequence.
- The equalizer has exhausted all symbols in the training sequence and still has more input symbols to process.

The example in "Controlling the Use of Training or Decision-Directed Mode" on page 4-247 uses training mode when processing the first `trainlen` symbols of the input signal, and decision-directed mode thereafter. The example below discusses another scenario.

Example: Equalizing Multiple Times, Varying the Mode

If you invoke `equalize` multiple times with the same equalizer object to equalize a series of signal vectors, you might use a training sequence the first time you call the function and omit the training sequence in subsequent calls. Each iteration of the `equalize` function after the first one operates completely in decision-directed mode. However, because the `ResetBeforeFiltering` property of the equalizer object is set to `0`, the `equalize` function uses the existing state information in the equalizer object when starting each iteration's equalization operation. As a result, the training affects all equalization operations, not just the first.

The code below illustrates this approach. Notice that the first call to `equalize` uses a training sequence as an input argument, and the second call to `equalize` omits a training sequence.

```
M = 4; % Alphabet size for modulation
msg = randi([0 M-1],1500,1); % Random message
hMod = comm.QPSKModulator('PhaseOffset',0);
modmsg = step(hMod,msg); % Modulate using QPSK.
trainlen = 500; % Length of training sequence
```

```
chan = [.986; .845; .237; .123+.31i]; % Channel coefficients
filtmsg = filter(chan,1,modmsg); % Introduce channel distortion.

% Set up equalizer.
eqlms = lineareq(8, lms(0.01)); % Create an equalizer object.
eqlms.SigConst = step(hMod,(0:M-1)')'; % Set signal constellation.
% Maintain continuity between calls to equalize.
eqlms.ResetBeforeFiltering = 0;

% Equalize the received signal, in pieces.
% 1. Process the training sequence.
s1 = equalize(eqlms,filtmsg(1:trainlen),modmsg(1:trainlen));
% 2. Process some of the data in decision-directed mode.
s2 = equalize(eqlms,filtmsg(trainlen+1:800));
% 3. Process the rest of the data in decision-directed mode.
s3 = equalize(eqlms,filtmsg(801:end));
s = [s1; s2; s3]; % Full output of equalizer
```

Equalize in Decision-Directed Mode Using Simulink

Decision-directed mode means that the equalizer uses a detected version of its output signal when adapting the weights. Adaptive equalizers typically start with a training sequence (as mentioned in "Equalizing Using a Training Sequence in Simulink" on page 4-245) and switch to decision-directed mode after exhausting all symbols in the training sequence. CMA equalizers are an exception, using neither training mode nor decision-directed mode. The non-CMA equalizer blocks in this library operate in decision-directed mode when one of these conditions is true:

- The equalizer started processing the current input frame in training mode, exhausted all symbols in the training sequence frame, and still has more input symbols to process.
- The **Mode input port** check box is selected and the `Mode` input signal is `0`.

Controlling the Use of Training or Decision-Directed Mode

You can configure a non-CMA equalizer block so that it adapts in training mode for the beginning or the entirety of *selected* frames. To achieve this level of control over the equalizer's mode, follow these steps:

1. Enable the `Mode` input port by checking the **Mode input port** check box.
2. Send a binary-valued scalar signal to the `Mode` input port. The `Mode` input enables you to toggle back and forth between training mode and decision-directed mode. The significance of this signal is as follows:

- When the `Mode` input is `0`, the equalizer operates in decision-directed mode on the entire frame and ignores the `Desired` input.
- When the `Mode` input is `1`, the equalizer operates in training mode at the beginning of the frame until it exhausts the symbols in the `Desired` input, and operates in decision-directed mode afterwards. If the `Mode` input is `1` and the `Desired` input has as many symbols as the `Input` signal has, then the equalizer operates in training mode on the entire frame.

Delays from Equalization

For proper equalization using adaptive algorithms other than CMA, you should set the reference tap so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay between the modulator output and the equalizer output is equal to `(RefTap-1)/nSampPerSym`

symbols. Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap in a linear equalizer, or the center tap of the forward filter in a decision-feedback equalizer.

For CMA equalizers, the expression above does not apply because a CMA equalizer has no reference tap. If you need to know the delay, you can find it empirically after the equalizer weights have converged. Use the `xcorr` function to examine cross-correlations of the modulator output and the equalizer output.

Techniques for Working with Delays

Here are some typical ways to take a delay of D into account by padding or truncating data:

- Pad your original data with D extra symbols at the end. Before comparing the original data with the received data, omit the first D symbols of the received data. In this approach, all the original data (not including the padding) is accounted for in the received data.
- Before comparing the original data with the received data, omit the last D symbols of the original data and the first D symbols of the received data. In this approach, some of the original symbols are not accounted for in the received data.

The example below illustrates the latter approach. For an example that illustrates both approaches in the context of interleavers, see “Delays of Convolutional Interleavers” on page 4-160.

```
M = 2; % Use BPSK modulation for this example.
msg = randi([0 M-1],1000,1); % Random data
hMod = comm.BPSKModulator('PhaseOffset',0);
modmsg = step(hMod,msg); % Modulate
trainlen = 100; % Length of training sequence
trainsig = modmsg(1:trainlen); % Training sequence

% Define an equalizer and equalize the received signal.
eqlin = lineareq(3,normlms(.0005,.0001),pskmod(0:M-1,M));
eqlin.RefTap = 2; % Set reference tap of equalizer.
[eqsig,detsym] = equalize(eqlin,modmsg,trainsig); % Equalize.

hDemod = comm.BPSKDemodulator('PhaseOffset',0);
detmsg = step(hDemod,detsym); % Demodulate the detected signal.

% Compute bit error rate while compensating for delay introduced by RefTap
% and ignoring training sequence.
D = (eqlin.RefTap -1)/eqlin.nSampPerSym;
hErrorCalc = comm.ErrorRate('ReceiveDelay',D);
berVec = step(hErrorCalc, msg(trainlen+1:end), detmsg(trainlen+1:end));
ber = berVec(1)
numerrs = berVec(2)
```

The output is below.

```
numerrs =

     0


ber =

     0
```

Equalize Using a Loop

If your data is partitioned into a series of vectors (that you process within a loop, for example), you can invoke the `equalize` function multiple times, saving the equalizer's internal state information for use in a subsequent invocation. In particular, the final values of the `WeightInputs` and `Weights` properties in one equalization operation should be the initial values in the next equalization operation. This section gives an example, followed by more general procedures for equalizing within a loop.

Example: Adaptive Equalization Within a Loop

The example below illustrates how to use `equalize` within a loop, varying the equalizer between iterations. Because the example is long, this discussion presents it in these steps:

If you want to equalize iteratively while potentially changing equalizers between iterations, see “Changing the Equalizer Between Iterations” on page 4-253 for help generalizing from this example to other cases.

Initializing Variables

The beginning of the example defines parameters and creates three equalizer objects:

- An RLS equalizer object.
- An LMS equalizer object.
- A variable, `eq_current`, that points to the equalizer object to use in the current iteration of the loop. Initially, this points to the RLS equalizer object. After the second iteration of the loop, `eq_current` is redefined to point to the LMS equalizer object.

```
% Set up parameters.
M = 16; % Alphabet size for modulation
sigconst = step(comm.RectangularQAMModulator(M),(0:M-1)');
                                    % Signal constellation for 16-QAM
chan = [1 0.45 0.3+0.2i]; % Channel coefficients
hMod = comm.RectangularQAMModulator(M); % QAMModulator System object

% Set up equalizers.
eqrls = lineareq(6, rls(0.99,0.1)); % Create an RLS equalizer object.
eqrls.SigConst = sigconst'; % Set signal constellation.
eqrls.ResetBeforeFiltering = 0; % Maintain continuity between iterations.
eqlms = lineareq(6, lms(0.003)); % Create an LMS equalizer object.
eqlms.SigConst = sigconst'; % Set signal constellation.
eqlms.ResetBeforeFiltering = 0; % Maintain continuity between iterations.
eq_current = eqrls; % Point to RLS for first iteration.
```

Simulating the System Using a Loop

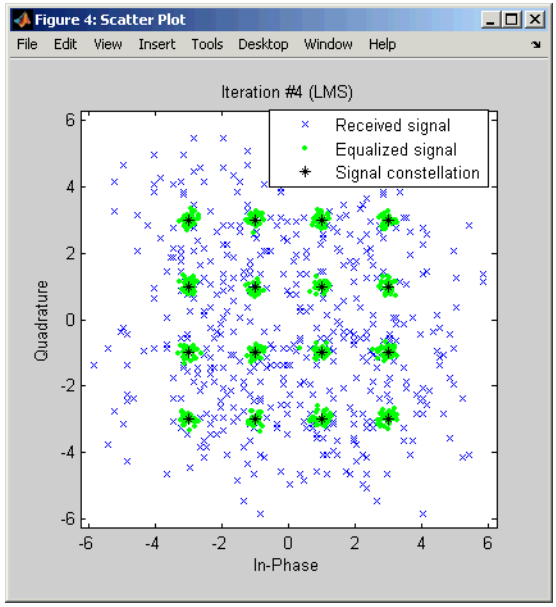
The next portion of the example is a loop that

- Generates a signal to transmit and selects a portion to use as a training sequence in the first iteration of the loop
- Introduces channel distortion
- Equalizes the distorted signal using the chosen equalizer for this iteration, retaining the final state and weights for later use
- Plots the distorted and equalized signals, for comparison
- Switches to an LMS equalizer between the second and third iterations

```
% Main loop
for jj = 1:4
```

```
    msg = randi([0 M-1],500,1); % Random message
    modmsg = step(hMod,msg); % Modulate using 16-QAM.

    % Set up training sequence for first iteration.
    if jj == 1
       ltr = 200; trainsig = modmsg(1:ltr);
    else
       % Use decision-directed mode after first iteration.
       ltr = 0; trainsig = [];
    end

    % Introduce channel distortion.
    filtmsg = filter(chan,1,modmsg);

    % Equalize the received signal.
    s = equalize(eq_current,filtmsg,trainsig);

    % Plot signals.
    h = scatterplot(filtmsg(ltr+1:end),1,0,'bx'); hold on;
    scatterplot(s(ltr+1:end),1,0,'g.',h);
    scatterplot(sigconst,1,0,'k*',h);
    legend('Received signal','Equalized signal','Signal constellation');
    title(['Iteration #' num2str(jj) ' (' eq_current.AlgType ')']);
    hold off;

    % Switch from RLS to LMS after second iteration.
    if jj == 2
       eqlms.WeightInputs = eq_current.WeightInputs; % Copy final inputs.
       eqlms.Weights = eq_current.Weights; % Copy final weights.
       eq_current = eqlms; % Make eq_current point to eqlms.
    end
end
```

The example produces one scatter plot for each iteration, indicating the iteration number and the adaptive algorithm in the title. A sample plot is below. Your plot might look different because this example uses random numbers.



Procedures for Equalizing Within a Loop

This section describes two procedures for equalizing within a loop. The first procedure uses the same equalizer in each iteration, and the second is useful if you want to change the equalizer between iterations.

Using the Same Equalizer in Each Iteration

The typical procedure for using `equalize` within a loop is as follows:

1. Before the loop starts, create the equalizer object that you want to use in the first iteration of the loop.
2. Set the equalizer object's `ResetBeforeFiltering` property to `0` to maintain continuity between successive invocations of `equalize`.
3. Inside the loop, invoke `equalize` using a syntax like one of these:

   ```
   y = equalize(eqz,x,trainsig);
   y = equalize(eqz,x);
   ```

   The `equalize` function updates the state and weights of the equalizer at the end of the current iteration. In the next iteration, the function continues from where it finished in the previous iteration because `ResetBeforeFiltering` is set to `0`.

This procedure is similar to the one used in "Example: Equalizing Multiple Times, Varying the Mode" on page 4-246. That example uses `equalize` multiple times but not within a loop.

Changing the Equalizer Between Iterations

In some applications, you might want to modify the adaptive algorithm between iterations. For example, you might use a CMA equalizer for the first iteration and an LMS or RLS equalizer in subsequent iterations. The procedure below gives one way to accomplish this, roughly following the example in "Example: Adaptive Equalization Within a Loop" on page 4-249:

1. Before the loop starts, create the different kinds of equalizer objects that you want to use during various iterations of the loop.

   For example, create one CMA equalizer object, `eqcma`, and one LMS equalizer object, `eqlms`.

2. For each equalizer object, set the `ResetBeforeFiltering` property to `0` to maintain continuity between successive invocations of `equalize`.

3. Create a variable `eq_current` that points to the equalizer object you want to use for the first iteration. Use `=` to establish the connection so that the two objects get updated together:

   ```
   eq_current = eqcma; % Point to eqcma.
   ```

   The purpose of `eq_current` is to represent the equalizer used in each iteration, where you can switch equalizers from one iteration to the next by using a command like `eq_current = eqlms`. The example illustrates this approach near the end of its loop.

4. Inside the loop, perform these steps:

   a. Invoke `equalize` using a syntax like one of these:

      ```
      y = equalize(eq_current,x,trainsig);
      y = equalize(eq_current,x);
      ```

   b. Copy the values of the `WeightInputs` and `Weights` properties from `eq_current` to the equalizer object that you want to use for the next iteration. Use dot notation. For example,

      ```
      eqlms.WeightInputs = eq_current.WeightInputs;
      eqlms.Weights = eq_current.Weights;
      ```

c Redefine `eq_current` to point to the equalizer object that you want to use for the next iteration, using `=`. Now `eq_current` is set up for the next iteration, because it represents the new kind of equalizer but retains the old values for the state and weights.

The reason for creating multiple equalizer objects and then copying the state and weights, instead of simply changing the equalizer class or adaptive algorithm in a single equalizer object, is that the class and adaptive algorithm properties of an equalizer object are fixed.

MLSE Equalizers

Section Overview

In Communications System Toolbox, the `mlseeq` function and MLSE Equalizer block use the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. Both the function and the block output the maximum likelihood sequence estimate of the signal, using an estimate of the channel modeled as a finite input response (FIR) filter.

Decoding a received signal uses these steps:

1 Applies the FIR filter, corresponding to the channel estimate, to the symbols in the input signal.

2 Uses the Viterbi algorithm to compute the traceback paths and the state metric, which are the numbers assigned to the symbols at each step of the Viterbi algorithm. The metrics are based on Euclidean distance.

3 Outputs the maximum likelihood sequence estimate of the signal, as a sequence of complex numbers corresponding to the constellation points of the modulated signal.

An MLSE equalizer yields the best possible performance, in theory, but is computationally intensive.

For background material about MLSE equalizers, see the works listed in "Selected Bibliography for Equalizers".

When using the MLSE Equalizer block, you specify the channel estimate and the signal constellation that the modulator in your model uses. If applicable, you can also specify a preamble and/or postamble that you expect to accompany your data. For full details on options, see the reference page for the MLSE Equalizer block.

Equalizing a Vector Signal

In its simplest form, the `mlseeq` function equalizes a vector of modulated data when you specify the estimated coefficients of the channel (modeled as an FIR filter), the signal constellation for the modulation type, and the traceback depth that you want the Viterbi algorithm to use. Larger values for the traceback depth can improve the results from the equalizer but increase the computation time.

An example of the basic syntax for `mlseeq` is below.

```
M = 4; hMod = comm.QPSKModulator;
const = step(hMod,(0:M-1)'); % 4-PSK constellation
msg = step(hMod,[1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]'); % Modulated message
chcoeffs = [.986; .845; .237; .12345+.31i]; % Channel coefficients
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
tblen =  10; % Traceback depth for equalizer
chanest = chcoeffs; % Assume the channel is known exactly.
hMLSEE = comm.MLSEEqualizer('TracebackDepth',tblen,...
    'Channel',chanest, 'Constellation',const);
msgEq = step(hMLSEE,filtmsg); % Equalize.
```

The `mlseeq` function has two operation modes:

- Continuous operation mode enables you to process a series of vectors using repeated calls to `mlseeq`, where the function saves its internal state information from one call to the next. To learn more, see "Equalizing in Continuous Operation Mode" on page 4-256.
- Reset operation mode enables you to specify a preamble and postamble that accompany your data. To learn more, see "Use a Preamble or Postamble" on page 4-259.

If you are not processing a series of vectors and do not need to specify a preamble or postamble, the operation modes are nearly identical. However, they differ in that continuous operation mode incurs a delay, while reset operation mode does not. The

example above could have used either mode, except that substituting continuous operation mode would have produced a delay in the equalized output. To learn more about the delay in continuous operation mode, see "Delays in Continuous Operation Mode" on page 4-256.

Equalizing in Continuous Operation Mode

If your data is partitioned into a series of vectors (that you process within a loop, for example), continuous operation mode is an appropriate way to use the `mlseeq` function. In continuous operation mode, `mlseeq` can save its internal state information for use in a subsequent invocation and can initialize using previously stored state information. To choose continuous operation mode, use `'cont'` as an input argument when invoking `mlseeq`.

Note: Continuous operation mode incurs a delay, as described in "Delays in Continuous Operation Mode" on page 4-256. Also, continuous operation mode cannot accommodate a preamble or postamble.

Procedure for Continuous Operation Mode

The typical procedure for using continuous mode within a loop is as follows:

1. Before the loop starts, create three empty matrix variables (for example, `sm`, `ts`, `ti`) that eventually store the state metrics, traceback states, and traceback inputs for the equalizer.
2. Inside the loop, invoke `mlseeq` using a syntax like

   ```
   [y,sm,ts,ti] = mlseeq(x,chcoeffs,const,tblen,'cont',nsamp,sm,ts,ti);
   ```

   Using `sm`, `ts`, and `ti` as input arguments causes `mlseeq` to continue from where it finished in the previous iteration. Using `sm`, `ts`, and `ti` as output arguments causes `mlseeq` to update the state information at the end of the current iteration. In the first iteration, `sm`, `ts`, and `ti` start as empty matrices, so the first invocation of the `mlseeq` function initializes the metrics of all states to 0.

Delays in Continuous Operation Mode

Continuous operation mode with a traceback depth of `tblen` incurs an output delay of `tblen` symbols. This means that the first `tblen` output symbols are unrelated to the input signal, while the last `tblen` input symbols are unrelated to the output signal. For

example, the command below uses a traceback depth of 3, and the first 3 output symbols are unrelated to the input signal of `ones(1,10)`.

```
y = step(comm.MLSEEqualizer('Channel',1, 'Constellation',[-7:2:7], ...
    'TracebackDepth',3,'TerminationMethod', 'Continuous'), ...
    complex(ones(10,1)))
y =

    -7    -7    -7     1     1     1     1     1     1     1
```

Keeping track of delays from different portions of a communication system is important, especially if you compare signals to compute error rates. The example in "Example: Continuous Operation Mode" on page 4-257 illustrates how to take the delay into account when computing an error rate.

Example: Continuous Operation Mode

The example below illustrates the procedure for using continuous operation mode within a loop. Because the example is long, this discussion presents it in multiple steps:

Initializing Variables

The beginning of the example defines parameters, initializes the state variables `sm`, `ts`, and `ti`, and initializes variables that accumulate results from each iteration of the loop.

```
n = 200; % Number of symbols in each iteration
numiter = 25; % Number of iterations
M = 4; % Use 4-PSK modulation.
hMod = comm .QPSKModulator('PhaseOffset',0);
const = step(hMod,(0:M-1)'); % PSK constellation
chcoeffs = [1 ; 0.25]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
tblen = 10; % Traceback depth for equalizer
sm = []; ts = []; ti = []; % Initialize equalizer data.
% Initialize cumulative results.
fullmodmsg = []; fullfiltmsg = []; fullrx = [];

hMLSEE = comm.MLSEEqualizer('TracebackDepth',tblen, 'Channel', chanest, ...
    'Constellation',const, 'TerminationMethod', 'Continuous');
```

Simulating the System Using a Loop

The middle portion of the example is a loop that generates random data, modulates it using baseband PSK modulation, and filters it. Finally, `mlseeq` equalizes the filtered

data. The loop also updates the variables that accumulate results from each iteration of the loop.

```
for jj = 1:numiter
    msg = randi([0 M-1],n,1); % Random signal vector
    modmsg = step(hMod,msg); % PSK-modulated signal
    filtmsg = filter(chcoeffs,1,modmsg); % Filtered signal
    rx = step(hMLSEE,filtmsg); % Equalize

    % Update vectors with cumulative results.
    fullmodmsg = [fullmodmsg; modmsg];
    fullfiltmsg = [fullfiltmsg; filtmsg];
    fullrx = [fullrx; rx];
end
```

Computing an Error Rate and Plotting Results

The last portion of the example computes the symbol error rate from all iterations of the loop. The `symerr` function compares selected portions of the received and transmitted signals, not the entire signals. Because continuous operation mode incurs a delay whose length in samples is the traceback depth (`tblen`) of the equalizer, it is necessary to exclude the first `tblen` samples from the received signal and the last `tblen` samples from the transmitted signal. Excluding samples that represent the delay of the equalizer ensures that the symbol error rate calculation compares samples from the received and transmitted signals that are meaningful and that truly correspond to each other.

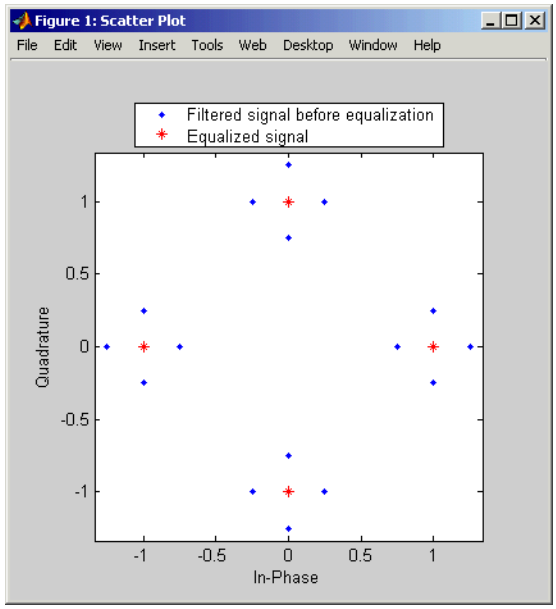
The example also plots the signal before and after equalization in a scatter plot. The points in the equalized signal coincide with the points of the ideal signal constellation for 4-PSK.

```
% Compute total number of symbol errors. Take the delay into account.
hErrorCalc = comm.ErrorRate('ReceiveDelay',10);
err = step(hErrorCalc, fullmodmsg, fullrx);
numsymerrs = err(1)

% Plot signal before and after equalization.
h = scatterplot(fullfiltmsg); hold on;
scatterplot(fullrx,1,0,'r*',h);
legend('Filtered signal before equalization','Equalized signal',...
   'Location','NorthOutside');
hold off;
```

The output and plot follow.

```
numsymerrs =
```

0



Use a Preamble or Postamble

Some systems include a sequence of known symbols at the beginning or end of a set of data. The known sequence at the beginning or end is called a *preamble* or *postamble*, respectively. The `mlseeq` function can accommodate a preamble and postamble that are already incorporated into its input signal. When you invoke the function, you specify the preamble and postamble as integer vectors that represent the sequence of known symbols by indexing into the signal constellation vector. For example, a preamble vector of `[1 4 4]` and a 4-PSK signal constellation of `[1 j -1 -j]` indicate that the modulated signal begins with `[1 -j -j]`.

If your system uses a preamble without a postamble, use a postamble vector of `[]` when invoking `mlseeq`. Similarly, if your system uses a postamble without a preamble, use a preamble vector of `[]`.

Use a Preamble in MATLAB

The example below illustrates how to accommodate a preamble when using `mlseeq`. The same preamble symbols appear at the beginning of the message vector and in the syntax

for `mlseeq`. If you want to use a postamble, you can append it to the message vector and also include it as the last input argument for `mlseeq`. In this example, however, the postamble input in the `mlseeq` syntax is an empty vector because the system uses no postamble.

```
M = 4; hMod = comm.QPSKModulator;% Use 4-PSK modulation.
const = step(hMod,(0:M-1)'); % PSK constellation
tblen = 16; % Traceback depth for equalizer

preamble = [3; 1]; % Expected preamble, as integers
msgIdx = randi([0 M-1],98,1); % Random symbols
msgIdx = [preamble; msgIdx]; % Include preamble at the beginning.
msg = step(hMod,msgIdx); % Modulated message
chcoeffs = [.623; .489+.234i; .398i; .21]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
hMLSEE = comm.MLSEEqualizer('TracebackDepth',tblen,...
    'Channel',chanest, 'Constellation',const, ...
    'PreambleSource', 'Property', 'Preamble', preamble);
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
d = step(hMLSEE,filtmsg);

% Symbol error rate
hErrorCalc = comm.ErrorRate;
serVec  = step(hErrorCalc, msg,d);
ser = serVec(1)
nsymerrs = serVec(2)
```

The output is below.

```
nsymerrs =

     0


ser =

     0
```

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, using your estimate of the channel modeled as a finite input response (FIR) filter.

The block decodes the received signal using these steps:

1. Applies the FIR filter corresponding to the channel estimate to the symbols in the input signal.
2. Uses the Viterbi algorithm to compute the traceback paths and the state metric, which are the numbers assigned to the symbols at each step of the Viterbi algorithm.
3. Outputs the maximum likelihood sequence estimate of the signal, as a sequence of complex numbers corresponding to the constellation points of the modulated signal.

An MLSE equalizer yields the best possible performance, in theory, but is computationally intensive.

When using the MLSE Equalizer block, you specify the channel estimate and the signal constellation that the modulator in your model uses. If applicable, you can also specify a preamble and/or postamble that you expect to accompany your data. For full details on options, see the reference page for the MLSE Equalizer block.

Selected Bibliography for Equalizers

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

[5] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, John Wiley & Sons, 1996.

Multiple-Input Multiple-Output (MIMO)

In this section...

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 is proposed for future standards (such as LTE-Advanced and IMT-Advanced).

The Communications System Toolbox product offers components to model:

- OSTBC (orthogonal space-time block coding technique)
- MIMO Fading Channels

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".

Orthogonal Space-Time Block Codes (OSTBC)

The Communications System Toolbox product provides components to 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 System 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

The Communications System Toolbox software also includes a MIMO fading channel object. You can use this object to model the fading channel characteristics of MIMO links. The object models both Rayleigh and Rician fading, and uses the Kronecker model for the spatial correlation between the links [1].

For more information, see the `comm.MIMOChannel` and `comm.LTEMIMOChannel` Help pages.

MIMO Examples

The following examples illustrate MIMO techniques or the use of MIMO components:

MATLAB

Concatenated OSTBC with TCM: OSTBC System objects

IEEE 802.11n Channel Models: `comm.MIMOChannel` System object

IEEE 802.16 Channel Models: `comm.MIMOChannel` System object

Introduction to MIMO Systems: Comparing MRC and OSTBC techniques

Spatial Multiplexing: techniques offering multiplexing gain

Simulink

Adaptive MIMO System with OSTBC: OSTBC and MIMO channel in Simulink

Concatenated OSTBC with TCM: OSTBC with blocks

IEEE® 802.16-2004 OFDM PHY Link, Including Space-Time Block Coding

LTE PHY Downlink with Spatial Multiplexing `comm.LTEMIMOChannel`System object

MIMO Decoder Using Simulink® and the MATLAB™ Function Block: Lattice decoder. You must install a HDL Coder user license to run this example.

OSTBC Over 3x2 Rayleigh Fading Channel

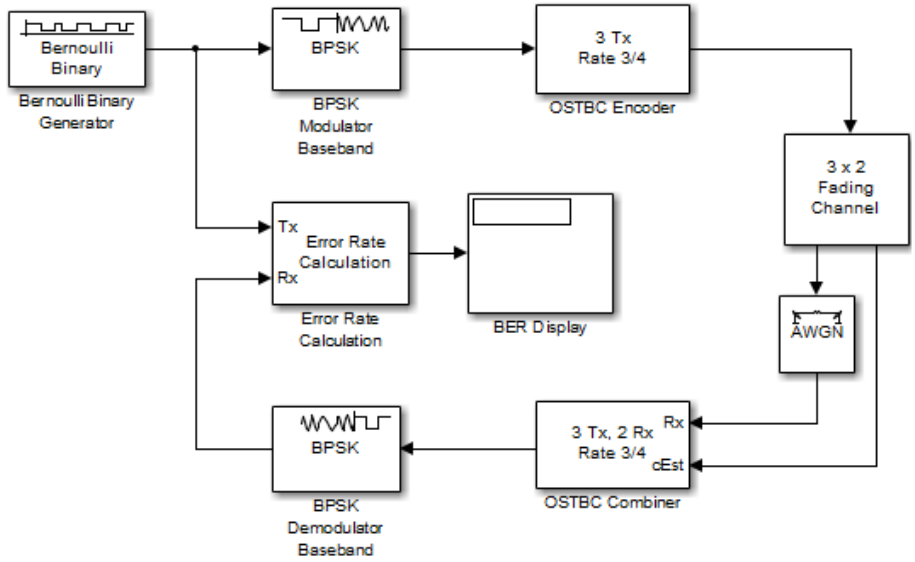
This example demonstrates the use of Orthogonal Space-Time Block Codes (OSTBC) to achieve diversity gains in a multiple-input multiple-output (MIMO) communication system. The example shows the transmission of data over three transmit antennas and two receive antennas (hence the 3x2 notation) using independent Rayleigh fading per link. This description covers the following:

- "Overview of the Simulation"
- "Orthogonal Space-Time Block Code"
- "Performance"

Overview of the Simulation

The model is shown in the following figure. To open the model, type `doc_ostbc32` at the MATLAB command line. The simulation creates a random binary signal, modulates it using a binary phase shift keying (BPSK) technique, and then encodes the waveform using a rate $\frac{3}{4}$ orthogonal space-time block code for transmission over the fading channel. The fading channel models six independent links, due to the three transmit by two receive antennae configuration as single-path Rayleigh fading processes. The simulation adds white Gaussian noise at the receiver. Then, it combines the signals from both receive antennas into a single stream for demodulation. For this combining process, the model assumes perfect knowledge of the channel gains at the receiver. Finally, the simulation compares the demodulated data with the original transmitted

data, computing the bit error rate. The simulation ends after processing 100 errors or 1e6 bits, whichever comes first.



Orthogonal Space-Time Block Code

This simulation uses an orthogonal space-time block code with three transmit antennas and a rate ¾ code, as shown below

$$\begin{pmatrix} s_1 & s_2 & s_2 \\ -s_2^* & s_1^* & 0 \\ s_3^* & 0 & -s_1^* \\ 0 & s_3^* & -s_2^* \end{pmatrix}$$

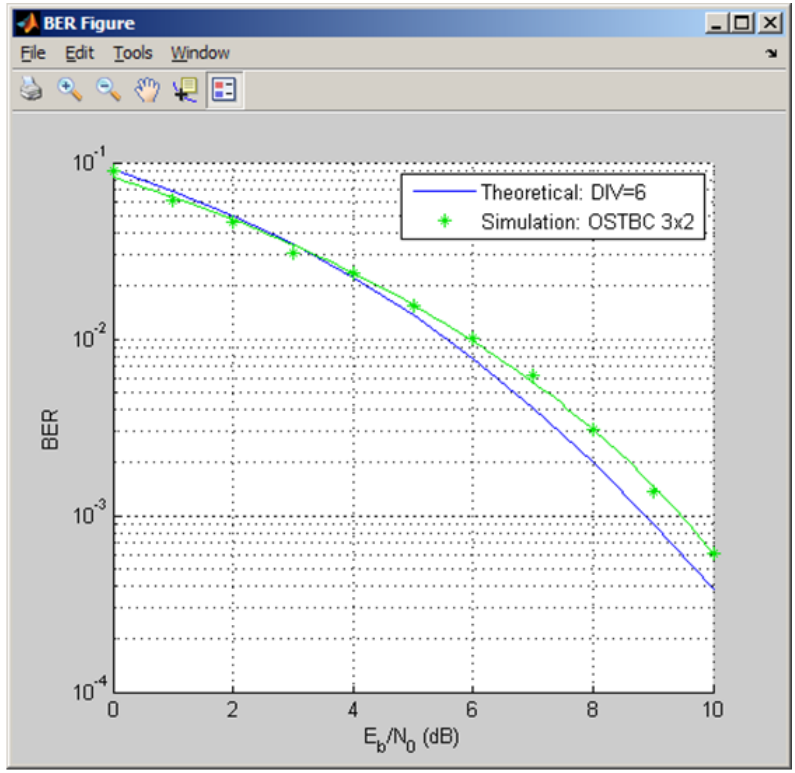
where s1, s2, s3 correspond to the three symbol inputs for which the output is given by the previous matrix. Note in the simulation that the input to the OSTBC Encoder block is a 3x1 vector signal and the output is a 4x3 matrix. The number of columns in the output signal indicates the number of transmit antennas for this simulation, where the first dimension is for time.

For the selected code, the output signal power per time step is $\frac{(12-3)}{4} = 2.25W$. Also, note that the channel symbol period for this simulation is $1e^{-3} * \frac{3}{4} = 7.5e^{-4}$ sec, due to the use of rate $\frac{3}{4}$ code. These two values are used in calibrating the white Gaussian noise added in the simulation. The parameters that the Receive Noise block specifies apply for each receiver the system employs.

Performance

Now compare the performance of the code with theoretical results using BERtool as an aid. For the theoretical results, the EbNo is directly scaled by the diversity order (six in this case). For the simulation, in the Receive Noise block, we account for only the diversity due to the transmitters (hence, the EbNo parameter is scaled by a factor of three).

The figure below compares the simulated BER for a range of EbNo values with the theoretical results for a diversity order of six.



Note the close alignment of the simulated results with the theoretical (especially. at low EbNo values). The fading channel modeled in the simulation is not completely static (has a low Doppler). As a result the channel is not held constant over the block symbols. Varying this parameter for the channel shows little variation between the results compared to the theoretical curve.

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 4-12.

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

For example, consider a data source that produces 1s with probability 0.1, 2s with probability 0.1, and 3s with probability 0.8. The main computational step in encoding data from this source using a Huffman code is to create a dictionary that associates each data symbol with a codeword. The 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 = [1 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 =

    [1]    [1x2 double]
    [2]    [1x2 double]
    [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 `1  0`.

```
dict{2,:}
```

```
ans =

     2


ans =

     1     0
```

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 = [1 2 3];
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 =

     1
```

Differential Pulse Code Modulation

In this section...

Section Overview

The quantization in the section "Quantize a Signal" on page 4-13 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 4-2 and "Represent Codebooks" on page 4-3, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

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

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

Represent Predictors

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

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

then the corresponding predictor vector for toolbox functions is

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

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

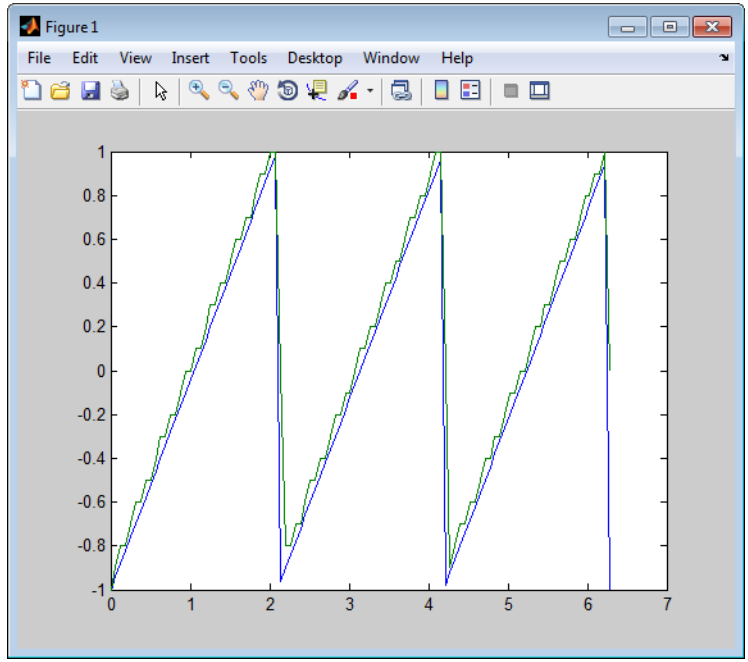
Example: DPCM Encoding and Decoding

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

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

The output is

```
distor =

    0.0327
```



Optimize DPCM Parameters

Section Overview

The section "Optimize Quantization Parameters" on page 4-4 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: μ-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 μ-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);
```

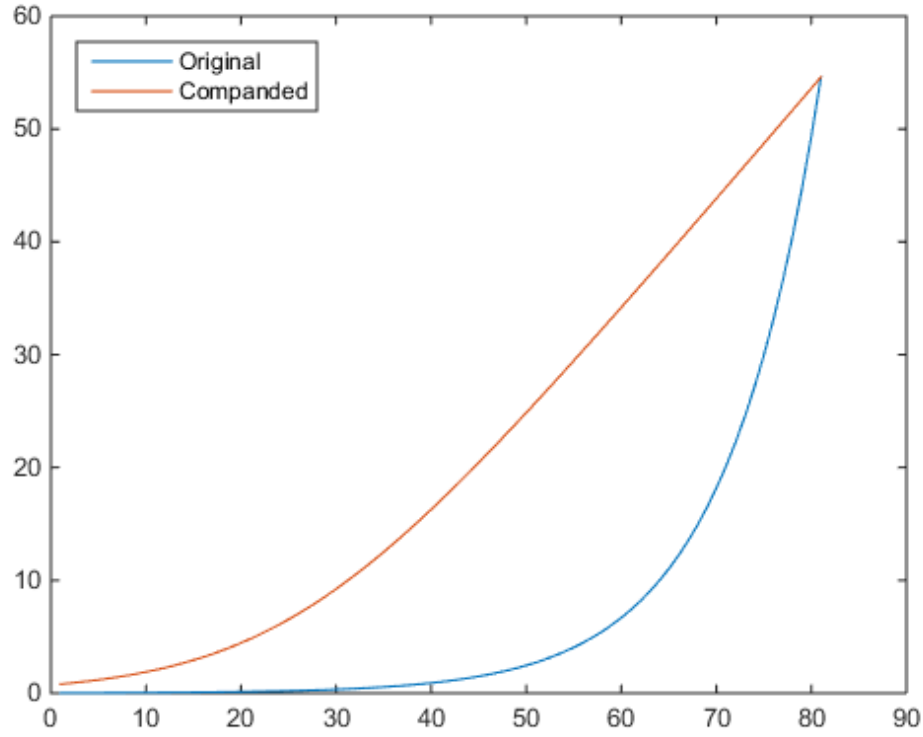
Compare the mean square distortions. The output shows that the distortion is smaller when companding is used. This is because equal-length intervals are well suited to the logarithm of `sig` but not well suited to `sig` itself.

```
[distor, distor2]
```

```
ans =

    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 = [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

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 = [3 3 1 3 3 3 3 3 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 = [10 20 70];
```

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 =

     1
```

Quantization

In this section...

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 and a codebook.

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

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    0.0024
```

Quantize a Signal

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.

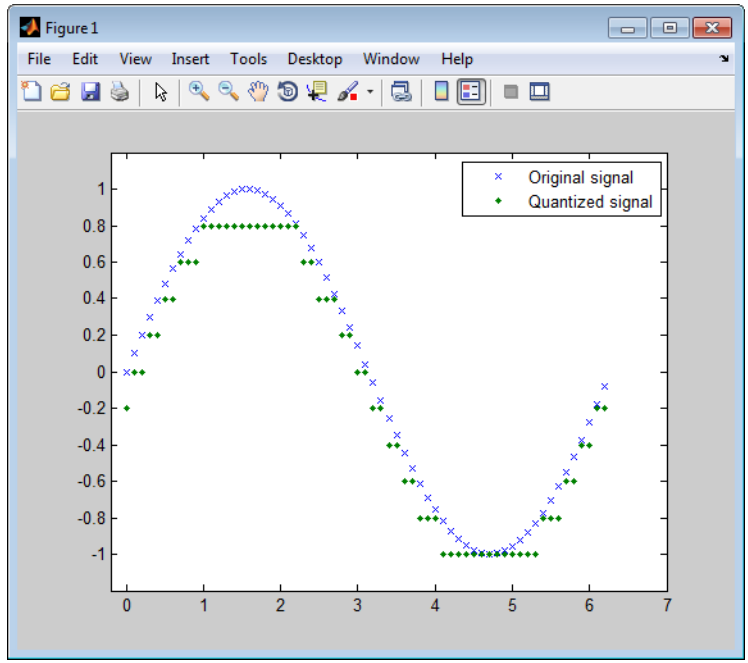
```
quantized =

  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])
```



OFDM Modulation

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 3x2 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.

```
hMod = 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;

hMod.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` method.

```
hDemod = comm.OFDMDemodulator(hMod);
```

```
hDemod.NumReceiveAntennas = 2;

modDim = info(hMod)


modDim =

     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(modDim.DataInputSize), randn(modDim.DataInputSize));
pilotIn = complex(randn(modDim.PilotInputSize), randn(modDim.PilotInputSize));
```

Apply OFDM modulation to the data and pilots.

```
modOut = step(hMod, dataIn, pilotIn);
```

Pass the modulated data through a 3x2 random channel.

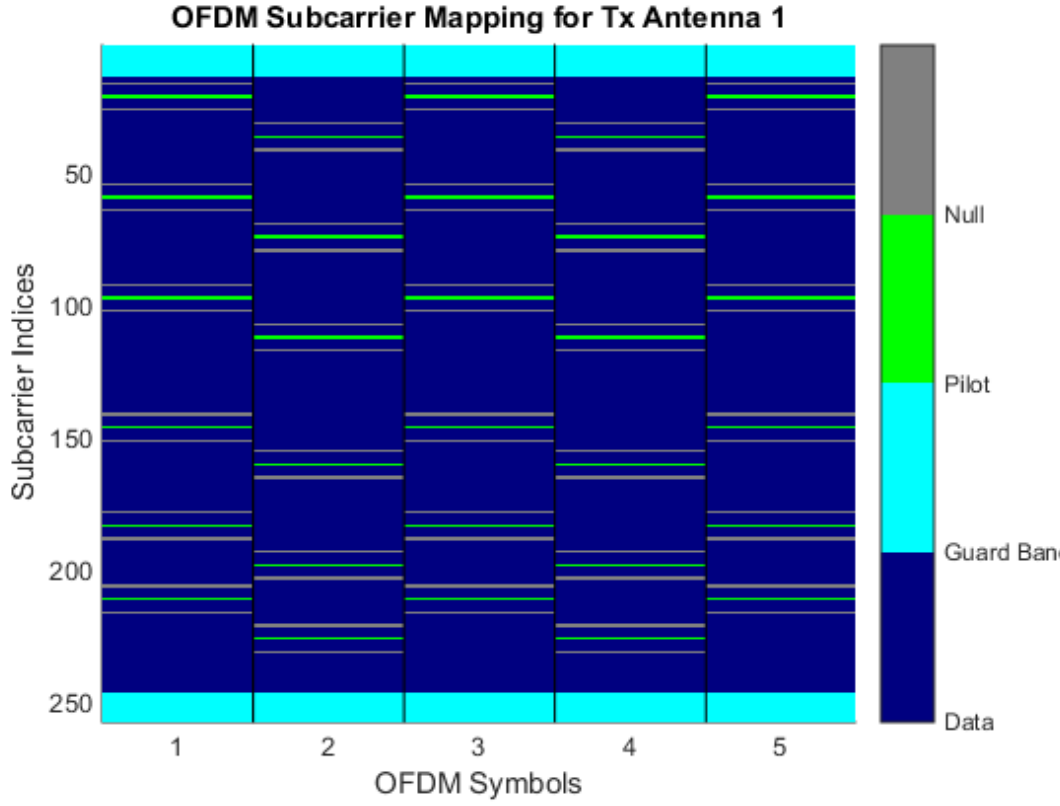
```
chanGain = complex(randn(3,2), randn(3,2));
chanOut = modOut * chanGain;
```

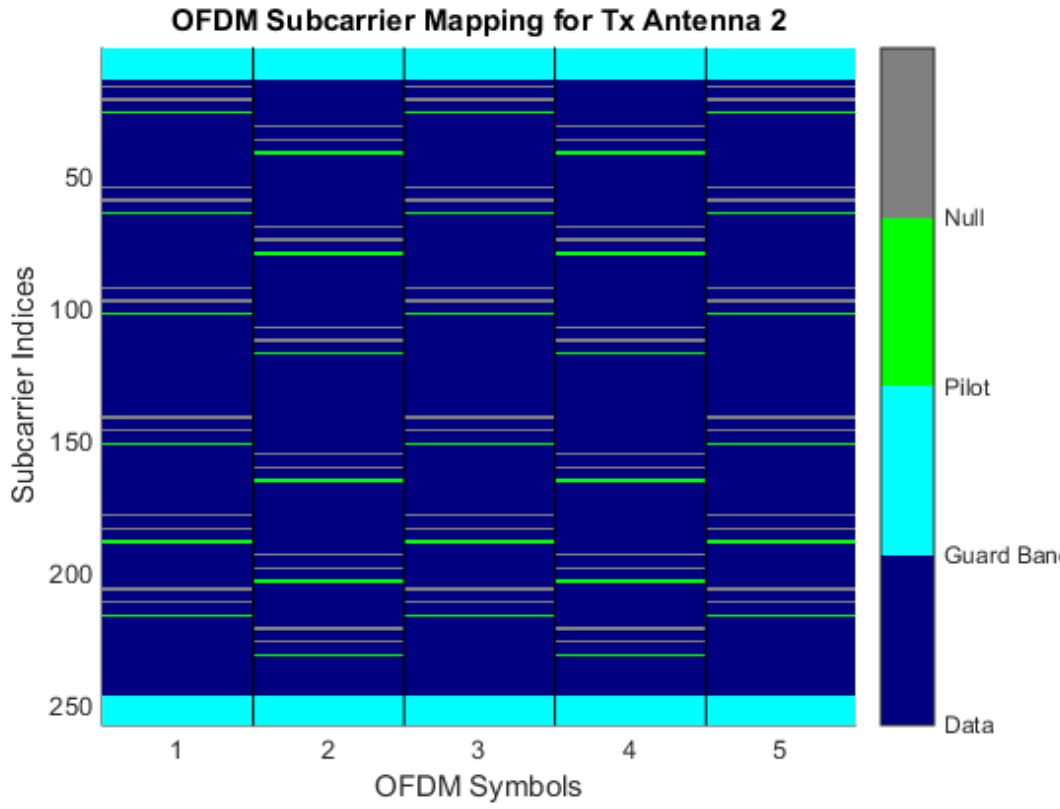
Demodulate the received data using the OFDM demodulator object.

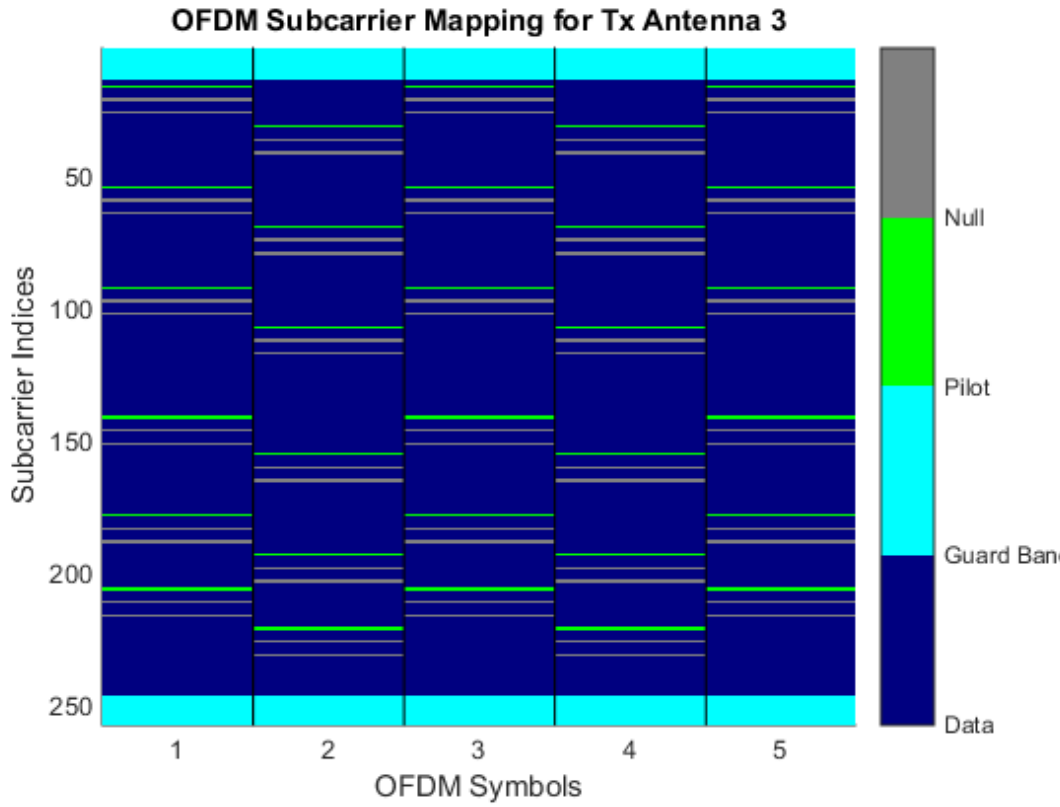
```
[dataOut, pilotOut] = step(hDemod, chanOut);
```

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(hMod)
```


OFDM Subcarrier Mapping for Tx Antenna 1
Subcarrier Indices
50
100
150
200
250
1
2
3
4
5
OFDM Symbols
Null
Pilot
Guard Band
Data


OFDM Subcarrier Mapping for Tx Antenna 2
Subcarrier Indices
50
100
150
200
250
1
2
3
4
5
OFDM Symbols
Null
Pilot
Guard Band
Data



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 =

     1
```

802.11a/g SER Simulation

This example shows how to perform a symbol error rate (SER) simulation of an 802.11 a/g radio link.

A basic 802.11 communications link using OFDM modulation with QPSK symbols is simulated. Consistent with the standard, there is a single transmit and a single receive antenna.

Construct the QPSK modulator and demodulator objects.

```
hQMod = comm.QPSKModulator;
hQDemod = comm.QPSKDemodulator;
```

Construct the OFDM modulator and demodulator pair using the default parameters. The default parameters are based on the 802.11 a/g standard.

```
hOFDMMod = comm.OFDMModulator;
hOFDMDemod = comm.OFDMDemodulator(hOFDMMod);
```

Use the `info` method to determine the required input dimensions for the OFDM modulator and display the results.

```
modDim = info(hOFDMMod);

disp(modDim)
```

```
    DataInputSize: [53 1]
       OutputSize: [80 1]
```

Create an AWGN channel object with a signal-to-noise ratio of 23 dB.

```
hAWGN = comm.AWGNChannel('NoiseMethod',...
    'Signal to noise ratio (SNR)','SNR', 23);
```

Set the random number generator to its default value to ensure that the results are repeatable. Determine the number of OFDM symbols per frame by using the first value of the `modDim.DataInputSize` array.

```
rng('default')      % Initialize random number generator
nFrames = 100;      % Number of OFDM frames to transmit
nSymbolsPerFrame = modDim.DataInputSize(1);
```

Create an error rate object for tracking error statistics and enable the object to be reset during each loop iteration.

```
hError = comm.ErrorRate('ResetInputPort',true);
SER = zeros(nFrames, 1);
```

Run the simulation over 100 OFDM frames (5300 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.

```
for k = 1:nFrames
    % Generate random data for each OFDM frame
    data = randi([0 3], nSymbolsPerFrame, 1);

    % Apply QPSK modulation
    modData = step(hQMod, data);

    % Modulate QPSK symbols using OFDM
    dataOFDM = step(hOFDMMod, modData);

    % Pass OFDM signal through AWGN channel
    receivedSignal = step(hAWGN, dataOFDM);

    % Demodulate OFDM data
    receivedOFDMData = step(hOFDMDemod, receivedSignal);

    % Demodulate QPSK data
    receivedData = step(hQDemod, receivedOFDMData);

    % Compute BER
    errors = step(hError, data, receivedData, 1);
    SER(k) = errors(1);
end
```

Display the symbol error data for the first ten frames.

```
disp(SER(1:10))
```

```
    0.0377
    0.0566
    0.0566
    0.1132
    0.0755
    0.0566
    0.0566
```

```
0.0566
0.0755
0.1509
```

OFDM with MIMO Simulation

This example shows how to the OFDM modulator and demodulator in a simple, 2×2 MIMO error rate simulation. The OFDM parameters are based on the 802.11n standard.

Construct a QPSK modulator and demodulator pair using the default settings.

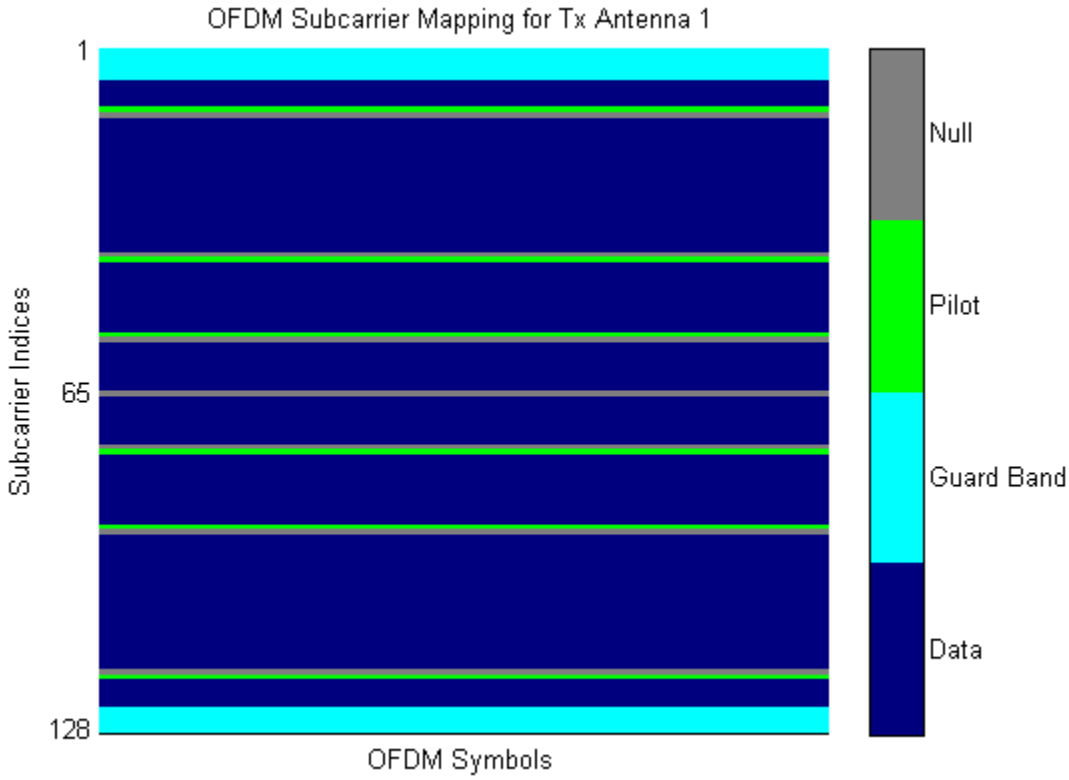
```
hQMod = comm.QPSKModulator;
hQDemod = comm.QPSKDemodulator;
```

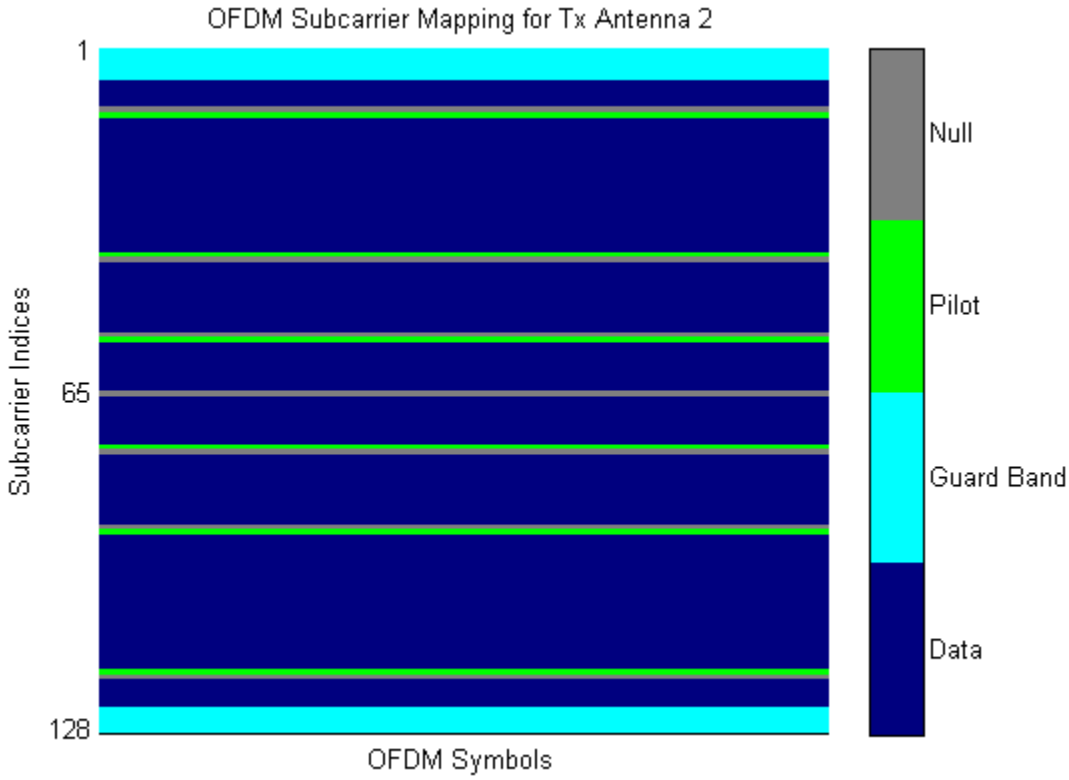
Construct an OFDM modulator and demodulator pair with user-specified pilot indices, an inserted DC null, two transmit antennas, and two receive antennas. Specify the pilot indices such that they vary from antenna to antenna.

```
hOFDMMod = 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);
hOFDMDemod = comm.OFDMDemodulator(hOFDMMod);
hOFDMDemod.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(hOFDMMod)
```


OFDM Subcarrier Mapping for Tx Antenna 1
1
65
128
Subcarrier Indices
OFDM Symbols
Null
Pilot
Guard Band
Data



Construct an AWGN channel System object with a 30 dB E_s/N_0 ratio.

```
hAWGN = comm.AWGNChannel(...
    'NoiseMethod', 'Signal to noise ratio (Es/No)', ...
    'EsNo', 30);
```

Determine the dimensions of the OFDM modulator by using the `info` method and generate random data symbols to fill 100 OFDM frames.

```
modDim = info(hOFDMMod);

% Number of OFDM frames to transmit
nFrames = 100;
```

```
% Generate data for each (subcarrier, symbol, tx antenna) triplet
rng('default')         % Initialize random number generator to default seed
dataInputDim = [nFrames 1 1] .* modDim.DataInputSize;
data = randi([0 3], dataInputDim);
```

Apply QPSK modulation to the random symbols and size the resulting data to match the OFDM modulator requirements.

```
modData = step(hQMod, data(:));

modData = reshape(modData, dataInputDim);
```

Create an `ErrorRate` System object to collect error statistics.

```
hError = comm.ErrorRate;
```

Simulate the OFDM system over 100 frames assuming a flat, 2×2, 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)*modDim.DataInputSize(1)+1:k*modDim.DataInputSize(1);

    % Generate random OFDM pilot symbols
    pilotData = complex(rand(modDim.PilotInputSize), ...
        rand(modDim.PilotInputSize));

    % Modulate QPSK symbols using OFDM
    dataOFDM = step(hOFDMMod, 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 the OFDM signal through Rayleigh and AWGN channels
    receivedSignal = step(hAWGN, dataOFDM * chGain);

    % Apply least squares solution to remove fading channel effects
    rxSigMF = chGain.' \ receivedSignal.';

    % Demodulate the OFDM data
    receivedOFDMData = step(hOFDMDemod, rxSigMF.');
```

```
    % Demodulate the QPSK data
    receivedData = step(hQDemod, receivedOFDMData(:));

    % Compute error statistics
    dataTmp = data(indData,:,:);
    errors = step(hError, dataTmp(:), receivedData);
end
```

Display the error data collected during the simulation.

```
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 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

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 $2\pi(M - 1)/M$.

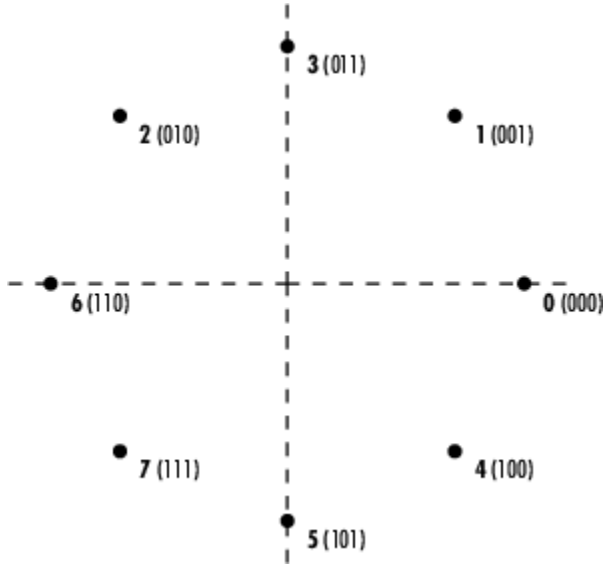
The following table indicates the relationship between binary representations in the input and phasors in the output. The second column of the table is an intermediate representation that the System object uses in its computations.

Modulator Input	Gray-Coded Ordering	Modulator Output
000	0	exp(0)

Modulator Input	Gray-Coded Ordering	Modulator Output
001	1	$\exp(j\pi/4)$
010	3	$\exp(j3\pi/4)$
011	2	$\exp(j\pi/2) = \exp(j2\pi/4)$
100	7	$\exp(j7\pi/4)$
101	6	$\exp(j3\pi/2) = \exp(j6\pi/4)$
110	4	$\exp(j\pi) = \exp(j4\pi/4)$
111	5	$\exp(j5\pi/4)$

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
$\exp(j\pi/2) = \exp(j2\pi/4)$	011
$\exp(j3\pi/4)$	010
$\exp(j\pi) = \exp(j4\pi/4)$	110
$\exp(j5\pi/4)$	111
$\exp(j3\pi/2) = \exp(j6\pi/4)$	101
$\exp(j7\pi/4)$	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 Eb/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 8-PSK gray coded constellation as well as a matching PSK demodulator (hDemod) System object

```
hMod = comm.PSKModulator('ModulationOrder',M, ...
                    'SymbolMapping','gray', ...
                    'PhaseOffset',0, ...
                    'BitInput',true);
hDemod = comm.PSKDemodulator('ModulationOrder',M, ...
                    'SymbolMapping','gray', ...
                    'PhaseOffset',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 0dB to 12dB in steps of 2dB.

```
% 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

$$P_E(M) = \text{erfc}\left(\sqrt{\frac{E_s}{N_0}}\sin\left(\frac{\pi}{M}\right)\right)$$

where erfc is the complementary error function, E_s/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, P_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, P_b, can be shown to be bounded by

$$\frac{P_E(M)}{\log_2 M} \le P_b \le \frac{M/2}{M-1}P_E(M)$$

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.

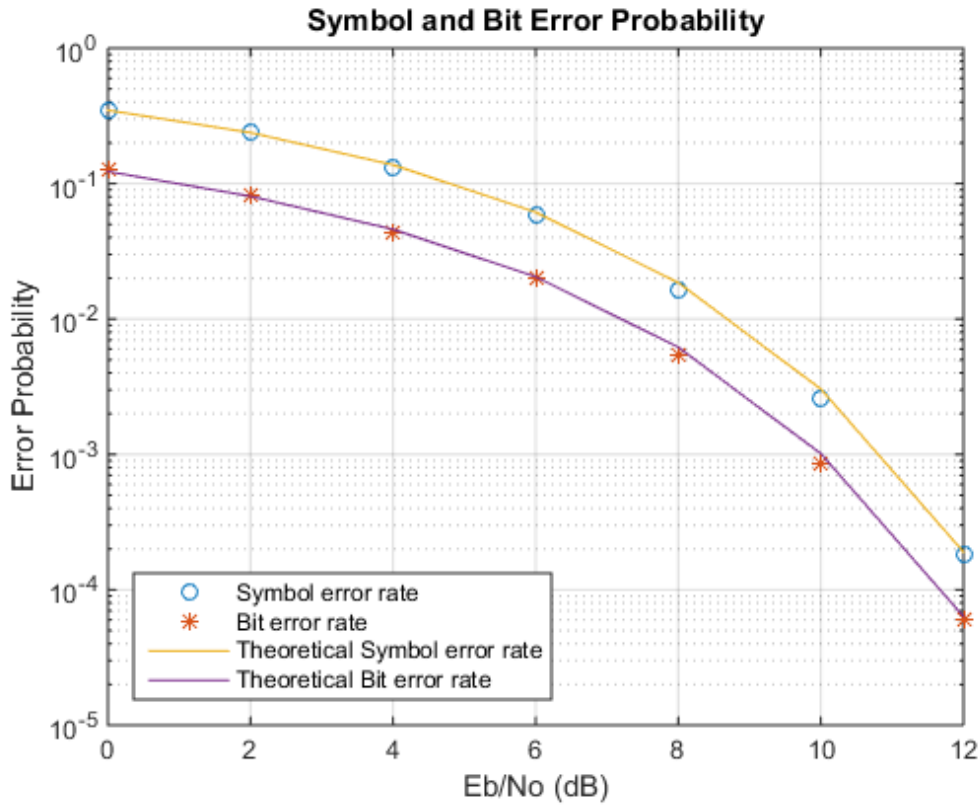
The following script plots the simulated symbol error rates (SERVec) and bit error rates (BERVec) together with the theoretical symbol error and bit error probabilities.

Calculate theoretical error probabilities.

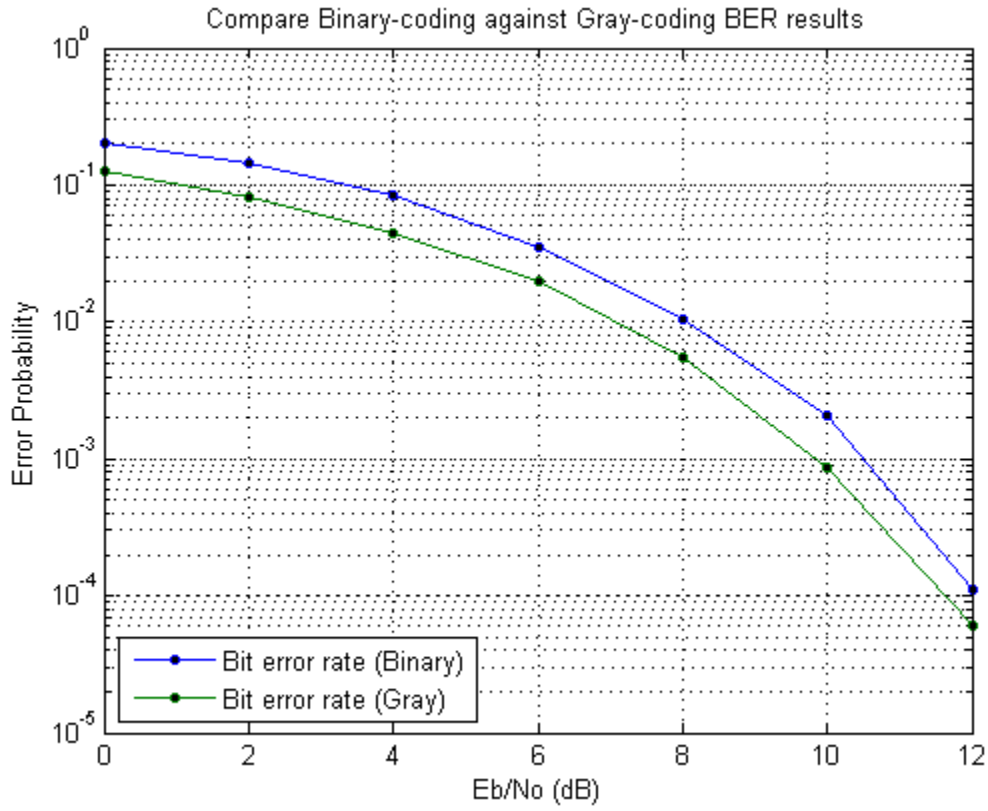
```
[theorBER, theorSER] = berawgn(EbNoVec, 'psk', M, 'nondiff');
```

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:


Compare Binary-coding against Gray-coding BER results
Error Probability
Eb/No (dB)
Bit error rate (Binary)
Bit error rate (Gray)
10^0
10^-1
10^-2
10^-3
10^-4
10^-5
0
2
4
6
8
10
12

Configure Eb/No for AWGN Channels with Coding

This example shows how to set the bit energy to noise density ratio (Eb/No) for communication links employing channel coding.

Construct a (15,9) Reed-Solomon encoder and a 16-PSK modulator. Set the System objects™ so that they accept bit inputs.

```
N = 15;
K = 9;
hEnc = comm.RSEncoder('CodewordLength', N, 'MessageLength', K, ...
    'BitInput',true);
hMod = comm.PSKModulator('ModulationOrder', 16, 'BitInput', true);
```

Create the corresponding Reed-Solomon decoder and 16-PSK demodulator objects.

```
hDec = comm.RSDecoder('CodewordLength', N,'MessageLength', K, ...
    'BitInput',true);
hDemod = comm.PSKDemodulator('ModulationOrder', 16, 'BitOutput', true);
```

Calculate the Reed-Solomon code rate based on the ratio of message symbols to the codeword length.

```
codeRate = hEnc.MessageLength/hEnc.CodewordLength;
```

Determine the bits per symbol for the PSK modulator.

```
bitsPerSymbol = log2(hMod.ModulationOrder);
```

Set the uncoded Eb/No in dB.

```
UncodedEbNo = 6;
```

Convert the uncoded Eb/No to the corresponding coded Eb/No using the code rate.

```
CodedEbNo = UncodedEbNo + 10*log10(codeRate);
```

Construct an AWGN channel taking into account the number of bits per symbol. Set the `EbNo` property of `hCh` to the coded Eb/No.

```
hCh = comm.AWGNChannel('BitsPerSymbol',bitsPerSymbol);
hCh.EbNo = CodedEbNo;
```

Set the random number generator seed so that results are repeatable.

```
prevState = rng;
rng(577)
```

Set the total number of errors and bits for the simulation. For accuracy, the simulation should run until a sufficient number of bit errors are encountered. The number of total bits is used to ensure that the simulation does not run too long.

```
totalErrors = 100;
totalBits = 1e6;
```

Construct an error rate calculator System object™ and initialize the error rate vector.

```
hErr = comm.ErrorRate;
errorVec = zeros(1,3);
```

Run the simulation to determine the BER.

```
while errorVec(2) < totalErrors && errorVec(3) < totalBits
    % Generate random bits
    dataIn = randi([0,1],360,1);
    % Use the RS (15,9) encoder to add error correction capability
    dataEnc = step(hEnc, dataIn);
    % Apply 16-PSK modulation
    txSig = step(hMod, dataIn);
    % Pass the modulated data through the AWGN channel
    rxSig = step(hCh, txSig);
    % Demodulate the received signal
    demodData = step(hDemod, rxSig);
    % Decode the demodulated data with the RS (15,9) decoder
    dataOut = step(hDec, demodData);
    % Collect error statistics
    errorVec = step(hErr, dataIn, demodData);
end
```

Display the resultant bit error rate.

```
ber = errorVec(1)
```

```
ber =

    0.1065
```

Return the random number generator to its initial state.

```
rng(prevState)
```

CPM Phase Tree

In this section...

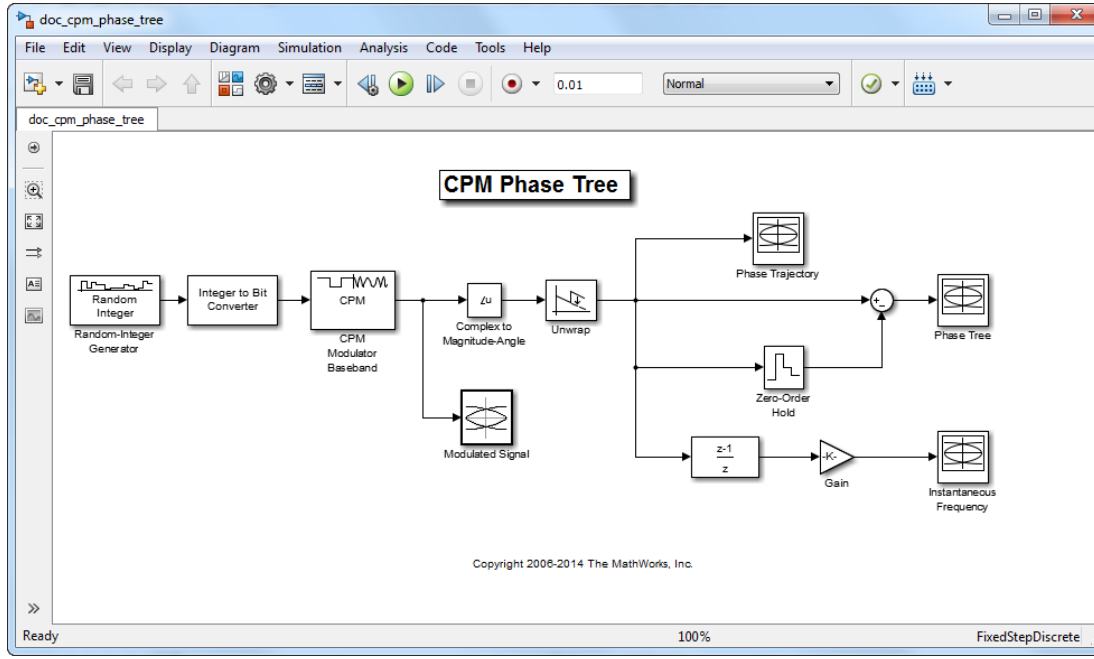
This model shows how to use the Eye Diagram block to view the phase trajectory, phase tree, and instantaneous frequency of a CPM modulated signal.

Structure of the Example

This example, doc_cpm_phase_tree, uses various Communications System 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

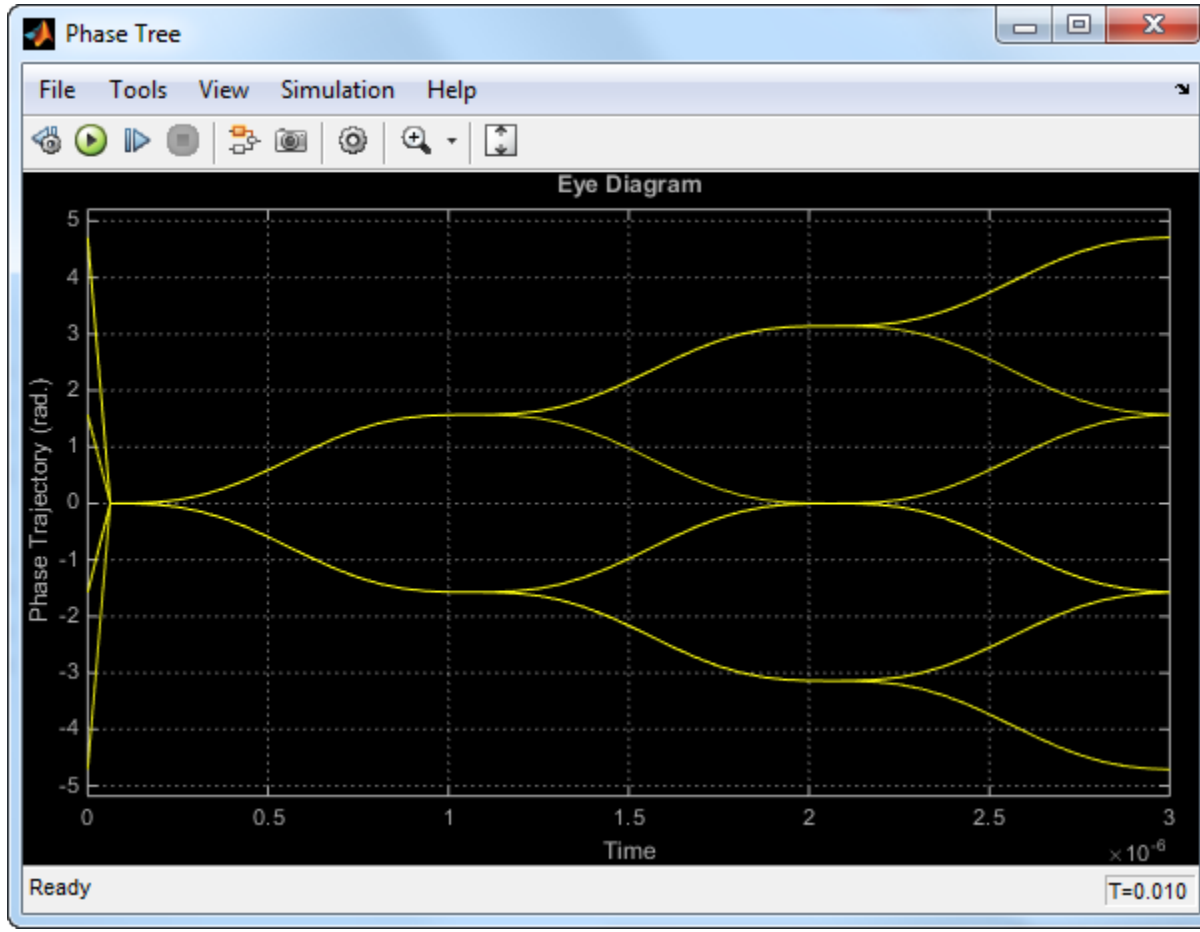


Results and Displays

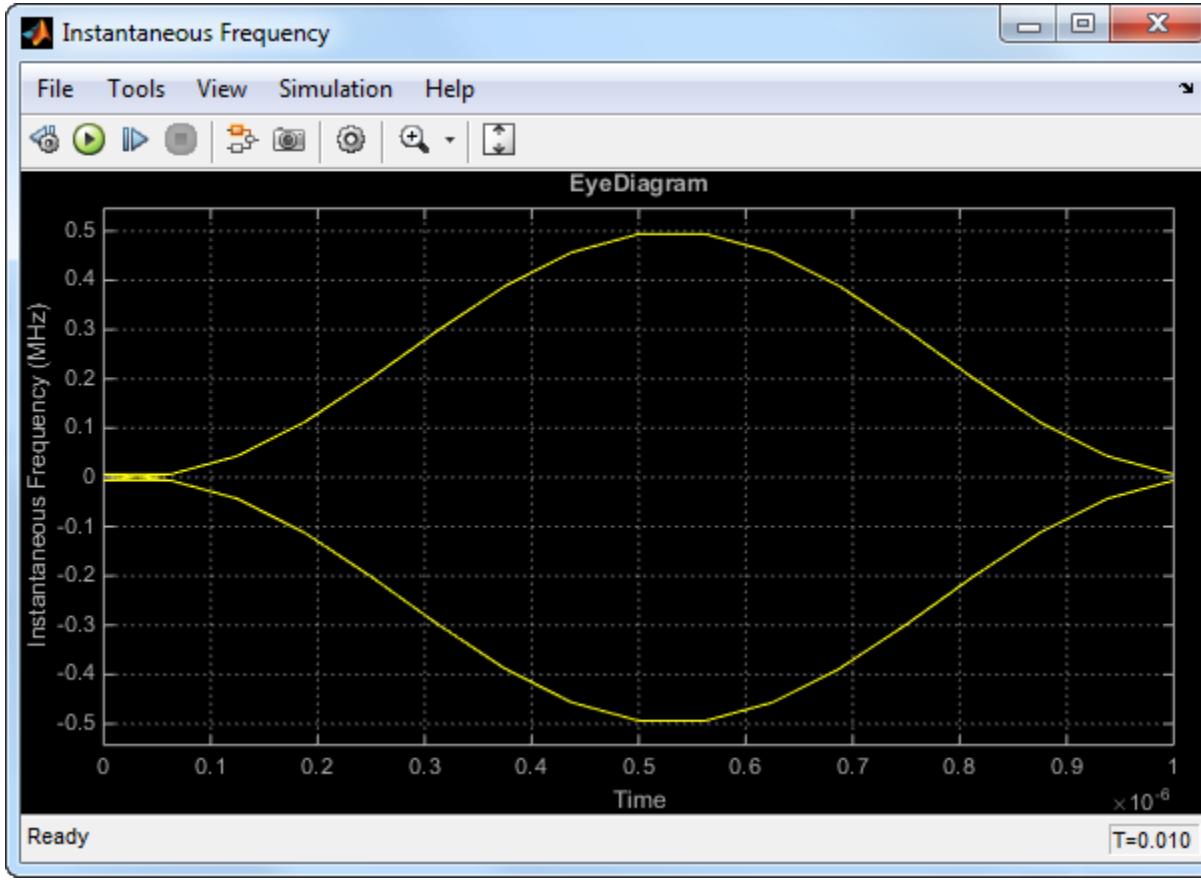
When you run the example, several Eye Diagram blocks show how the CPM signal changes over time:

- 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...

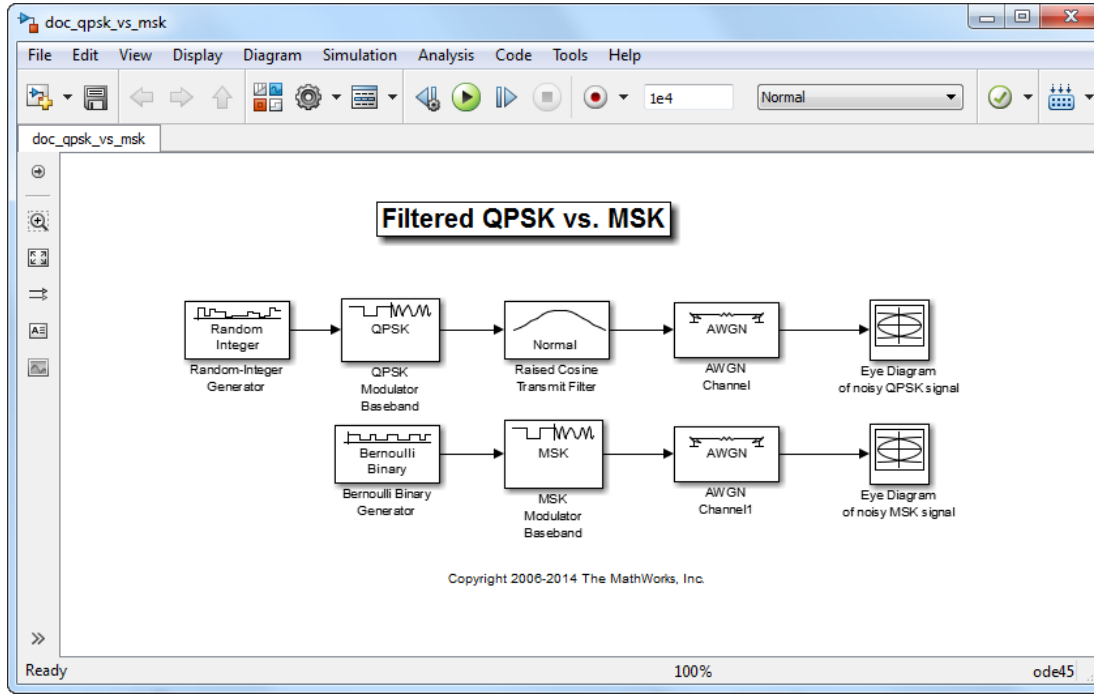
This model shows filtered quadrature phase shift keying (QPSK) and minimum shift keying (MSK) modulation schemes and visually compare them.

Structure of the Example

This example, doc_qpsk_vs_msk, uses Communications System Toolbox blocks to model filtered QPSK and MSK modulation.

The example model includes these blocks:

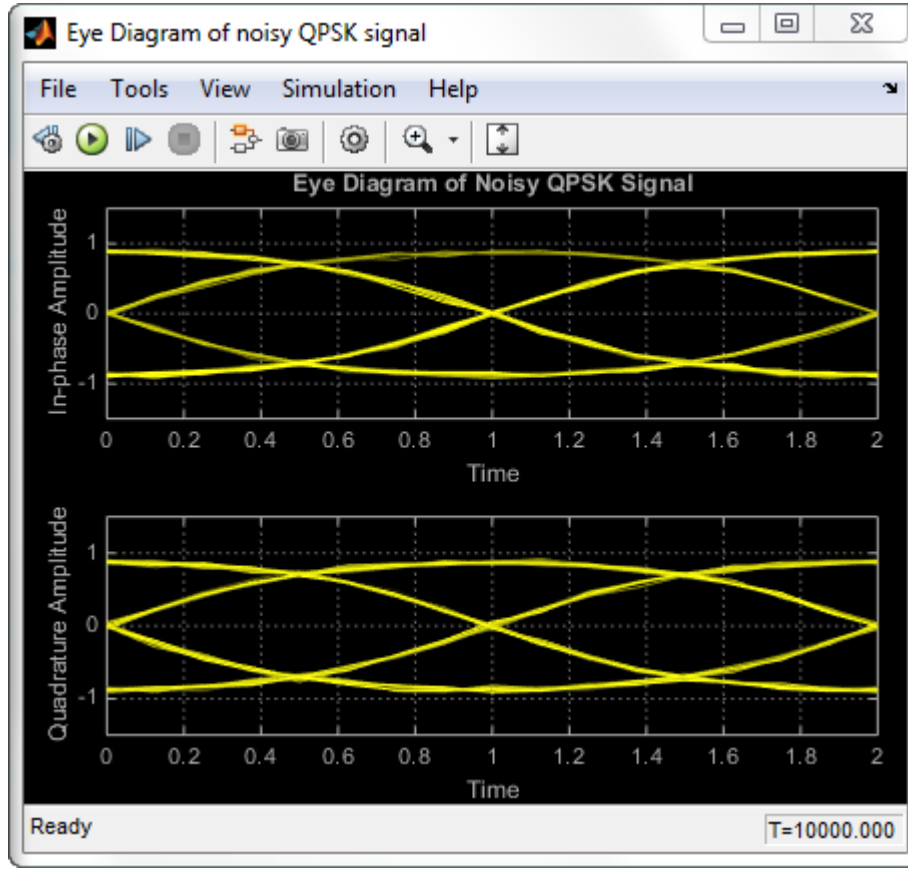
- Random Integer Generator block (for QPSK)
- Bernoulli Binary Generator block (for MSK)
- QPSK Modulator Baseband block
- MSK Modulator Baseband block
- Raised Cosine Transmit Filter block
- AWGN Channel block
- Eye Diagram block



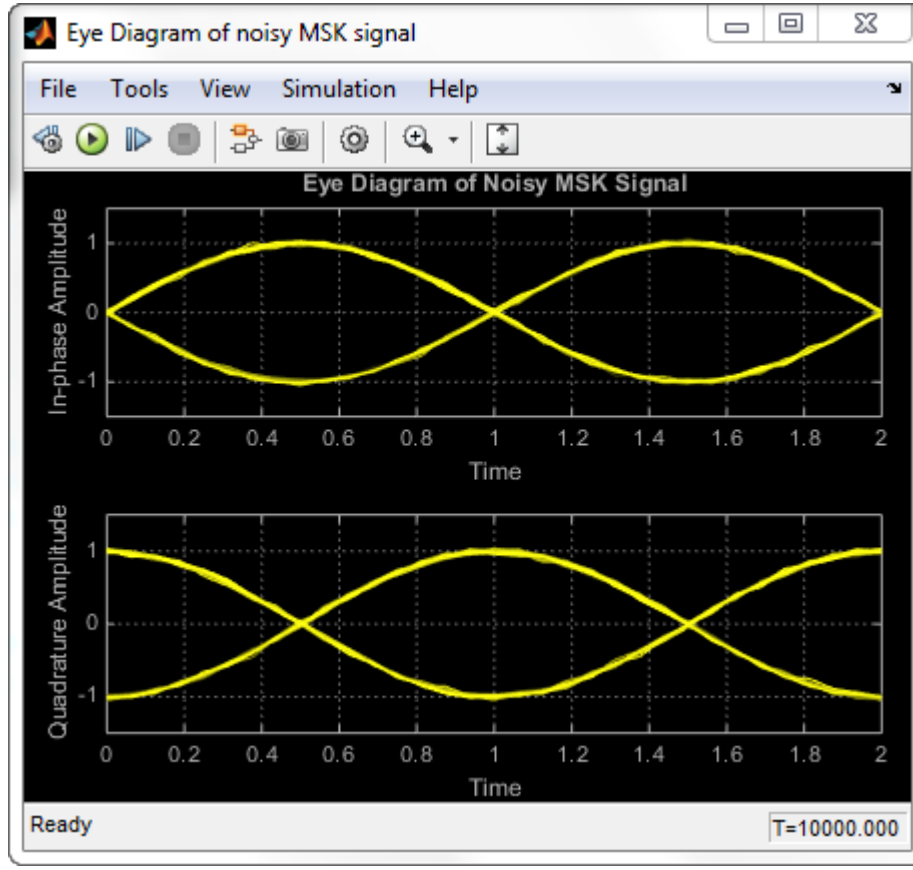
Results and Displays

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,..., while the ideal sampling period for MSK is 0.5, 1.5, 2.5,... for the in-phase signal and 1, 2, 3,... for the quadrature signal.


Eye Diagram of noisy MSK signal
File
Tools
View
Simulation
Help
Eye Diagram of Noisy MSK Signal
In-phase Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Quadrature Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Ready
T=10000.000

GMSK vs. MSK

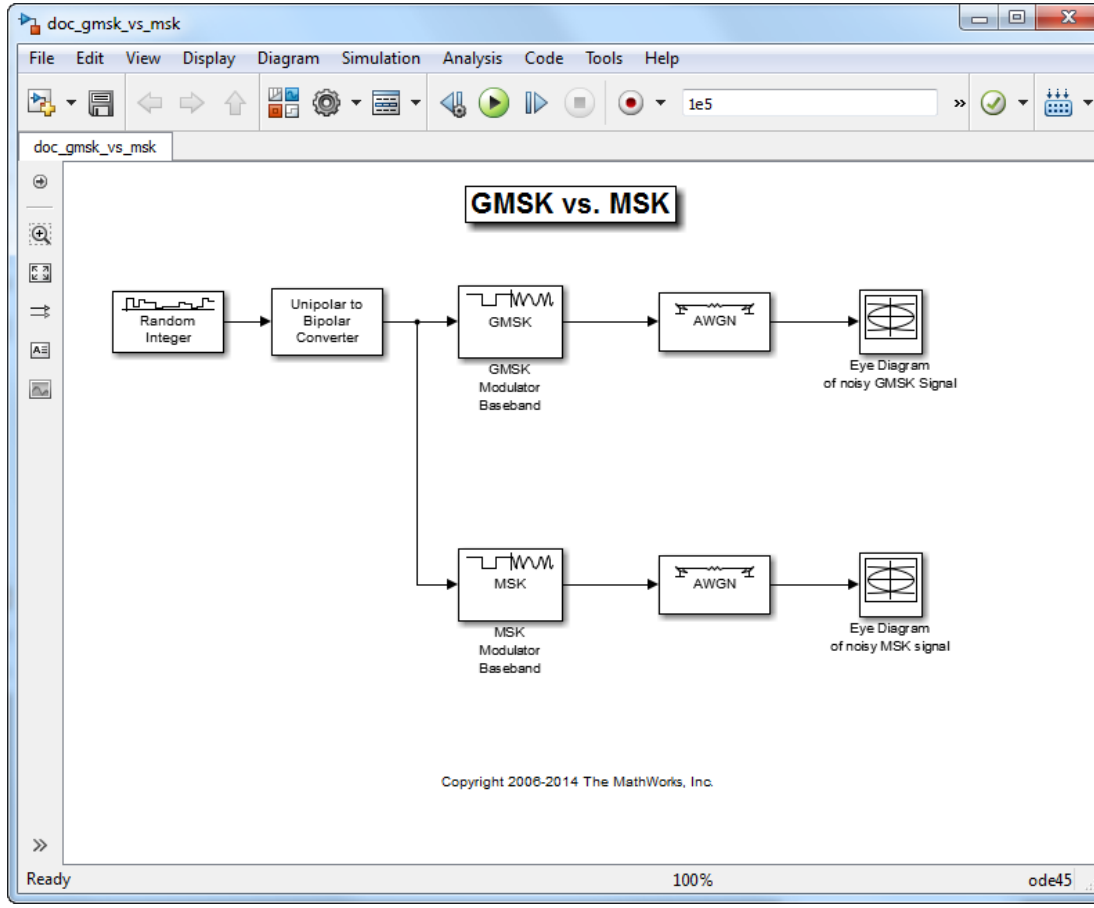
In this section...

This model shows how to visually compare Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes.

Structure of the Example

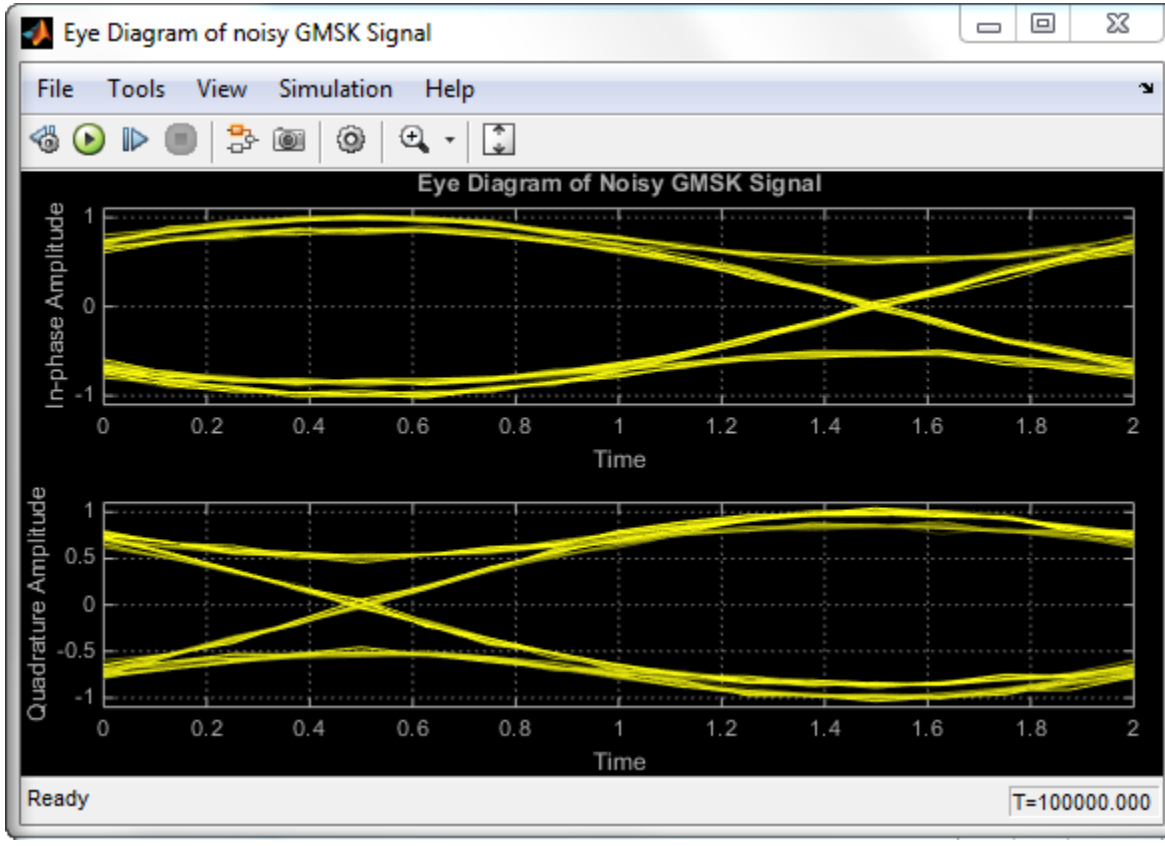
The example model, doc_gmsk_vs_msk, includes these blocks:

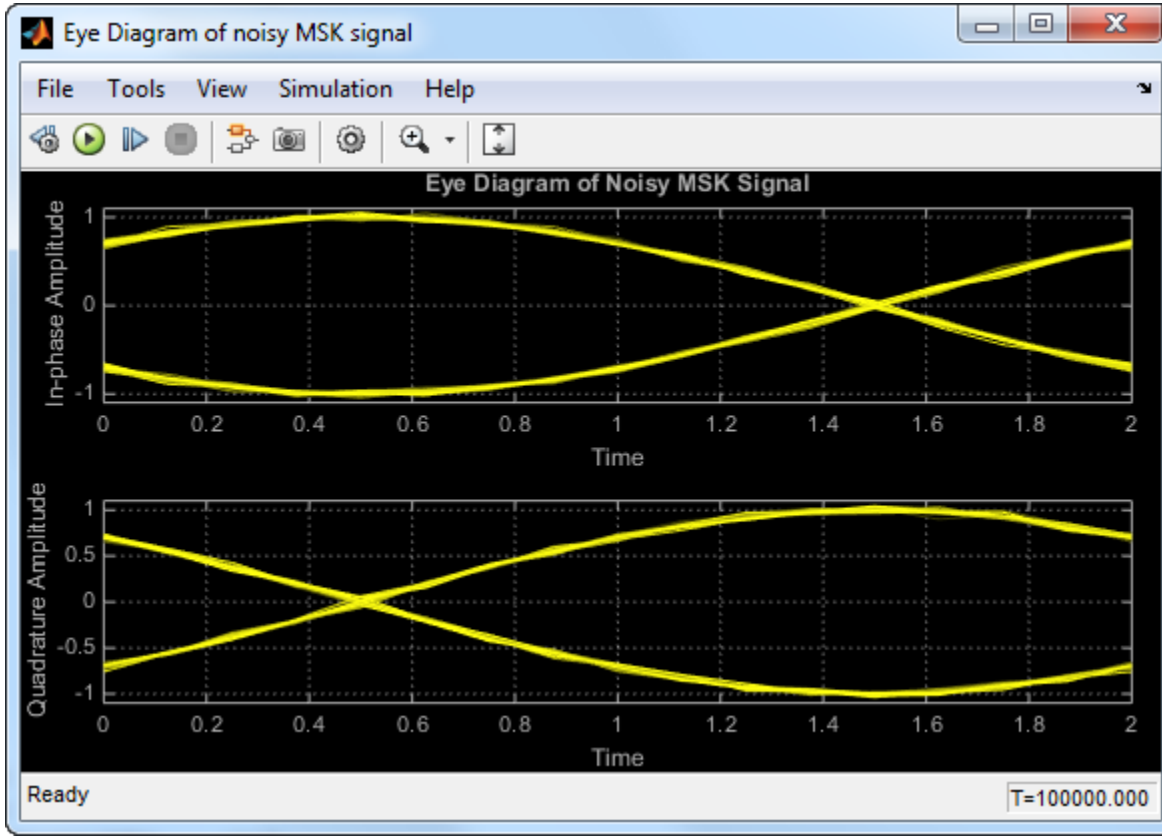
- Random Integer Generator block, which provides a source of uniformly distributed random integers in the range [0, M-1], where M is the constellation size of the GMSK or MSK signal
- Unipolar to Bipolar Converter block
- GMSK Modulator Baseband block
- MSK Modulator Baseband block
- AWGN Channel block
- 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.


Eye Diagram of noisy GMSK Signal
File Tools View Simulation Help
Eye Diagram of Noisy GMSK Signal
In-phase Amplitude
1
0
-1
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
Time
Quadrature Amplitude
1
0.5
0
-0.5
-1
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
Time
Ready
T=100000.000



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” 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.

```
hGMSK = comm.GMSKModulator('BitInput',true,'PulseLength',1, ...
    'SamplesPerSymbol',sps);
hMSK = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',sps);
```

Modulate the data using the GMSK and MSK modulators.

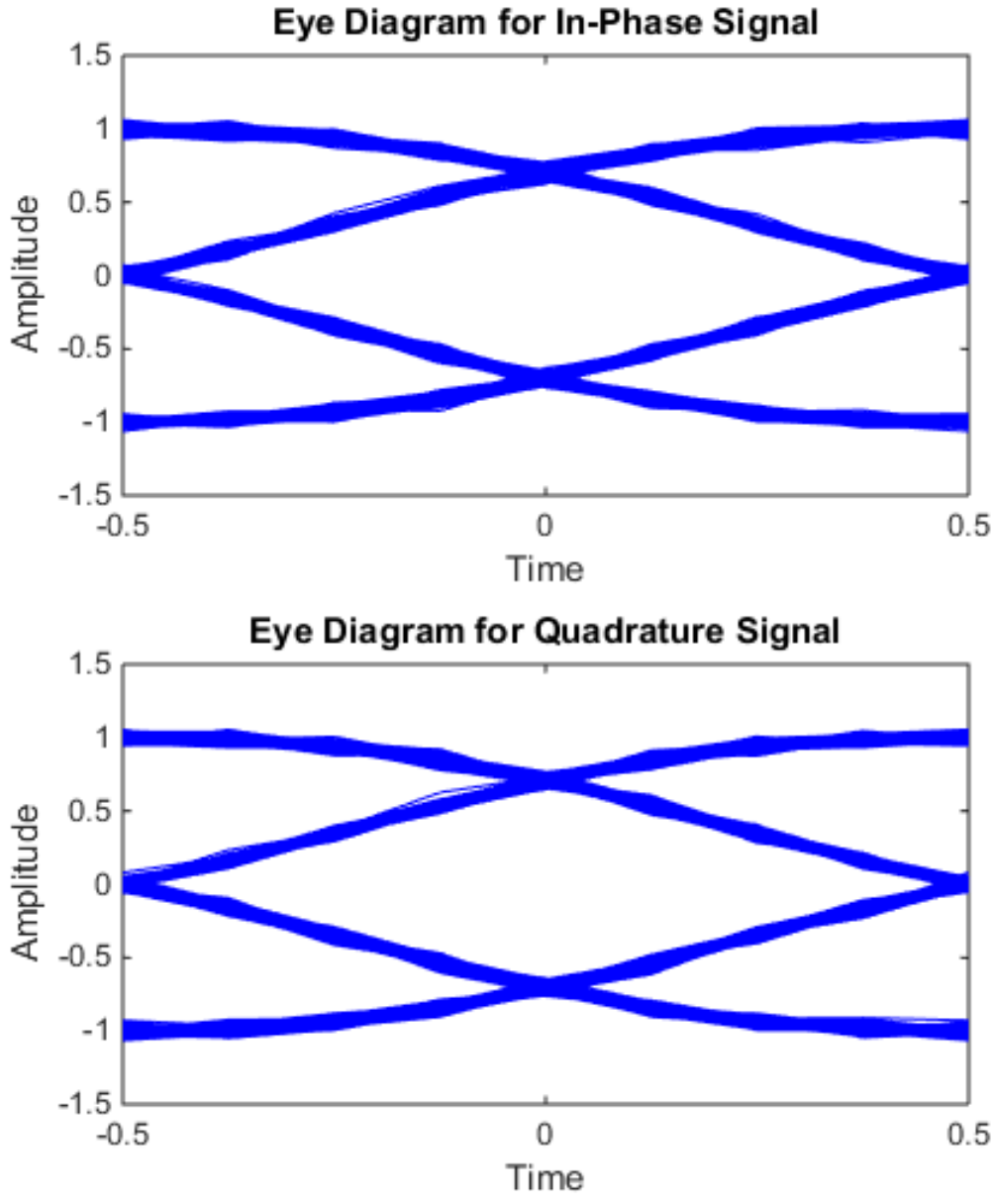
```
modSigGMSK = step(hGMSK,data);
modSigMSK = step(hMSK,data);
```

Pass the modulated signals through an AWGN channel having an SNR of 30 dB.

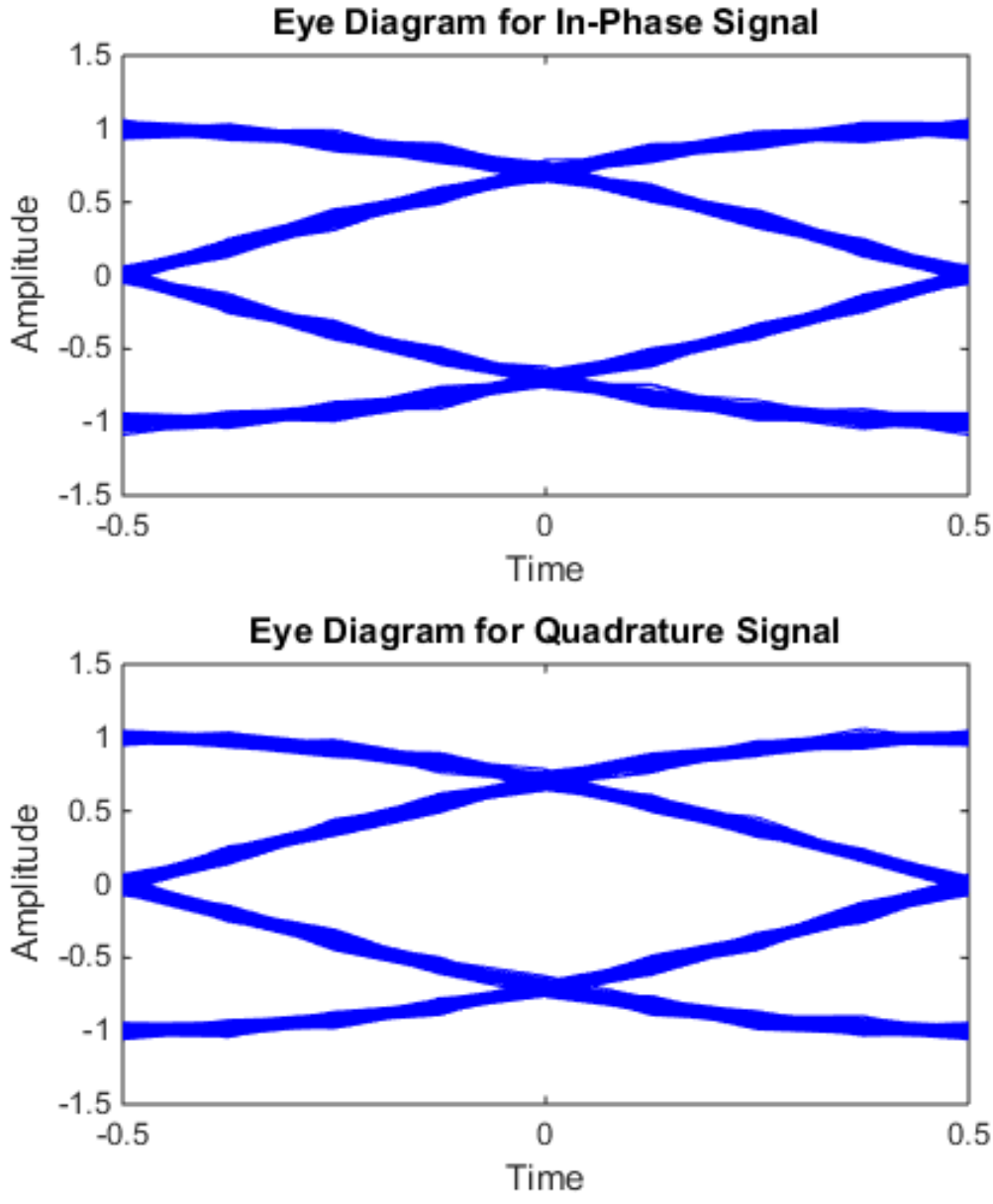
```
rxSigGMSK = awgn(modSigGMSK,30);
rxSigMSK = awgn(modSigMSK,30);
```

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)
```


Eye Diagram for In-Phase Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time
Eye Diagram for Quadrature Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time

```
eyediagram(rxSigMSK,sps,1,sps/2)
```


Eye Diagram for In-Phase Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time
Eye Diagram for Quadrature Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time

Set the `PulseLength` property of `hGMSK` to 3. Because the property is nontunable, the object must be released first.

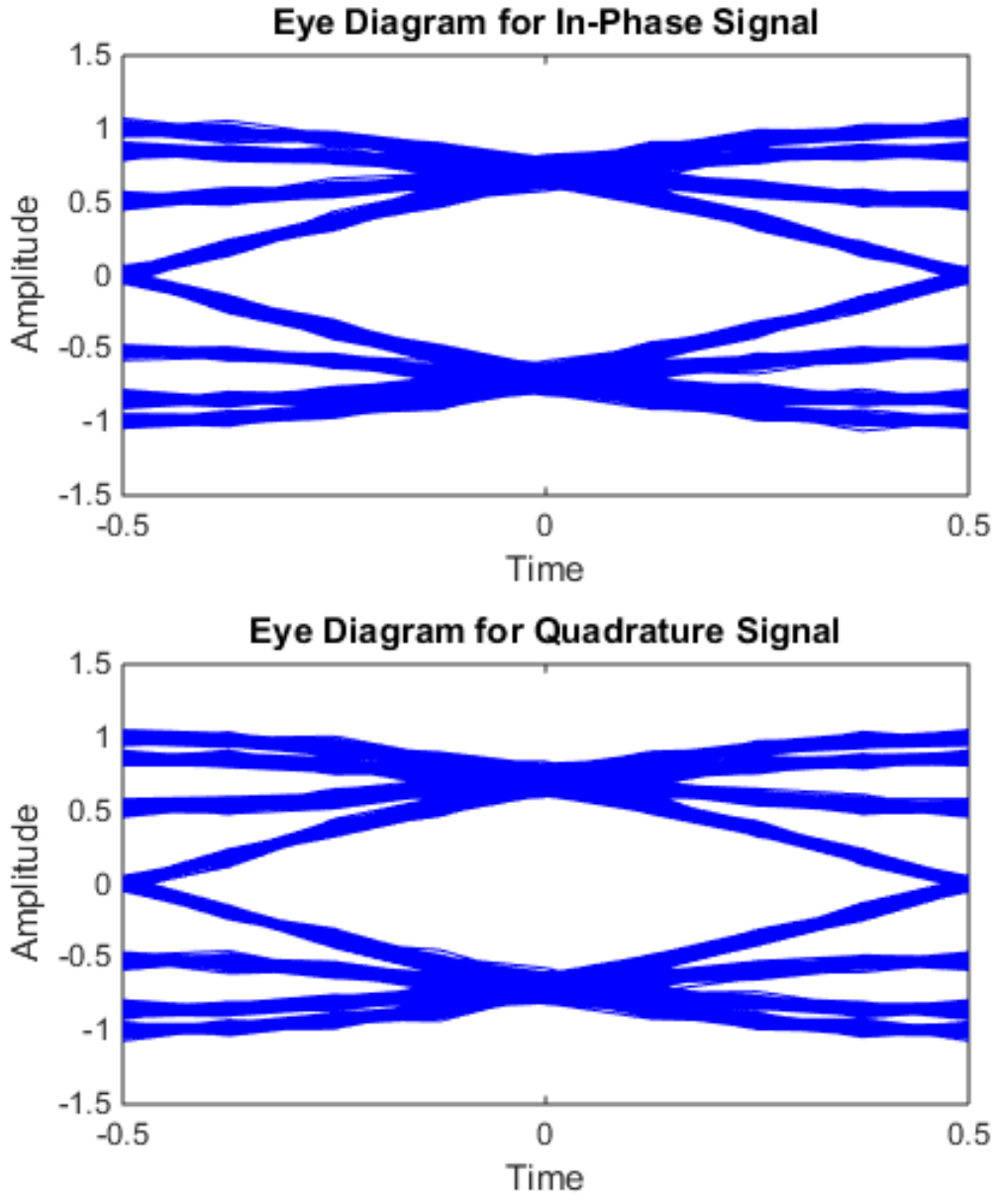
```
release(hGMSK)
hGMSK.PulseLength = 3;
```

Generate a modulated signal using the updated GMSK modulator object and pass it through the AWGN channel.

```
modSigGMSK = step(hGMSK,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)
```


Eye Diagram for In-Phase Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time
Eye Diagram for Quadrature Signal
Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Time

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 the Gaussian pulse shape, you must use scopes that display the phase of the signal, as described in the CPM Phase Tree example.

Gray Coded 8-PSK

In this section...

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

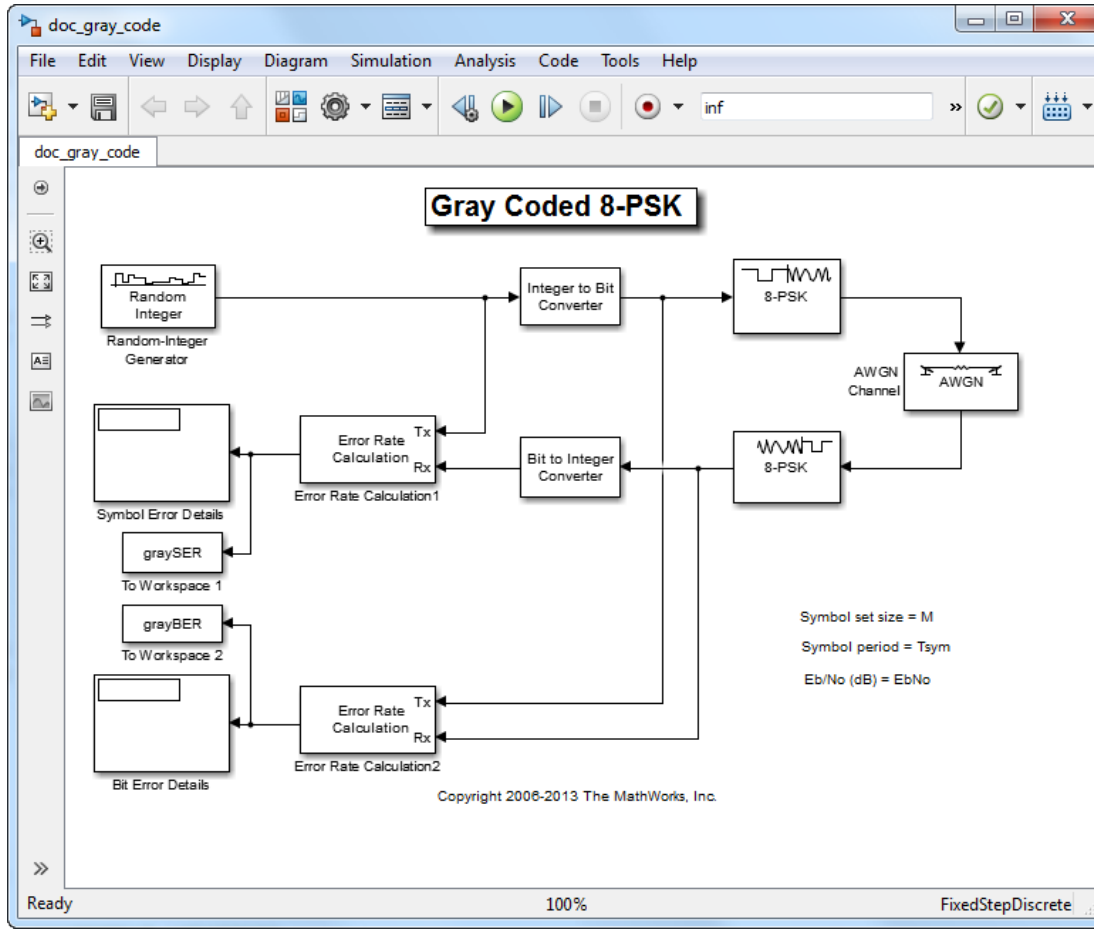
The example model includes these blocks:

- 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 $2\pi(M - 1)/M$

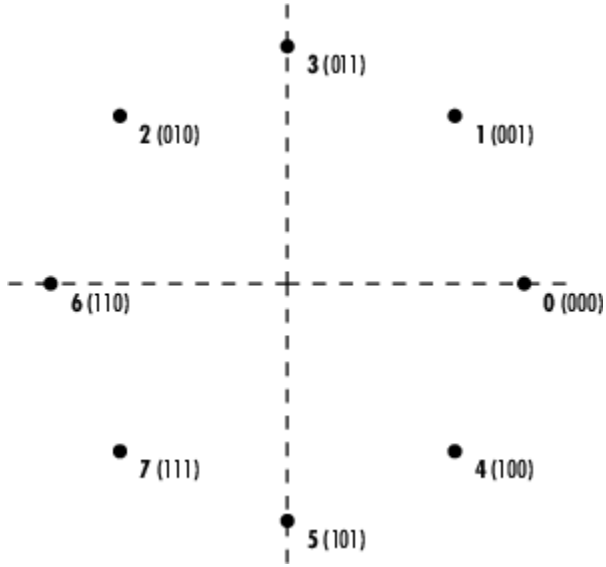


The table indicates which binary representations in the input correspond to which phasors in the output. The second column of the table is an intermediate representation that the block uses in its computations.

Modulator Input	Gray-Coded Ordering	Modulator Output
000	0	$\exp(0) = 1$
001	1	$\exp(j\pi/4)$
010	3	$\exp(j3\pi/4)$
011	2	$\exp(j2\pi/4) = \exp(j\pi/2)$
100	7	$\exp(j7\pi/4)$
101	6	$\exp(j6\pi/4) = \exp(j3\pi/2)$
110	4	$\exp(j4\pi/4) = \exp(j\pi)$
111	5	$\exp(j5\pi/4)$

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
$\exp(j2\pi/4) = \exp(j\pi/2)$	011
$\exp(j3\pi/4)$	010
$\exp(j4\pi/4) = \exp(j\pi)$	110
$\exp(j5\pi/4)$	111
$\exp(j6\pi/4) = \exp(j3\pi/2)$	101
$\exp(j7\pi/4)$	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

$$P_E(M) = \mathrm{erfc}\left(\sqrt{\frac{E_s}{N_0}}\sin\left(\frac{\pi}{M}\right)\right)$$

where `erfc` is the complementary error function, E_s/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

$$\frac{P_E(M)}{\log_2 M} \le P_b \le \frac{M/2}{M-1}P_E(M)$$

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".

The rest of this section studies both the bit and symbol error rates and hence does not use `bertool`.

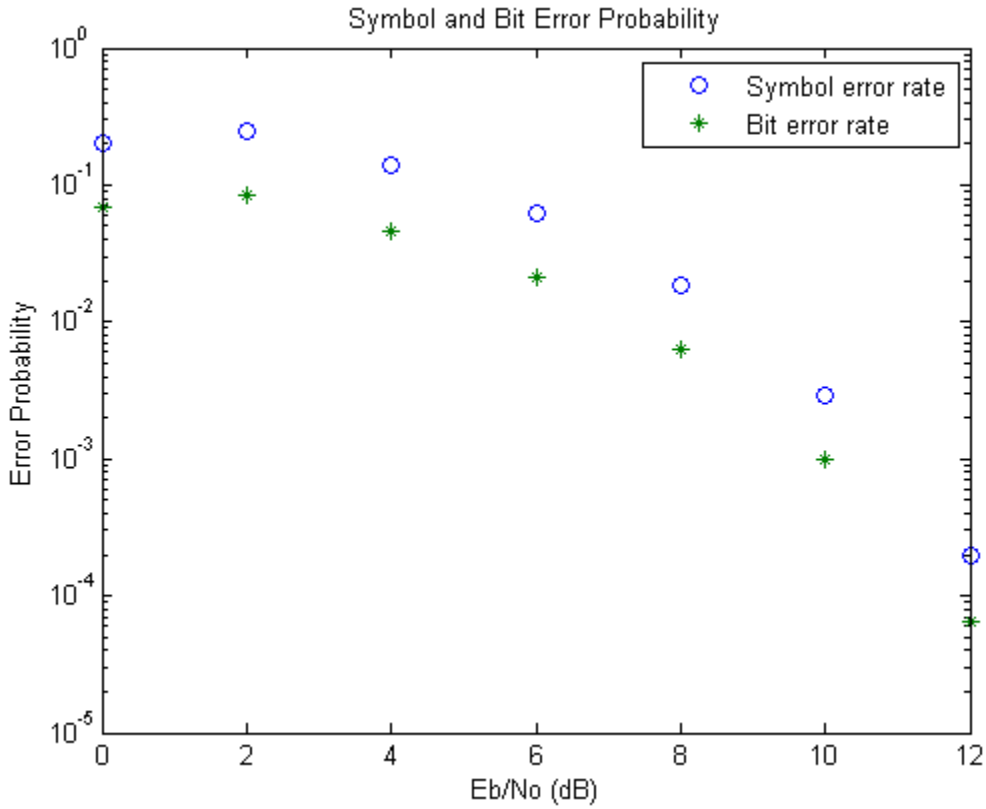
Because increasing the value of E_b/N_0 lowers the number of errors produced, the length of each simulation must be increased to ensure that the statistics of the errors remain stable.

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

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.

In this section...

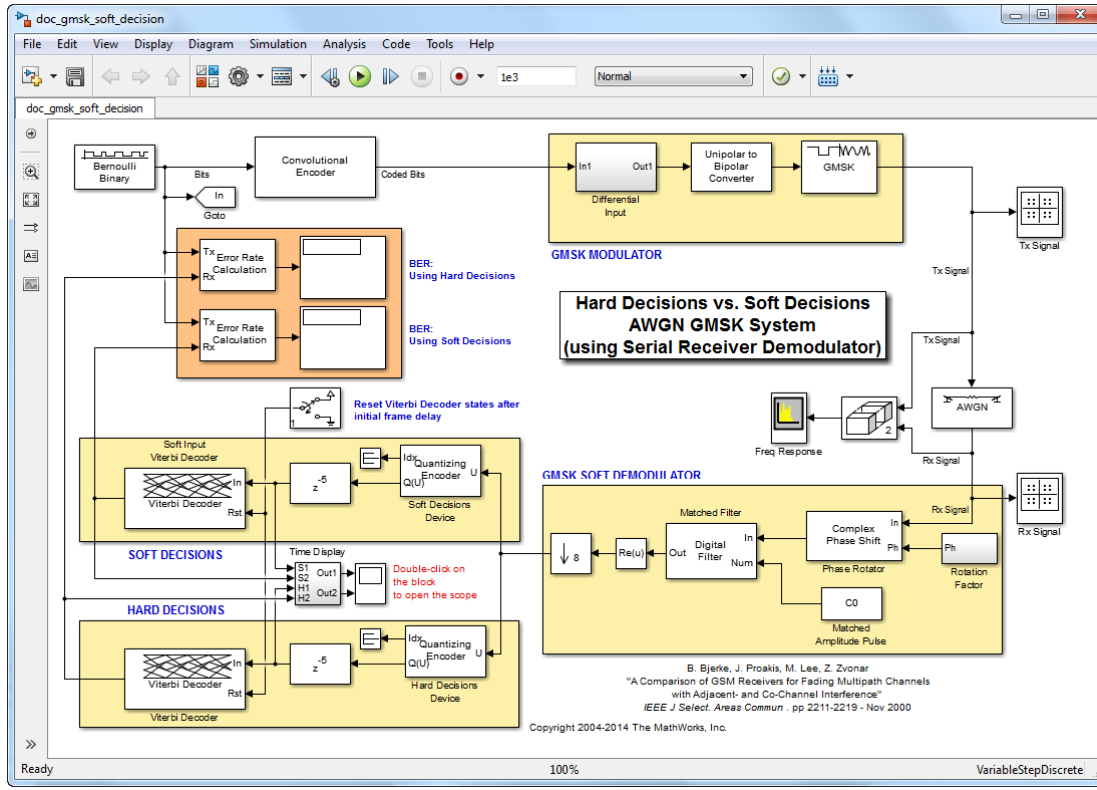
Structure of the Example

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 soft-decision 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.



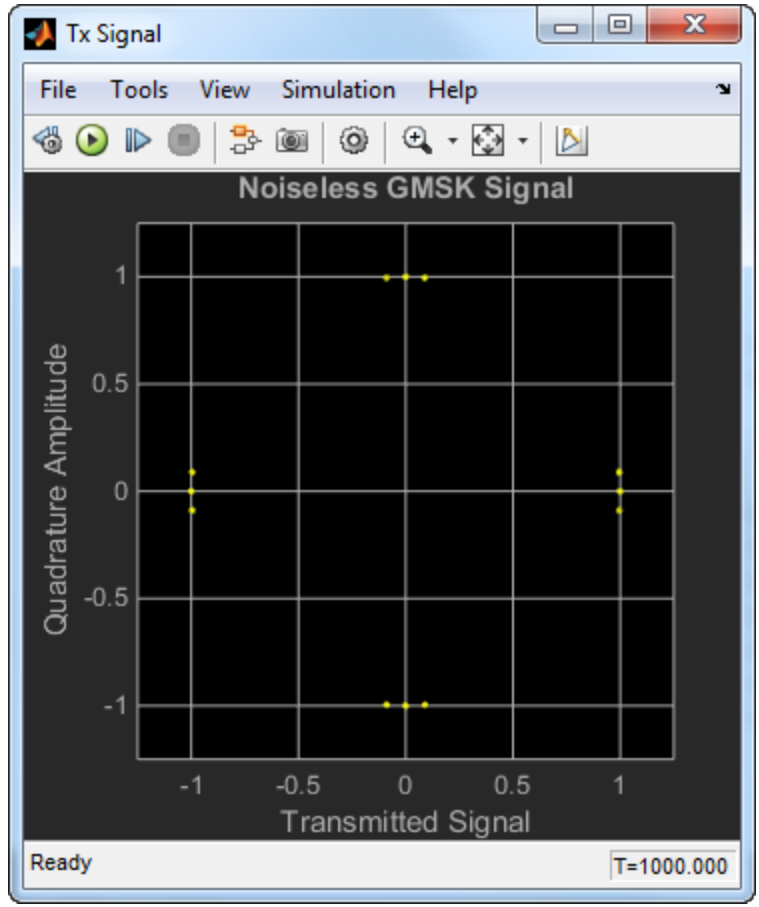
The Serial GMSK Receiver

The serial GMSK receiver is based on the fact that GMSK can be represented as a combination of amplitude pulses [2] - [3], and can, therefore, be demodulated with a matched filter. The GMSK waveform used in this model has a BT product of 0.3 and a frequency pulse length of 4 symbols. As such, it can be represented by eight different amplitude pulses, which are shown in Figure 2 of [3]. The matched filter in this model uses only the largest pulse of the eight, because of its simplicity of implementation. That same simplicity, however, yields BER performance that is inferior to the more traditional Viterbi-based demodulator.

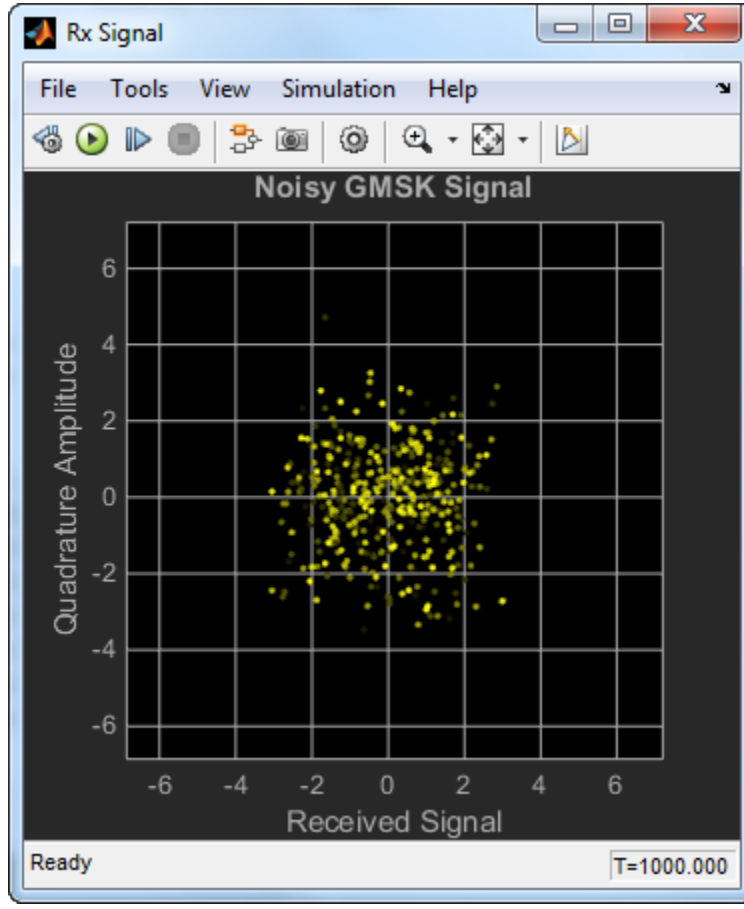
Results and Displays

The example model includes these visualizations to illustrate its performance:

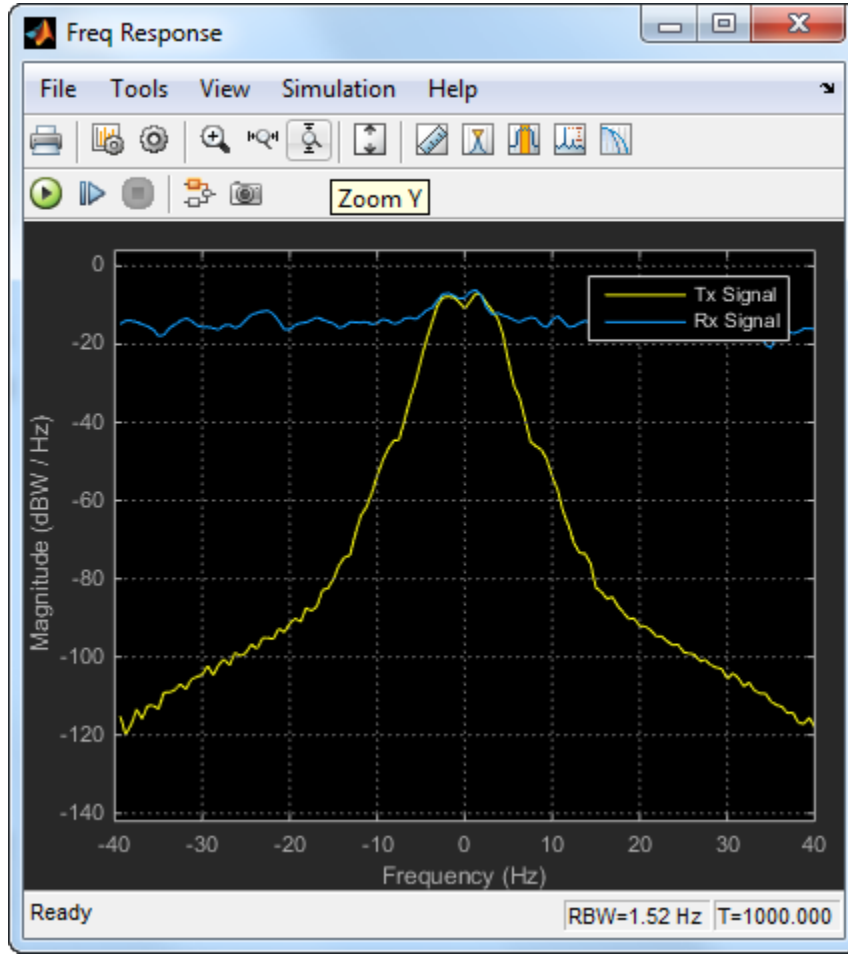
- The Display blocks illustrate that the soft decision receiver performs better (that is, has a smaller BER) than the hard decision receiver.
- The Tx Signal window shows the scatter plot of the signal before the AWGN channel.



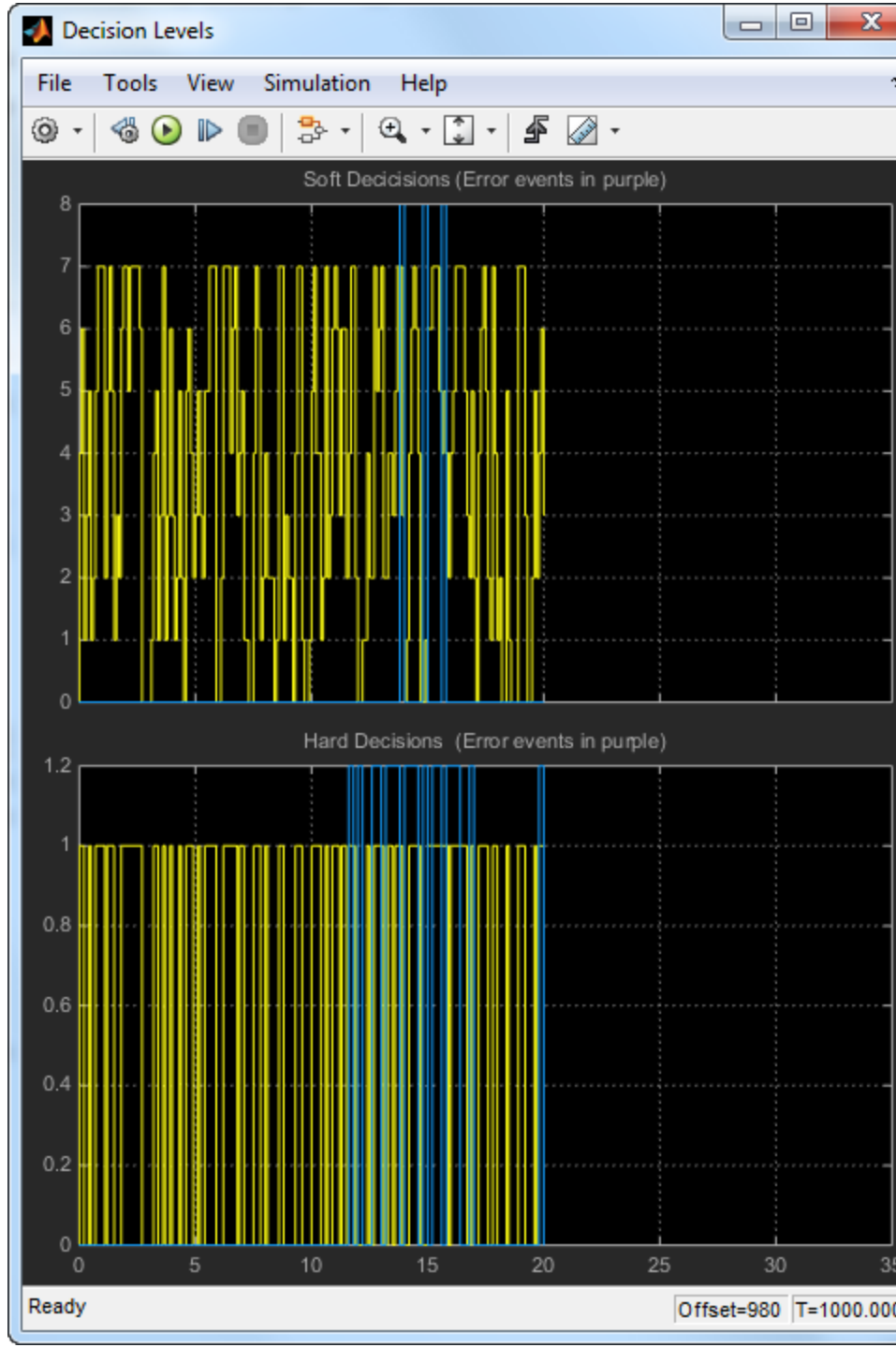
- The Rx Signal window shows the scatter plot of the signal after the AWGN channel.



- The Freq Response window shows the frequency response of the GMSK signal before and after the AWGN channel.



- The Decision Levels window shows, in yellow, the various soft decision levels in the top plot and the binary hard decisions in the bottom plot. This window also indicates, in blue, when errors occur.


Decision Levels
File Tools View Simulation Help
Soft Decicisions (Error events in purple)
8
7
6
5
4
3
2
1
0
Hard Decisions (Error events in purple)
1.2
1
0.8
0.6
0.4
0.2
0
0 5 10 15 20 25 30 35
Ready
Offset=980
T=1000.000

References

[1] Bjerke, B., J. Proakis, M. Lee, and Z. Zvonar, "A Comparison of GSM Receivers for Fading Multipath Channels with Adjacent- and Co-Channel Interference," IEEE J. Select. Areas Commun., Nov. 2000, pp. 2211-2219.

[2] Laurent, Pierre, "Exact and Approximate Construction of Digital Phase Modulations by Superposition of Amplitude Modulated Pulses (AMP)," IEEE Trans. Comm., Vol. COM-34, No. 2, Feb. 1986, pp. 150-160.

[3] Jung, Peter, "Laurent's Representation of Binary Digital Continuous Phase Modulated Signals with Modulation index 1/2 Revisited", IEEE Trans. Comm., Vol. COM-42, No. 2/3/4, Feb./Mar./Apr. 1994, pp. 221-224.

16-PSK with Custom Symbol Mapping

Create 16-PSK modulator and demodulator System objects™ in which custom symbol mapping is used. Estimate the BER in an AWGN channel and compare the performance with that of a theoretical Gray-coded PSK system.

Create a custom symbol mapping for the 16-PSK modulation scheme. The 16 integer symbols must have values which fall between 0 and 15.

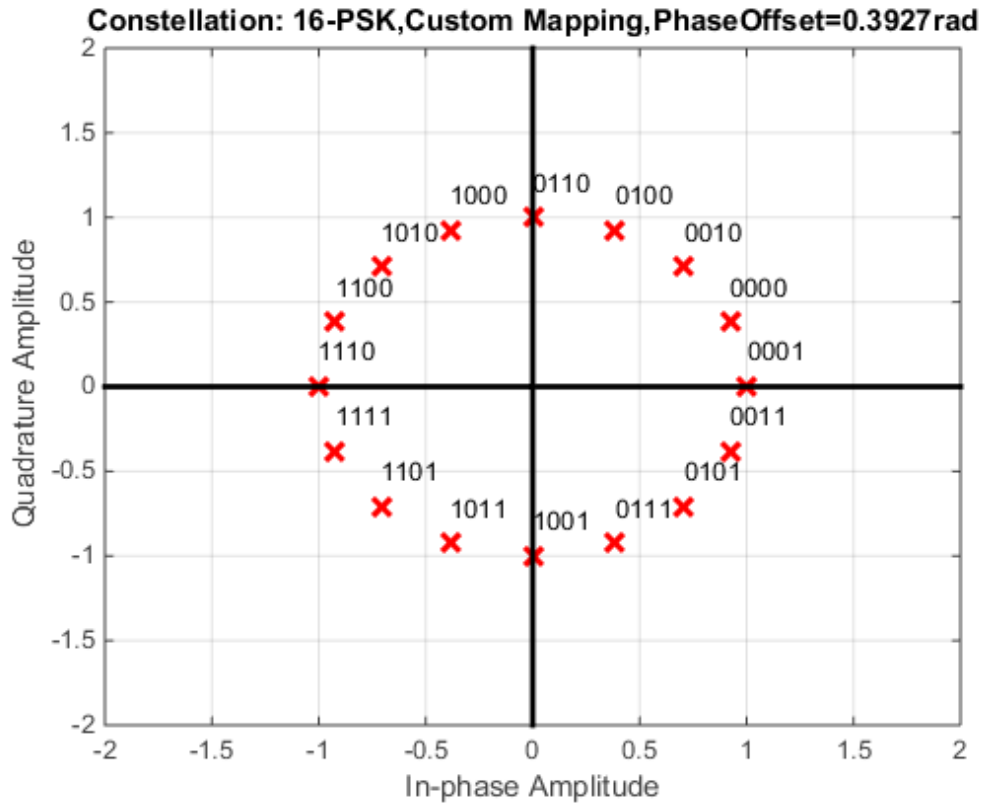
```
custMap = [0 2 4 6 8 10 12 14 15 13 11 9 7 5 3 1];
```

Create a 16-PSK modulator and demodulator pair having custom symbol mapping defined by the array, `custMap`.

```
hMod = comm.PSKModulator(16,'BitInput',true, ...
    'SymbolMapping','Custom', ...
    'CustomSymbolMapping',custMap);
hDemod = comm.PSKDemodulator(16,'BitOutput',true, ...
    'SymbolMapping','Custom', ...
    'CustomSymbolMapping',custMap);
```

Display the modulator constellation.

```
constellation(hMod)
```



Create an AWGN channel System object for use with 16-ary data.

```
hChan = comm.AWGNChannel('BitsPerSymbol',log2(16));
```

Create an error rate object to track the BER statistics.

```
hErr = comm.ErrorRate;
```

Initialize the simulation vectors. The Eb/No is varied from 6 to 18 dB in 1 dB steps.

```
ebnoVec = 6:18;
ber = zeros(size(ebnoVec));
```

Estimate the BER by modulating binary data, passing it through an AWGN channel, demodulating the received signal, and collecting the error statistics.

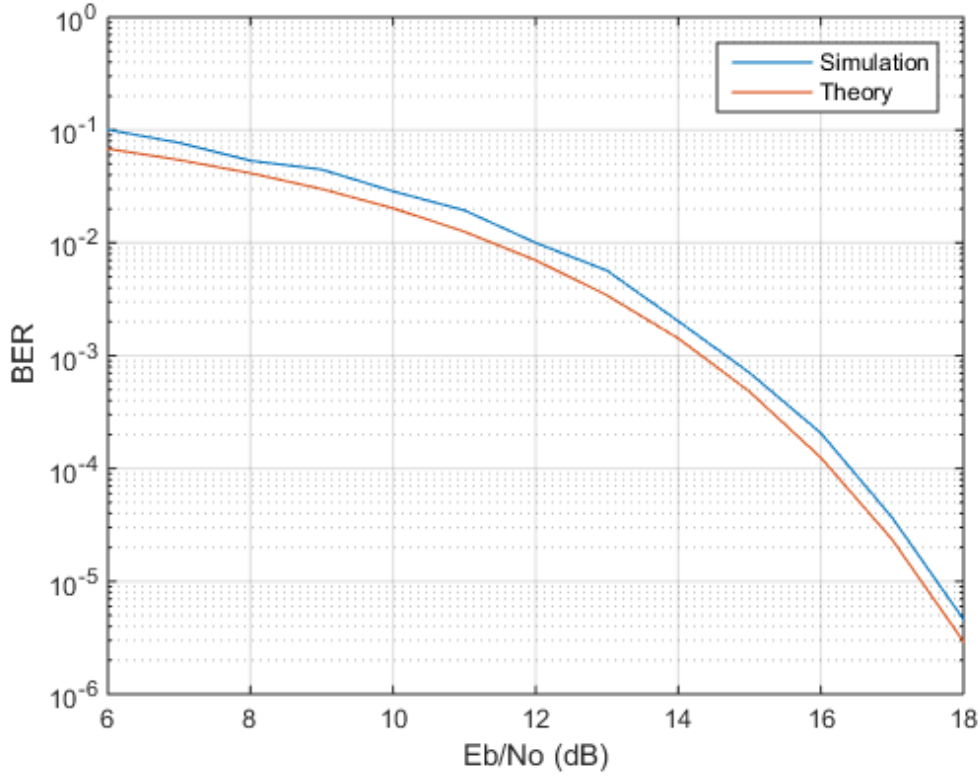
```
for k = 1:length(ebnoVec)

    % Reset the error counter for each Eb/No value
    reset(hErr)
    % Reset the array used to collect the error statistics
    errVec = [0 0 0];
    % Set the channel Eb/No
    hChan.EbNo = ebnoVec(k);

    while errVec(2) < 200 && errVec(3) < 1e7
        % Generate a 1000-symbol frame
        data = randi([0 1],4000,1);
        % Modulate the binary data
        modData = step(hMod,data);
        % Pass the modulated data through the AWGN channel
        rxSig = step(hChan,modData);
        % Demodulate the received signal
        rxData = step(hDemod,rxSig);
        % Collect the error statistics
        errVec = step(hErr,data,rxData);
    end

    % Save the BER data
    ber(k) = errVec(1);
end
```

Generate theoretical BER data for an AWGN channel using `berawgn`.

```
berTheory = berawgn(ebnoVec,'psk',16,'nondiff');
```

Plot the simulated and theoretical results. Because the simulated results rely on 16-PSK modulation that does not use Gray codes, the performance is not as good as that predicted by theory.

```
figure
semilogy(ebnoVec,[ber; berTheory])
xlabel('Eb/No (dB)')
ylabel('BER')
grid
legend('Simulation','Theory','location','ne')
```


Simulation
Theory
BER
Eb/No (dB)
10^0
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
6
8
10
12
14
16
18

General QAM Modulation in an AWGN Channel

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

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

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

Generate random symbols.

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

Modulate the data using the `genqammod` function. It is necessary to use general QAM modulation because the custom constellation is not rectangular.

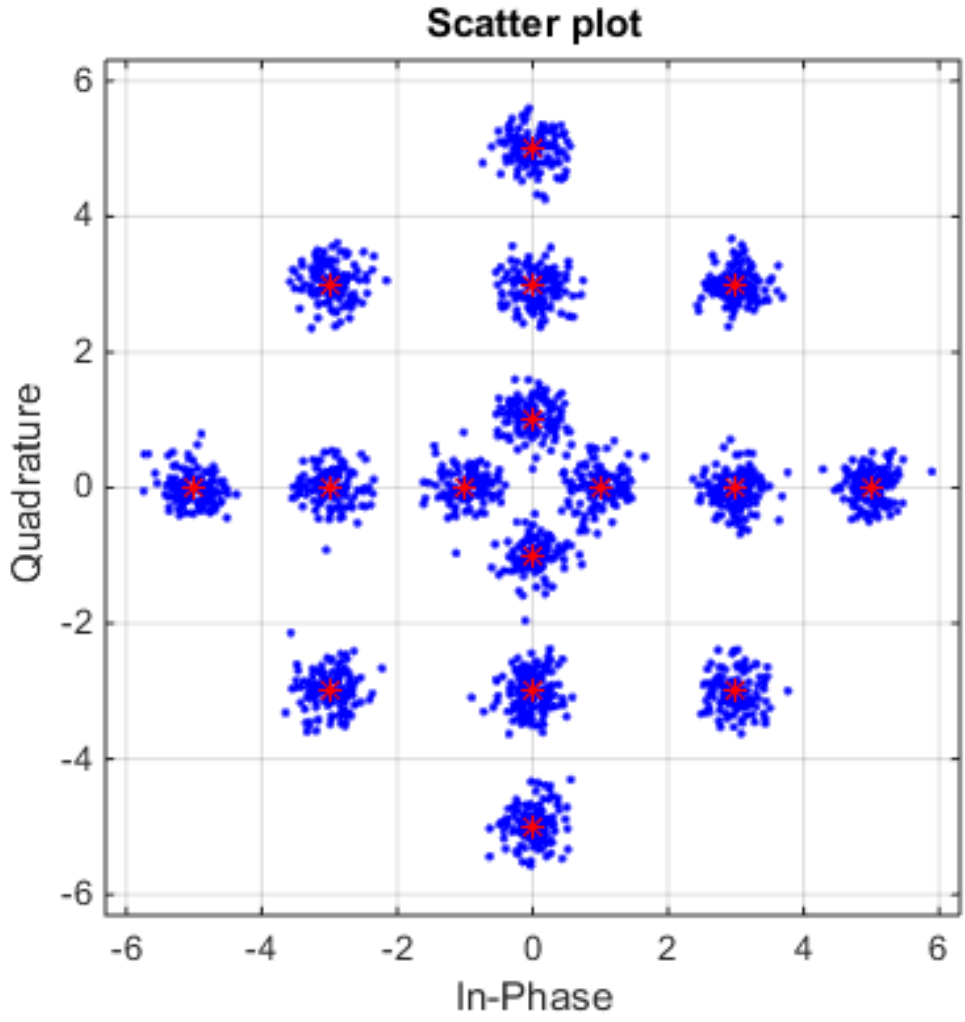
```
modData = genqammod(data,c);
```

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

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

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

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



Demodulate the received signal using the `genqamdemod` function and determine the number of symbol errors and the symbol error ratio.

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

```
numErrors =

     1


ser =

   5.0000e-04
```

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

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

```
numErrors =

   462


ser =

    0.2310
```

MSK

MSK Signal Recovery

This example shows how to model channel impairments such as timing phase offset, carrier frequency offset, and carrier phase offset for a minimum shift keying (MSK) signal. The example also shows the use of System objects to synchronize such signals at the receiver.

Introduction

This example models an MSK transmitted signal undergoing channel impairments such as timing, frequency, and phase offset as well as AWGN noise. `MSKTimingSynchronizer` and `CPMCarrierPhaseSynchronizer` System objects recover the timing and phase offsets, while a delay-and-multiply scheme recovers the carrier frequency offset.

First, initialize system variables. See the MATLAB script configureMSKSignalRecovery for details. Then, define control variables to specify that the system performs timing, carrier frequency, and carrier phase recovery.

```
configureMSKSignalRecovery;
recoverTimingPhase = true;
recoverCarrierFrequency = true;
recoverPhase = true;
```

Modeling Channel Impairments

Specify the sample delay, `timingOffset`, that the channel model applies and create a `VariableFractionalDelay` System object to introduce the timing delay to the transmitted signal.

```
timingOffset = 0.2;
hVFD = dsp.VariableFractionalDelay;
```

Create and configure a phase and frequency offset System object, `hPFO`, to introduce carrier phase and frequency offsets to the transmitted signal. Since the MSK modulator up-samples the transmitted symbols, set the `SampleRate` property appropriately.

```
freqOffset = 50;    % Frequency offset applied by the channel model
phaseOffset = 30;   % Phase offset applied by the channel model
hPFO = comm.PhaseFrequencyOffset('FrequencyOffset', freqOffset, ...
                'PhaseOffset', phaseOffset, ...
                'SampleRate', samplesPerSymbol/Ts);
```

Create an AWGN channel System object to add additive white Gaussian noise to the modulated signal. The noise power is determined by the bit energy to noise power spectral density ratio `EbNo` property. Since the MSK modulator generates symbols with 1 Watt of power, the signal power property of the AWGN channel is also set to 1.

```
EbNo = 60;          % in dB
hAWGN = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)', ...
                'EbNo',EbNo,...
                'SignalPower', 1, ...
                'SamplesPerSymbol',samplesPerSymbol);
```

Timing, Carrier Frequency, and Carrier Phase Synchronization

Construct an MSK timing synchronizer to recover symbol timing phase using a fourth-order nonlinearity method.

```
hTimeSync = comm.MSKTimingSynchronizer('SamplesPerSymbol',samplesPerSymbol, ...
                    'ErrorUpdateGain',0.2);
```

Create and configure a delay, moving average filter, and cumulative sum System object to recover the carrier frequency using a combination of the 2P-power method and the delay-and-multiply scheme.

- `hDelayInstFreq` - delay
- `hMovAve` - moving average of the carrier frequency shift estimates over 100 frames
- `hCumSum` - cumulative sum of the instantaneous frequency shift estimates

```
hDelayInstFreq = dsp.Delay;
hMovAve = dsp.DigitalFilter('TransferFunction','FIR (all zeros)', ...
              'Numerator', ones(1,100)/100);
hCumSum = dsp.CumulativeSum('Dimension','Channels (running sum)', ...
              'FrameBasedProcessing', true);
```

Create and configure a CPM carrier phase synchronizer object, `hPhaseSync`, to recover carrier phase using the 2P-Power method. This phase synchronizer is suitable for use with MSK modulated signals when the denominator of the modulation index property, `P`, is set to 2, which corresponds to MSK modulation.

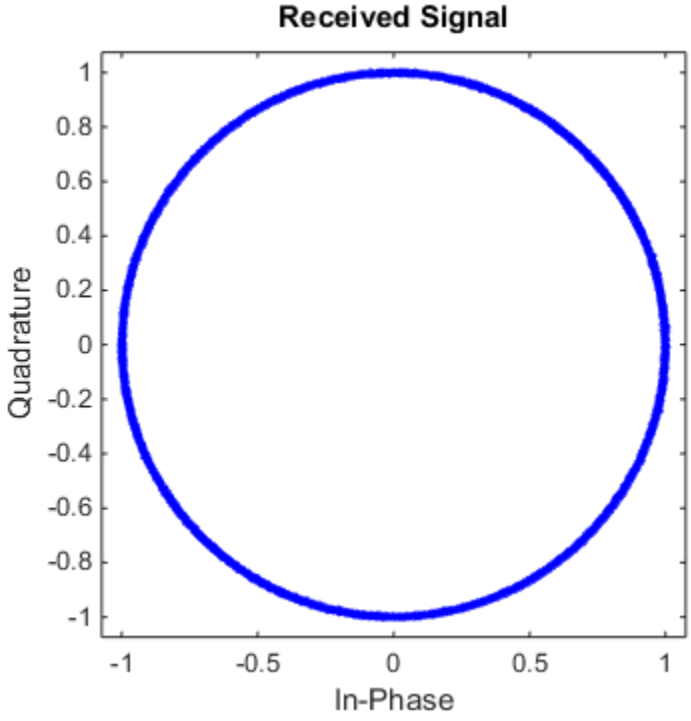
```
hPhaseSync = comm.CPMCarrierPhaseSynchronizer('P',2, ...
                     'ObservationInterval',1000);
```

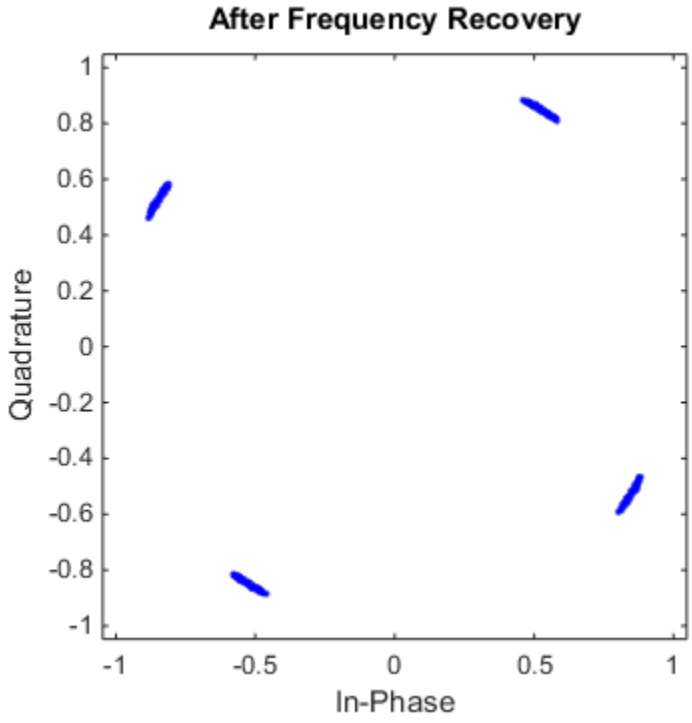
Stream Processing Loop

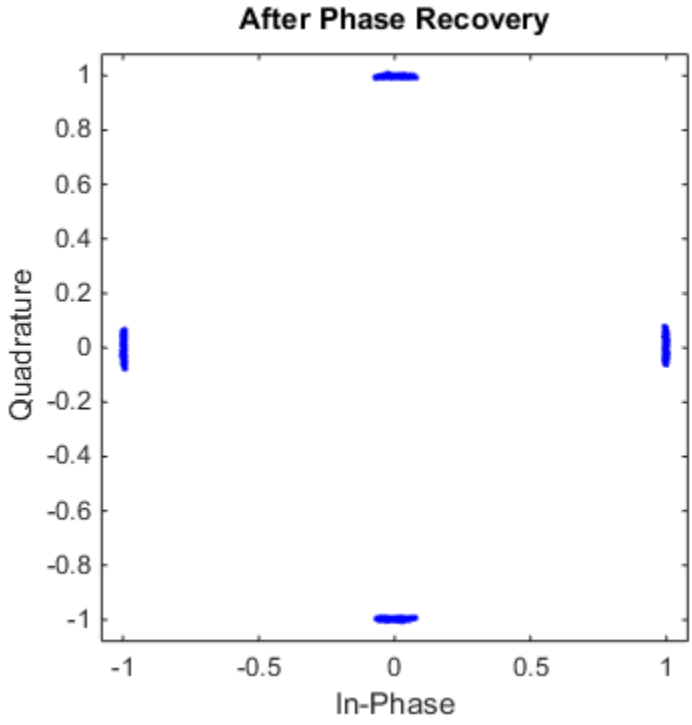
The system modulates data using MSK modulation. The modulated symbols pass through the channel model, which applies timing delay, carrier frequency and phase shift, and additive white Gaussian noise. In this system, the receiver performs timing, carrier frequency, and carrier phase recovery. Finally, the system demodulates the symbols and calculates the bit error rate using an error rate calculator System object. The plotResultsMSKSignalRecovery script generates scatter plots at three points in the system: after applying channel impairments, after carrier frequency synchronization, and after carrier phase synchronization. At the end of the simulation, the example displays the timing phase, frequency, and phase estimates as a function of simulation time.

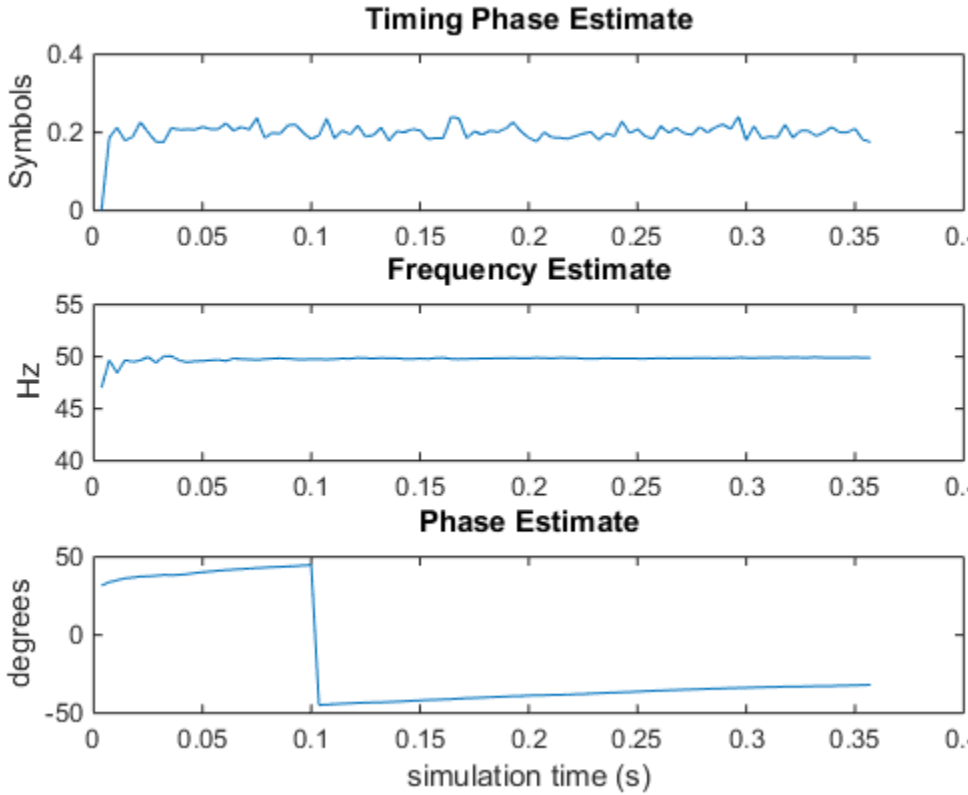
```
for p = 1:numFrames
  %------------------------------------------------------------------------
  % Generate and modulate data
  %------------------------------------------------------------------------
  txBits = randi([0 1], samplesPerFrame, 1);
  txSym = step(hMod, txBits);
  %------------------------------------------------------------------------
  % Transmit through channel
  %------------------------------------------------------------------------
  chanSym = step(hVFD,txSym,timingOffset*samplesPerSymbol);  % Timing offset
  chanSym = step(hPFO,chanSym);         % Carrier frequency and phase offset
  chanSym = step(hAWGN, chanSym);       % Additive white Gaussian noise
  % Save the transmitted signal for plotting
  plot_rx = chanSym;
  %------------------------------------------------------------------------
  % Timing recovery
  %------------------------------------------------------------------------
  if recoverTimingPhase
    % Recover symbol timing phase using fourth-order nonlinearity method
    [rxSym timEst] = step(hTimeSync, chanSym);
    % Calculate the timing delay estimate for each sample
    timEst = timEst(1)/samplesPerSymbol;
  else
    % Do not apply timing recovery and simply downsample the received signal
    rxSym = downsample(chanSym, samplesPerSymbol);
    timEst = 0;
  end
  %------------------------------------------------------------------------
  % Carrier frequency recovery
  %------------------------------------------------------------------------
  if recoverCarrierFrequency
    % The following script applies carrier frequency recovery using a
    % combination of the 2P-power method and the delay-and-multiply scheme
    syncCarrierFreqMSKSignalRecovery;
```

```
  else
    freqShiftEst = 0;
  end
  % Save the carrier frequency synchronized received signal for plotting
  plot_rxFreqSync = rxSym;
  %------------------------------------------------------------------------
  % Carrier phase recovery
  %------------------------------------------------------------------------
  if recoverPhase
    % Recover carrier phase using 2P-Power method
    [rxSym, phEst] = step(hPhaseSync, rxSym);
    % Resolve phase ambiguity
    removePhaseAmbiguityMSKSignalRecovery;
  else
    phEst = 0;
  end
  % Save the phase synchronized received signal for plotting
  plot_rxPhSync = rxSym;
  %------------------------------------------------------------------------
  % Demodulate the received symbols
  %------------------------------------------------------------------------
  rxBits = step(hDemod, rxSym);
  %------------------------------------------------------------------------
  % Calculate the bit error rate
  %------------------------------------------------------------------------
  BER = step(hBER, txBits, rxBits);
  %------------------------------------------------------------------------
  % Plot results
  %------------------------------------------------------------------------
  plotResultsMSKSignalRecovery;
end   %#ok<*UNRCH,*SAGROW,*NOPTS>
```


Received Signal
Quadrature
In-Phase
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1


After Frequency Recovery
Quadrature
In-Phase
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1


After Phase Recovery
Quadrature
In-Phase
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1



Display the bit error rate, `BitErrorRate`, for E_b/N_0 of 60 dB as well as the total number of symbols, `NumberOfSymbols`, processed by the error rate calculator.

```
BitErrorRate = BER(1)
NumberOfSymbols = BER(3)
```

```
BitErrorRate =

     0


NumberOfSymbols =

       99982
```

Conclusion

The constellation plots after timing, carrier frequency, and carrier phase synchronization provide a compelling visual rendition of the recovery algorithms in action. This is especially evident when you turn the synchronization algorithms on and off using the control variables: `recoverTimingPhase`, `recoverCarrierFrequency`, and `recoverPhase`.

Appendix

This example uses the following scripts and helper functions:

- configureMSKSignalRecovery
- plotResultsMSKSignalRecovery
- removePhaseAmbiguityMSKSignalRecovery
- syncCarrierFreqMSKSignalRecovery

MSK Signal Recovery

In this section...

This model shows how channel impairments such as timing phase offset, carrier frequency offset, and phase offset for a minimum shift keying (MSK) signal are modeled. The model uses blocks from the Synchronization library to recover the signal. Open the model, doc_commmsksync.

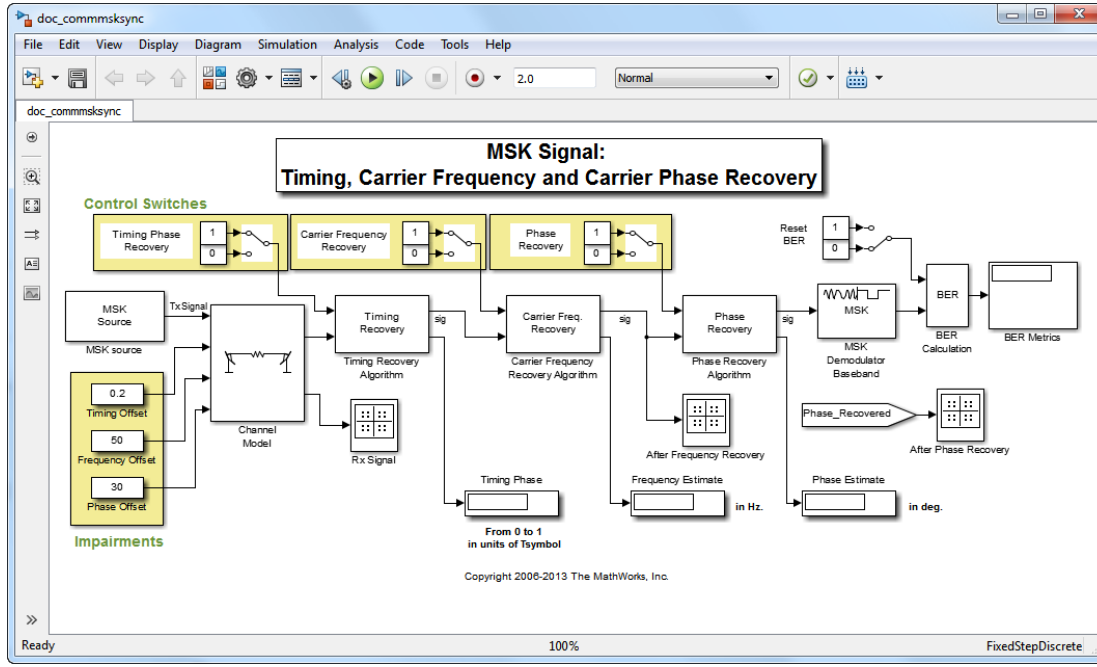
```
doc_commmsksync
```

Exploring the Model

The example models an MSK transmitted signal undergoing channel impairments, including these components:

1. An MSK signal source that uses the Bernoulli Binary Generator block to output equiprobable symbols and modulates the symbols using an MSK Modulator Baseband block
2. A channel model that incorporates independently variable offsets in the timing phase, frequency, and phase. The channel model also includes the AWGN Channel block
3. Signal recovery, consisting of:
   - Timing recovery using the MSK-Type Signal Timing Recovery block
   - Carrier frequency recovery using the delay and multiply methods
   - Carrier phase recovery using the CPM Phase Recovery block
4. An MSK Demodulator Baseband block
5. Blocks that compute and display the system's bit error rate (BER)

When you load the model, it also initializes some parameters that several blocks share.

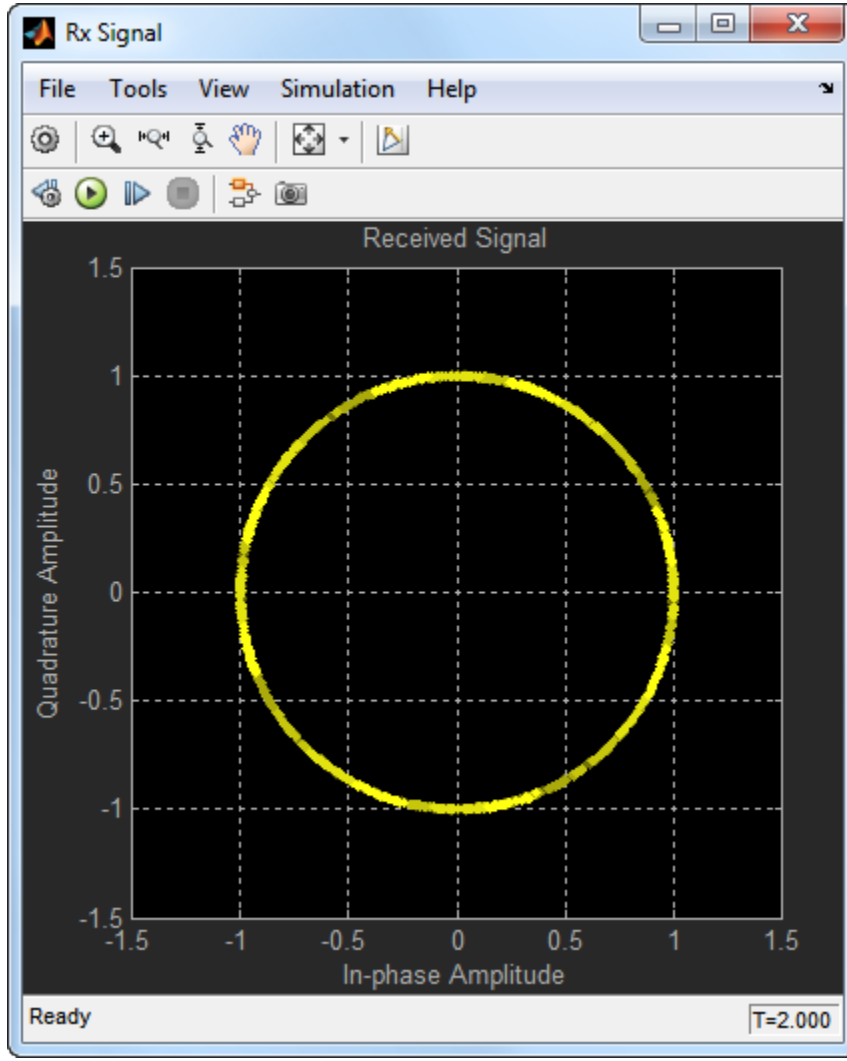


Results and Displays

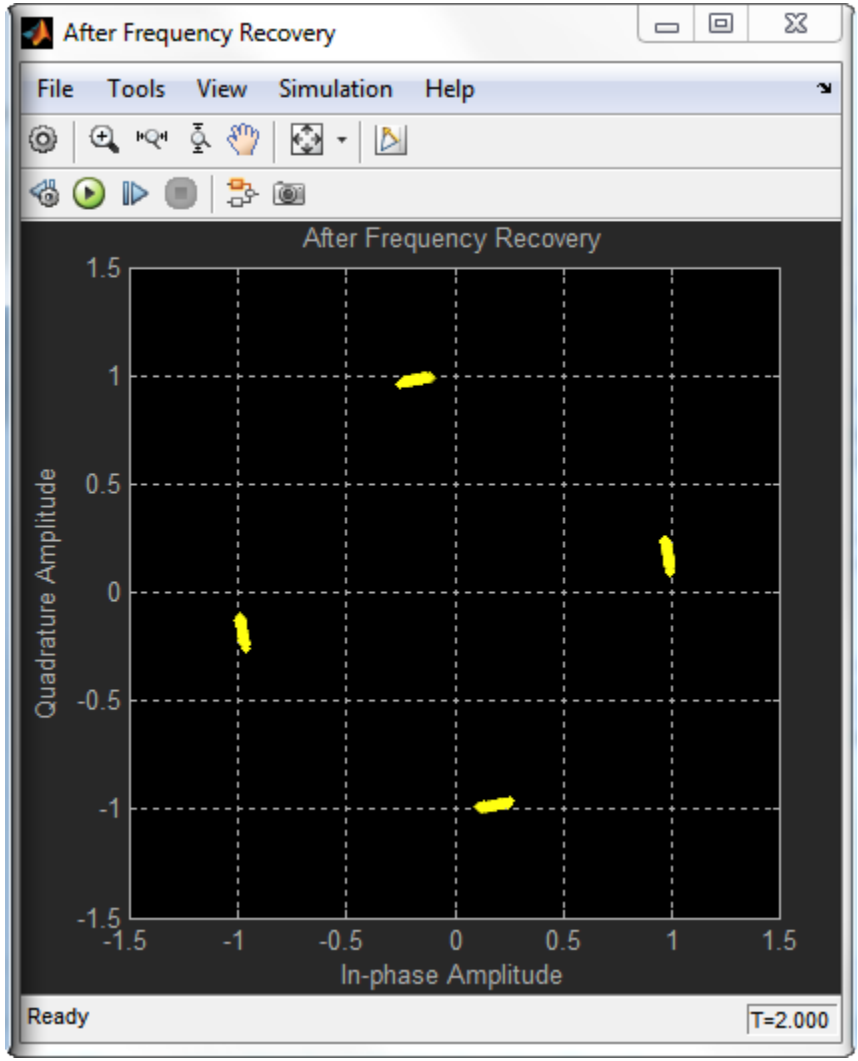
When you run the simulation, the displays show the estimated values for the impairments as well as the BER metrics. In particular, the display labeled BER Metrics shows a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

You can view the MSK signal via the Constellation Diagram blocks at the different stages. This provides a compelling visual rendition of the recovery algorithms in action, especially as you turn the algorithms on and off using the three control switches.

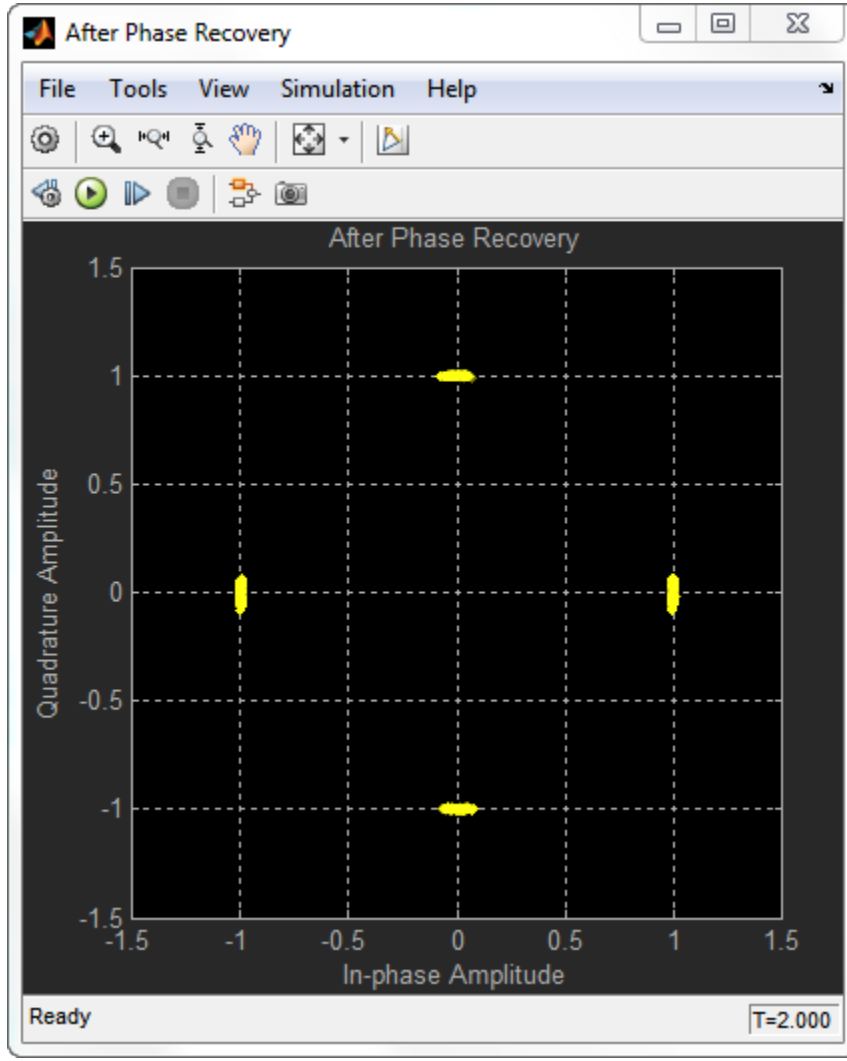
Scatter plot of received signal:



Scatter plot of signal after timing and carrier frequency recovery:



Scatter plot of signal after carrier phase recovery:



You can also reset the BER computation after the signal has reached a steady state.

Experimenting with the Example

The example is designed so that you can vary the impairments independently while the simulation is running. You can also use the toggle switches to turn the recovery schemes on and off while the simulation is running, and then see the effects on the scatter plots.

Gardner Timing Phase Recovery

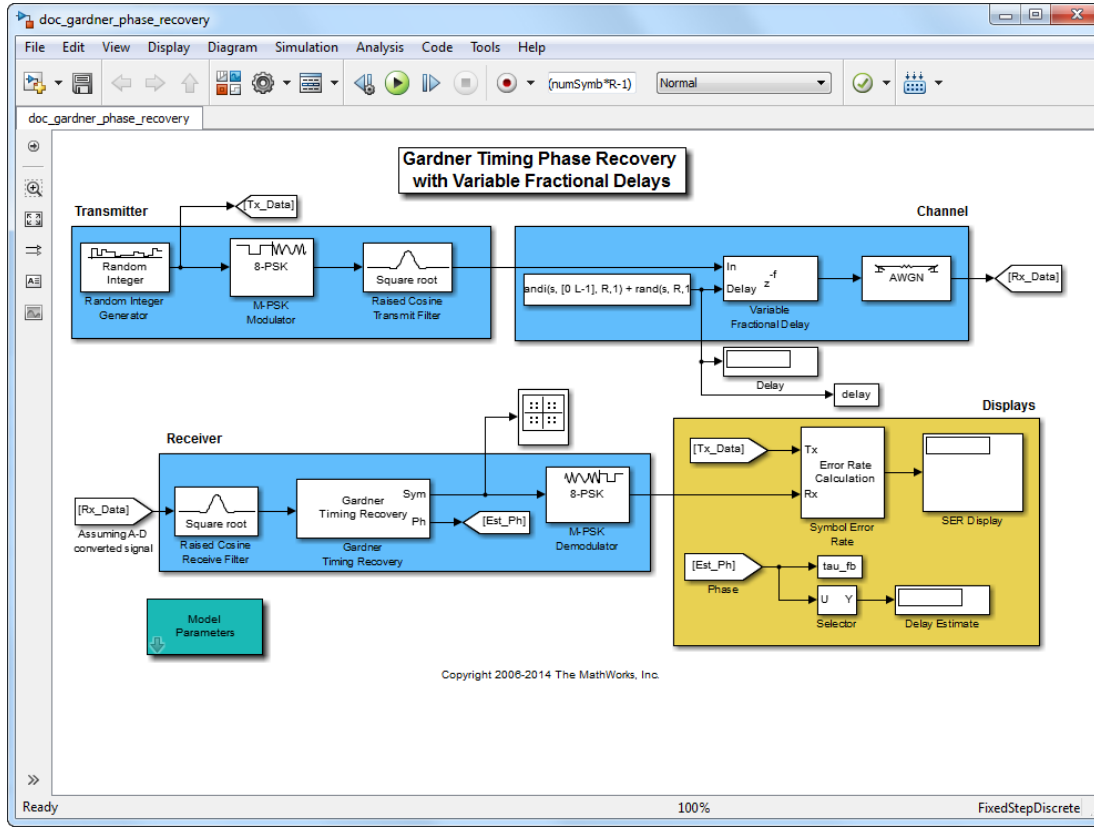
In this section...

This model shows symbol timing recovery for an 8-PSK modulated and filtered signal that is transmitted over a channel whose delay varies.

Exploring the Example

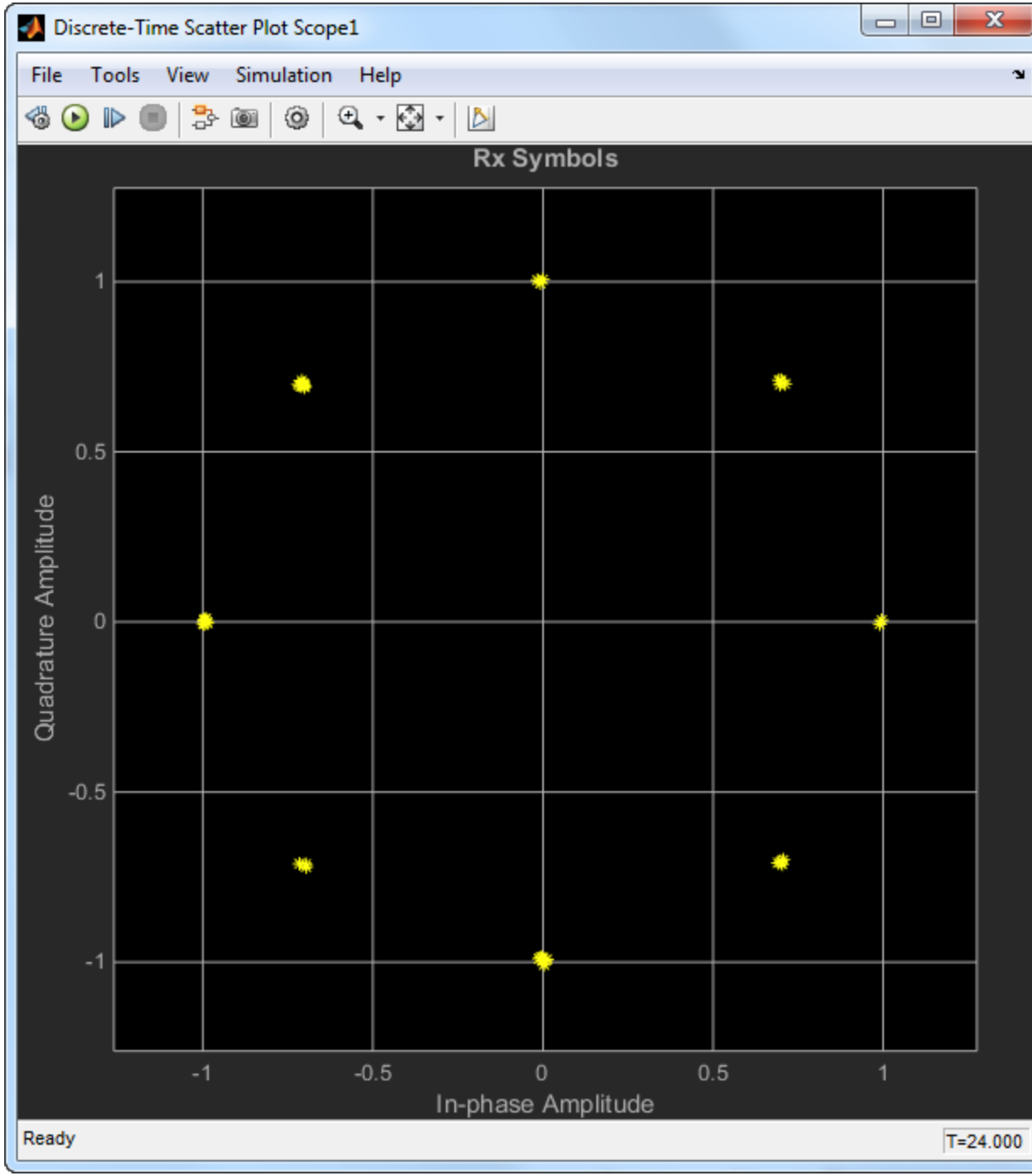
The example, doc_gardner_phase_recovery, models an 8-PSK transmitted signal and includes these components:

- A Random Integer Generator block that generates uniformly distributed integers in the range [0, M-1], where M is the alphabet size.
- An M-PSK Modulator Baseband block followed by a Raised Cosine Transmit Filter block, which upsamples and applies pulse shaping.
- A channel consisting of a Variable Integer Delay block, which introduces random integer delays less than the number of samples per symbol, and an AWGN Channel block.
- A Raised Cosine Receive Filter block whose characteristics match those of the transmit filter.
- A Gardner Timing Recovery block.
- An Error Rate Calculation block.

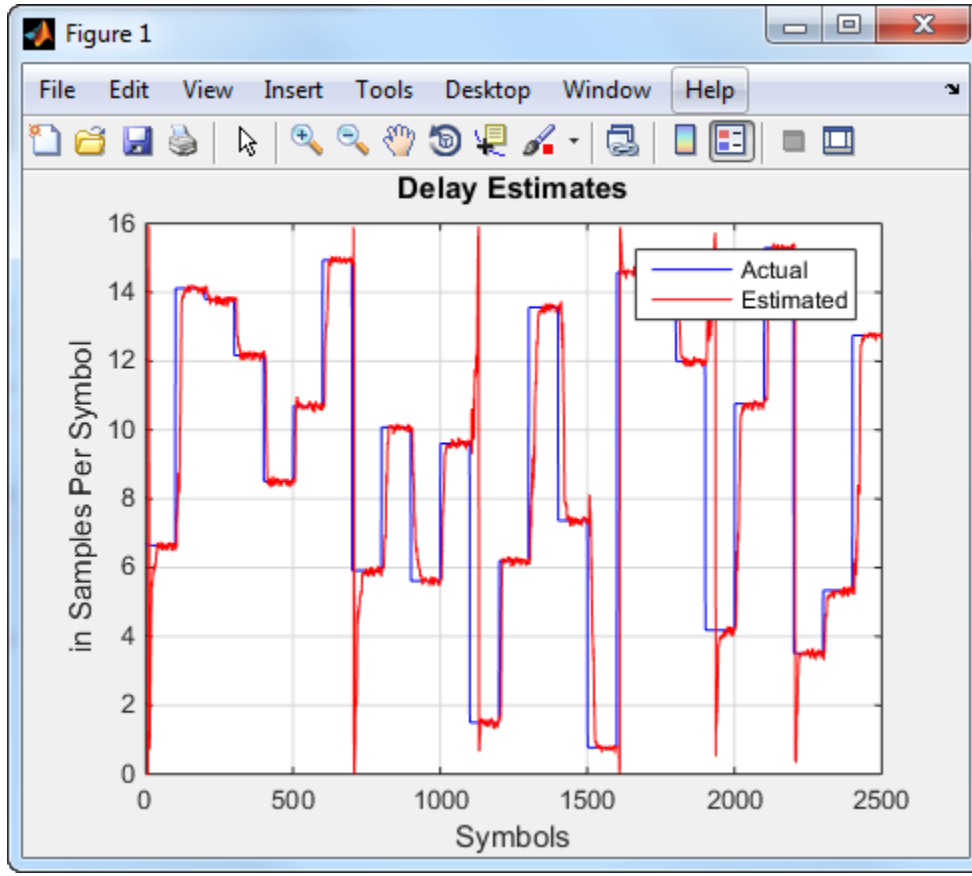


Results and Displays

When you run the simulation, the display blocks show how closely the Gardner method tracks the delay introduced into the link. The block labeled SER Display shows the calculated SER, the number of observed errors , and the number of transmitted symbols.


Discrete-Time Scatter Plot Scope1
File
Tools
View
Simulation
Help
Rx Symbols
1
0.5
0
-0.5
-1
Quadrature Amplitude
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=24.000

At the end of the simulation, the actual and estimated delays are plotted for sets of 100 symbols. The actual delay is shown in blue while the estimated delay is shown in red. From this plot, you can see the tracking behavior of the Gardner method.



Experimenting with the Example

The Model Parameters block makes it easy for you to vary certain parameters in the model. Among the ways that you can modify the model to learn more about its components or about timing recovery include:

- Vary the **Rolloff factor** of the raised cosine filters.

- Vary the **Step size** parameter (equivalent to the **Error update gain** of the Gardner Timing Recovery block).
- Vary the **SNR (dB)** of the AWGN channel.
- Vary the **Number of symbols per frame**.

You can also try using a different modulation scheme such as QPSK or 16-QAM. To switch from PSK to another modulation type, you would need to replace the modulator and demodulator blocks as well as to set the **Alphabet size** parameter in the Model Parameters block to the desired value.

7

Reed-Solomon Coding

Reed-Solomon Coding Part I – Erasures

This example shows how to configure the `RSEncoder` and `RSDecoder` System objects to perform Reed-Solomon (RS) block coding with erasures when simulating a communications system. RS decoders can correct both errors and erasures. A receiver that identifies the most unreliable symbols in a given codeword can generate erasures. When a receiver erases a symbol, it replaces that symbol with a zero. The receiver then passes a flag to the decoder, indicating that the symbol is an erasure, not a valid code symbol. In addition, an encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures the exact same way when it decodes a symbol. Puncturing also has the added benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance, given the same demodulator input energy per bit to noise power spectral density ratio (E_b/N_0). Note that puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword. In addition to this example, the examples "Reed-Solomon Coding Part II – Punctures" and "Reed-Solomon Coding Part III – Shortening" show RS block coding with punctures and shortened codes, respectively.

Introduction

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a rectangular 64-QAM modulator, an AWGN channel, a rectangular 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the rectangular QAM modulator to outputs from the rectangular QAM demodulator. This example obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The script file RSCodingConfigExample configures the rectangular 64-QAM modulator and demodulator, the AWGN channel, and the error rate measurement System objects used to simulate the communications system. The script also sets an uncoded E_b/N_0 ratio to EbNoUncoded = 15 dB, and sets the simulation stop criteria by defining the target number of errors and the maximum number of bit transmissions to 500 and 5×10^6 respectively.

```
RSCodingConfigExample
```

Configuring the RS Encoder/Decoder

This example shows a (63,53) RS code operating with a 64-QAM modulation scheme. This code can correct (63-53)/2 = 5 errors, or it can alternatively correct (63-53) = 10 erasures. For each codeword at the output of the 64-QAM demodulator, the receiver determines the six least reliable symbols using the RSCodingGetErasuresExample function. The indices that point to the location of these unreliable symbols are passed to the RS decoder via an input to the step method. The RS decoder treats these symbols as erasures resulting in an error correction capability of (10-6)/2 = 2 errors per codeword.

Create a (63,53) `RSEncoder` System object and set the `BitInput` property to `false` to specify that the encoder inputs and outputs are integer symbols.

```
N = 63;  % Codeword length
K = 53;  % Message length
hEnc = comm.RSEncoder(N,K, 'BitInput', false);
numErasures = 6;
```

Create an `RSDecoder` System object using the same settings as in the encoder. Request an additional input for specifying erasures via an input to the `step` method. This is done by setting the `ErasuresInputPort` property to `true`.

```
hDec = comm.RSDecoder(N,K, 'BitInput', false, 'ErasuresInputPort', true);
```

Set the `NumCorrectedErrorsOutputPort` property to `true` so that the `step` method of the decoder outputs the number of corrected errors. A non negative value in the error output denotes the number of corrected errors in the input codeword. A value of −1 in the error output indicates a decoding error. A decoding error occurs when the input codeword has more errors than the error correction capability of the RS code.

```
hDec.NumCorrectedErrorsOutputPort = true;
```

Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded E_b/N_0 values so that they correspond to the energy ratio at the encoder output.

This ratio is the coded E_b/N_0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. Set the `EbNo` property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/N);
hChan.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
    (codedErrorStats(3) < maxNumTransmissions)

  % Data symbols - transmit 1 message word at a time. Each message word has
  % K symbols in the [0 N] range.
  data = randi([0 N],K,1);

  % Encode the message word. The encoded word encData is N symbols long.
  encData = step(hEnc, data);

  % Modulate encoded data.
  modData = step(hMod, encData);

  % Add noise.
  chanOutput = step(hChan, modData);

  % Demodulate channel output.
  demodData = step(hDemod, chanOutput);

  % Find the 6 least reliable symbols and generate an erasures vector using
  % the RSCodingGetErasuresExample function. The length of the erasures vector
  % must be equal to the number of symbols in the demodulated codeword. A
  % one in the ith element of the vector erases the ith symbol in the
  % codeword. Zeros in the vector indicate no erasures.
  erasuresVec = RSCodingGetErasuresExample(chanOutput, numErasures);

  % Decode data.
  [estData, errs] = step(hDec, demodData, erasuresVec);

  % If a decoding error did not occur, accumulate the number of corrected
  % errors using the cumulative sum object.
```

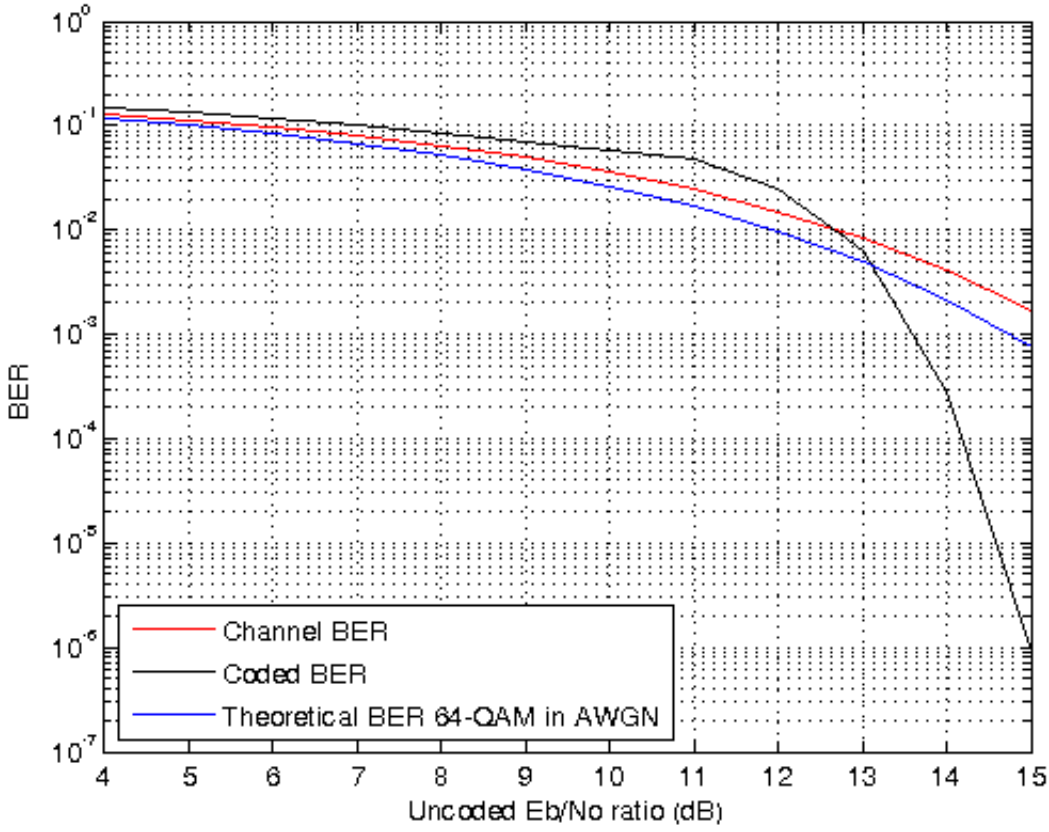
```
  if errs >= 0
    correctedErrors = step(hCumSum, errs);
  end

  % Convert integers to bits and compute the channel BER.
  chanErrorStats(:,1) = ...
    step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));

  % Convert integers to bits and compute the coded BER.
  codedErrorStats(:,1) = ...
    step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
```

The `step` method of the error rate measurement objects, `hChanBERCalc` and `hCodedBERCalc`, outputs a 3-by-1 vector containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the coded BER and the total number of errors corrected by the RS decoder.

```
codedBitErrorRate = codedErrorStats(1)
totalCorrectedErrors = correctedErrors
```

```
codedBitErrorRate =

     0


totalCorrectedErrors =

   882
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. The results from the simulation are shown. The channel BER is worse than the theoretical 64-QAM BER because E_b/N_0 is reduced by the code rate.



Summary

This example utilized several System objects to simulate a rectangular 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS decoder to decode symbols with erasures. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

The examples “Reed-Solomon Coding Part II – Punctures” and “Reed-Solomon Coding Part III – Shortening” show how to perform RS block coding with punctured and shortened codes, respectively.

Appendix

This example uses the following script and helper function:

- RSCodingConfigExample
- RSCodingGetErasuresExample

Selected Bibliography

[1] G. C. Clark, Jr., J. B. Cain, *Error-Correction Coding for Digital Communications*, Plenum Press, New York, 1981.

Reed-Solomon Coding Part II – Punctures

This example shows how to set up the Reed-Solomon (RS) encoder/decoder to use punctured codes. An encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures in exactly the same way when it decodes. Puncturing has the added benefit of making the code rate more flexible, at the expense of some error correction capability.

In addition to this example, the example "Reed-Solomon Coding Part I – Erasures" shows a rectangular 64-QAM communications system with a (63,53) RS block code with erasures, and the example "Reed-Solomon Coding Part III – Shortening" shows RS block coding with shortened codes.

Introduction

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a rectangular 64-QAM modulator, an AWGN channel, a rectangular 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures and puncturing by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the rectangular QAM modulator to outputs from the rectangular QAM demodulator. This example obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The script file RSCodingConfigExample configures the rectangular 64-QAM modulator and demodulator, the AWGN channel, and the error rate measurement System objects used to simulate the communications system. The script also sets an uncoded E_b/N_0 ratio to EbNoUncoded = 15 dB, and sets the simulation stop criteria by defining the target number of errors and the maximum number of bit transmissions to 500 and 5×10^6, respectively.

```
RSCodingConfigExample
```

Configuring the RS Encoder/Decoder

Consider the same (63,53) RS code operating in concert with a 64-QAM modulation scheme that was used in the example "Reed-Solomon Coding Part I – Erasures" to showcase how to decode erasures. This example shows how to set up the RS encoder/

decoder to use a punctured code. In addition to decoding receiver-generated erasures, the RS decoder can correct encoder-generated punctures. The decoding algorithm is identical for the two cases. For each codeword, the sum of the punctures and erasures cannot exceed twice the error-correcting capability of the code.

```
N = 63;  % Codeword length
K = 53;  % Message length
numErasures = 6;
hEnc = comm.RSEncoder(N,K, 'BitInput', false);
hDec = comm.RSDecoder(N,K, 'BitInput', false, 'ErasuresInputPort', true);
```

To enable code puncturing you set the `PuncturePatternSource` property to 'Property' and set the `PuncturePattern` property to the desired puncture pattern vector. The same puncture vector must be specified in both the encoder and decoder. This example punctures two symbols from each codeword. Values of one in the puncture pattern vector indicate nonpunctured symbols, while values of zero indicate punctured symbols.

```
numPuncs = 2;

hEnc.PuncturePatternSource = 'Property';
hEnc.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];

hDec.PuncturePatternSource = 'Property';
hDec.PuncturePattern = hEnc.PuncturePattern;
```

Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The length of the codewords generated by the RS encoder is reduced by the number of punctures specified in the puncture pattern vector. For this reason, the value of the coded E_b/N_0 ratio needs to be adjusted to account for these punctures. In this example, the uncoded E_b/N_0 ratio relates to the coded E_b/N_0 as shown below. Set the `EbNo` property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/(N - numPuncs));
hChan.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
```

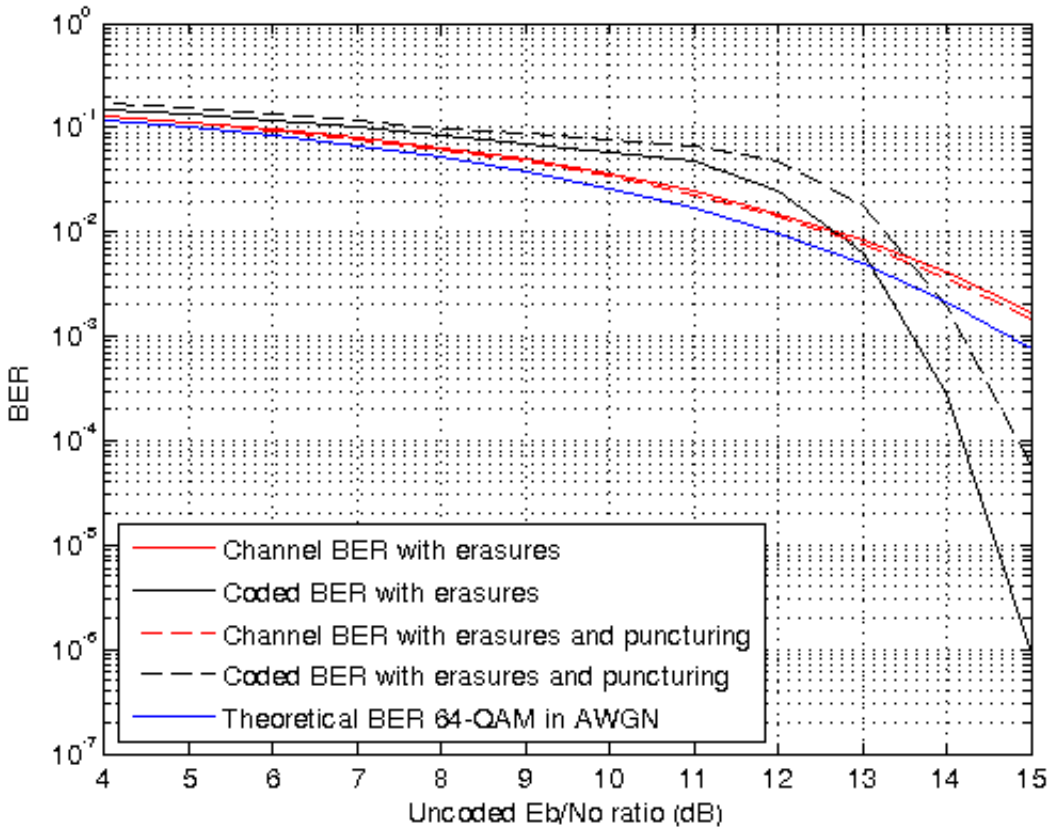
```
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
    (codedErrorStats(3) < maxNumTransmissions)

  % Data symbols - transmit 1 message word at a time. Each message word
  % has K symbols in the [0 N] range.
  data = randi([0 N],K,1);

  % Encode the message word. The encoded word encData is N-numPuncs symbols
  % long.
  encData = step(hEnc, data);

  % Modulate encoded data.
  modData = step(hMod, encData);

  % Add noise.
  chanOutput = step(hChan, modData);

  % Demodulate channel output.
  demodData = step(hDemod, chanOutput);

  % Get erasures vector.
  erasuresVec = RSCodingGetErasuresExample(chanOutput,numErasures);

  % Decode data.
  [estData, errs] = step(hDec, demodData, erasuresVec);

  % If a decoding error did not occur, accumulate the number of corrected
  % errors using the cumulative sum objet.
  if errs >= 0
    correctedErrors = step(hCumSum, errs);
  end

  % Convert integers to bits and compute the channel BER.
  chanErrorStats(:,1) = ...
    step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));

  % Convert integers to bits and compute the coded BER.
  codedErrorStats(:,1) = ...
    step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
```

The `step` method of the error rate measurement objects, `hChanBERCalc` and `hCodedBERCalc`, outputs a 3-by-1 vector containing updates of the measured BER value, the number of errors, and the total number of bit transmissions. Display the coded BER and the total number of errors corrected by the RS decoder.

```
codedBitErrorRate = codedErrorStats(1)
totalCorrectedErrors = correctedErrors
```

```
codedBitErrorRate =

   4.3198e-05


totalCorrectedErrors =

   578
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. The results from the simulation are shown in the following figure. For comparison, the figure also shows the results obtained in the example "Reed-Solomon Coding Part I – Erasures", which correspond to a system with erasures but no puncturing.



From the curves, observe that the channel BER is slightly better in the punctured case. The reason for this is that the coded E_b/N_0 is slightly higher. On the other hand, the coded BER is worse in the punctured case, because the two punctures reduce the error correcting capability of the code by one, leaving it able to correct only (10-6-2)/2 = 1 error per codeword.

Summary

The example utilized several System objects to simulate a rectangular 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS encoder/decoder System objects to obtain punctured codes. System

performance was measured using channel and coded BER curves obtained using the error rate measurement System objects.

The example “Reed-Solomon Coding Part III – Shortening” shows how to perform RS block coding with shortened codes.

Appendix

This example uses the following script and helper function:

- RSCodingConfigExample
- RSCodingGetErasuresExample

Selected Bibliography

[1] G. C. Clark, Jr., J. B. Cain, *Error-Correction Coding for Digital Communications*, Plenum Press, New York, 1981.

Reed-Solomon Coding Part III – Shortening

This example shows how to set up the Reed-Solomon (RS) encoder/decoder to shorten the (63,53) code to a (28,18) code.

In addition to this example, the examples "Reed-Solomon Coding Part I – Erasures" and "Reed-Solomon Coding Part II – Punctures" show a rectangular 64-QAM communications system with a (63,53) RS block code with erasures and punctures respectively. As was mentioned in the two examples, puncturing has the benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance, given the same demodulator input E_b/N_0. Note that puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword.

Introduction

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a rectangular 64-QAM modulator, an AWGN channel, a rectangular 64-QAM demodulator, and an RS decoder. It analyzes the effects of RS coding with erasures, puncturing, and shortening, by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the rectangular QAM modulator to outputs from the rectangular QAM demodulator. This example obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The script file RSCodingConfigExample configures the rectangular 64-QAM modulator and demodulator, the AWGN channel, and the error rate measurement System objects used to simulate the communications system. The script also sets an uncoded E_b/N_0 ratio to EbNoUncoded = 15 dB, and sets the simulation stop criteria by defining the target number of errors and the maximum number of bit transmissions to 500 and 5×10^6, respectively.

```
configureRSCodingDemo
```

Configuring the RS Encoder/Decoder

Consider the same (63,53) RS code operating in concert with a 64-QAM modulation scheme that was used in the examples Reed-Solomon Coding Part I - Erasures and Reed-Solomon Coding Part II - Punctures to showcase how to decode erasures and how to puncture the code. This example shows how to shorten the (63,53) code to a (28,18) code.

Shortening a block code removes symbols from its message portion, while puncturing removes symbols from its parity portion. You can incorporate both techniques with the RS encoder and decoder System objects.

For example, to shorten a (63,53) code to a (53,43) code, you can simply enter 53 and 43 for the `CodewordLength` and `MessageLength` properties, respectively (since $2\lceil \log_2(53+1) \rceil - 1 = 63$). However, to shorten it by 35 symbols to a (28,18) code, you must explicitly specify that the symbols belong to the Galois field GF(2^6). Otherwise, the RS blocks will assume that the code is shortened from a (31,21) code (since $2\lceil \log_2(28+1) \rceil - 1 = 31$).

Set the RS encoder/decoder System objects so that they perform block coding with a (28,18) code shortened from a (63,53) code. This corresponds to a shortening length of 35.

```
shortenLength = 35;
```

Create a (63,53) `RSEncoder` System object, and an `RSDecoder` object with an erasures input port, and two punctures.

```
N = 63;  % Codeword length
K = 53;  % Message length
numErasures = 6;
numPuncs = 2;
hEnc = comm.RSEncoder('BitInput', false);
hDec = comm.RSDecoder('BitInput', false, 'ErasuresInputPort', true);
hEnc.PuncturePatternSource = 'Property';
hEnc.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];
hDec.PuncturePatternSource = 'Property';
hDec.PuncturePattern = hEnc.PuncturePattern;

% Set the shortened codeword length and message length values
hEnc.CodewordLength = N - shortenLength;
hEnc.MessageLength = K - shortenLength;

hDec.CodewordLength = N - shortenLength;
hDec.MessageLength = K - shortenLength;
```

Specify the field of GF(2^6) in the RS encoder/decoder System objects, by setting the `PrimitivePolynomialSource` property to 'Property' and the `PrimitivePolynomial` property to a 6th degree primitive polynomial.

```
hEnc.PrimitivePolynomialSource = 'Property';
```

```
hEnc.PrimitivePolynomial = de2bi(primpoly(6, 'nodisplay'), 'left-msb');

hDec.PrimitivePolynomialSource = 'Property';
hDec.PrimitivePolynomial = de2bi(primpoly(6, 'nodisplay'), 'left-msb');
```

Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

Shortening alters the code rate much like puncturing does. You must adjust the value of the coded E_b/N_0 ratio to account for punctures and shortening. In this example, the uncoded E_b/N_0 ratio relates to the coded E_b/N_0, as shown in the following syntax. Set the property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + ...
  10*log10((K-shortenLength)/(N-numPuncs-shortenLength));

hChan.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
while (codedErrorStats(2) < targetErrors) && ...
    (codedErrorStats(3) < maxNumTransmissions)

  % Data symbols - transmit 1 message word at a time, each message word has
  % K-shortenLength symbols in the [0 2^P-1] range. P is the degree of the
  % primitive polynomial specified in the RS encoder/decoder, which in this
  % example equals 6.
  data = randi([0 2^6-1],K-shortenLength,1);

  % Encode the shortened message word. The encoded word encData is
  % N-numPuncs-shortenLength symbols long.
  encData = step(hEnc, data);

  % Modulate encoded data.
  modData = step(hMod, encData);

  % Add noise.
  chanOutput = step(hChan, modData);
```

```
  % Demodulate channel output.
  demodData = step(hDemod, chanOutput);

  % Get erasures vector.
  erasuresVec = getErasuresRSCodingDemo(chanOutput,numErasures);

  % Decode data.
  [estData, errs] = step(hDec, demodData, erasuresVec);

  % If a decoding error did not occur, accumulate the number of corrected
  % errors using the cumulative sum object.
  if errs >= 0
    correctedErrors = step(hCumSum, errs);
  end

  % Convert integers to bits and compute the channel BER.
  chanErrorStats(:,1) = ...
    step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));

  % Convert integers to bits and compute the coded BER.
  codedErrorStats(:,1) = ...
    step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
```

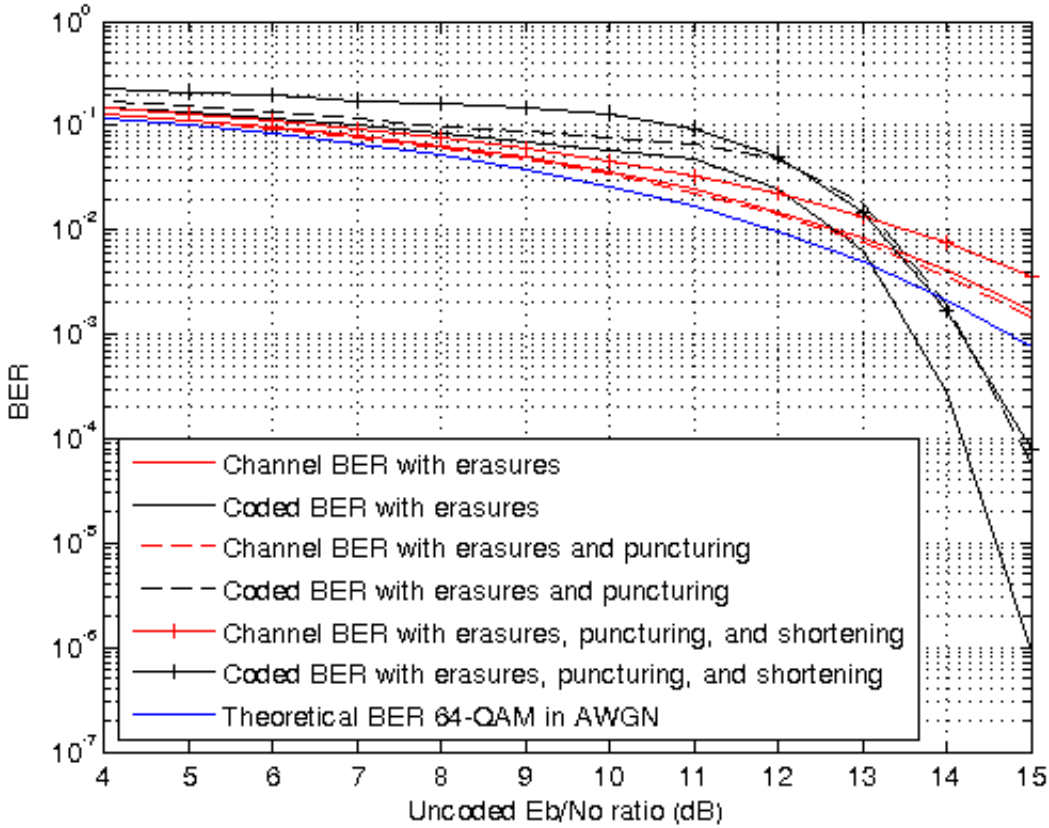
The `step` method of the error rate measurement objects, `hChanBERCalc` and `hCodedBERCalc`, outputs a 3-by-1 vector containing updates of the measured BER value, the number of errors, and the total number of bit transmissions. Display the coded BER and the total number of errors corrected by the RS decoder.

```
codedBitErrorRate = codedErrorStats(1)
totalCorrectedErrors = correctedErrors
```

```
codedBitErrorRate =

   9.6599e-05


totalCorrectedErrors =

        1436
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target

number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. The results from the simulation are shown in the following figure. For comparison, the figure also shows the results obtained in the examples "Reed-Solomon Coding Part I – Erasures" and "Reed-Solomon Coding Part II – Punctures". The results of "Reed-Solomon Coding Part I – Erasures" correspond to a system with erasures but no puncturing. The results of "Reed-Solomon Coding Part II – Punctures" correspond to a system with erasures and punctures.



From the curves, observe that the channel BER is worse with shortening because the coded E_b/N_0 is worse. This degraded coded E_b/N_0 occurs because the code rate of the

shortened code is much lower than that of the nonshortened code. As a result, the coded BER is also worse with shortening than without, especially at lower E_b/N_0 values.

Summary

This example utilized several System objects to simulate a rectangular 64-QAM communications system over an AWGN channel with a shortened RS block code. It showed how to configure the RS encoder/decoder to shorten a (63,53) code to a (28,18) code. System performance was measured using channel and coded BER curves obtained with the help of error rate measurement System objects.

The examples "Reed-Solomon Coding Part I – Erasures", "Reed-Solomon Coding Part II – Punctures", and "Reed-Solomon Coding Part III – Shortening" show how to configure RS encoders and decoders to perform block coding with erasures, puncturing, and shortening.

Appendix

This example uses the following script and helper function:

- RSCodingConfigExample
- RSCodingGetErasures

Selected Bibliography

[1] G. C. Clark, Jr., J. B. Cain, *Error-Correction Coding for Digital Communications*, Plenum Press, New York, 1981.

Reed-Solomon Coding with Erasures, Punctures, and Shortening

In this section...

This model shows how to configure Reed-Solomon (RS) codes to perform block coding with erasures, punctures, and shortening.

RS decoders can correct both errors and erasures. The erasures can be generated by a receiver that identifies the most unreliable symbols in a given codeword. When a receiver erases a symbol, it replaces the symbol with a zero and passes a flag to the decoder indicating that the symbol is an erasure, not a valid code symbol.

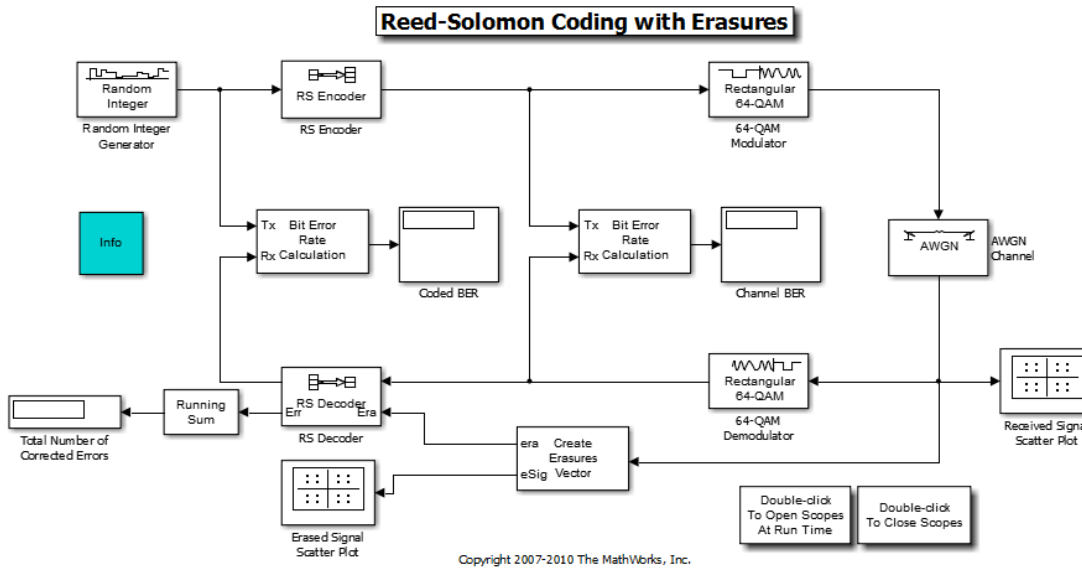
In addition, an encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures in exactly the same way when it decodes.

Puncturing has the added benefit of making the code rate a bit more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance, given the same demodulator input E_b/N_0. Note that puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword.

Decoding with Receiver Generated Erasures

This example shows a (63,53) RS code operating in concert with a 64-QAM modulation scheme. Since the code can correct (63-53)/2 = 5 errors, it can alternatively correct

(63-53) = 10 erasures. For each demodulated codeword, the receiver determines the six least reliable symbols by finding the symbols within a decision region that are nearest to a decision boundary. It then erases those symbols. We first open the model RSCodingErasuresExample.



Simulation and Visualization with Erasures Only

We then define system simulation parameters:

```
RS_TsUncoded = 1;                % Sample time (s)
RS_n = 63;                       % Codeword length
RS_k = 53;                       % Message length
RS_MQAM = 64;                    % QAM order
RS_numBitsPerSymbol = ...        % 6 bits per symbol
    log2(RS_MQAM);
RS_sigPower = 42;                % Assume points at +/-1, +/-3, +/-5, +/-7
RS_numErasures = 6;              % Number of erasures
RS_EbNoUncoded = 15;             % In dB
```

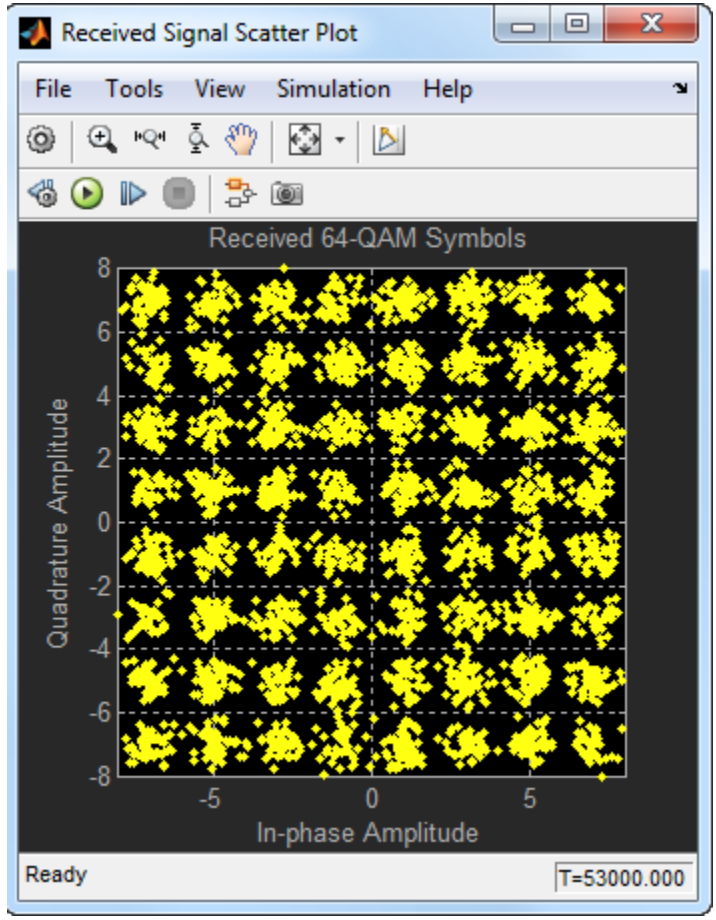
The system is simulated at an uncoded E_b/N_0 of 15 dB. However, the coded E_b/N_0 is reduced because of the redundant symbols added by the RS Encoder. Also, the period of

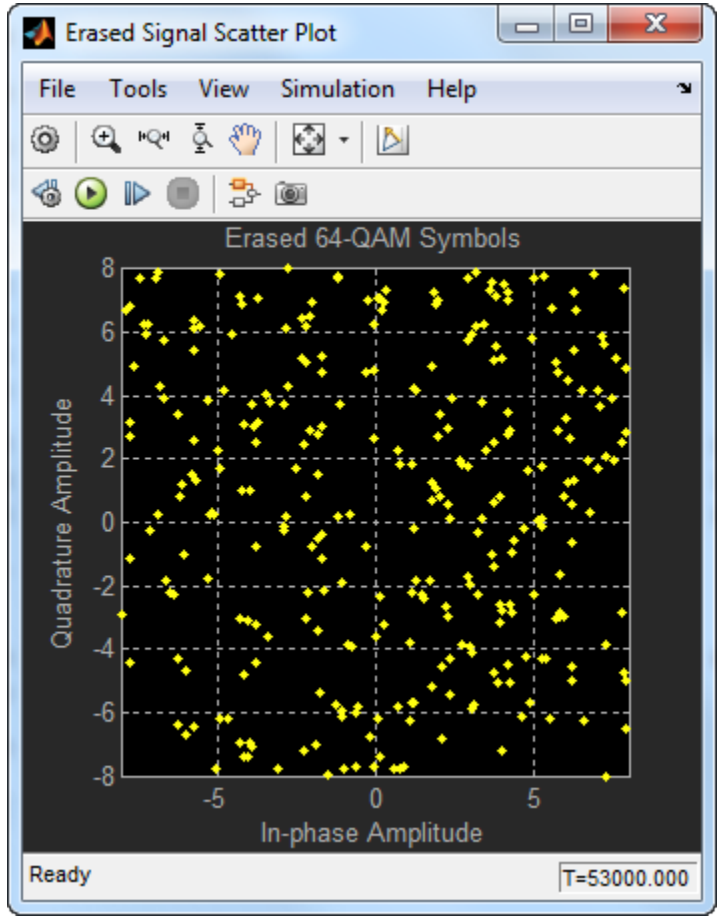
each frame in the model remains constant at 53 seconds, corresponding to a sample time of 1 second at the output of the Random Integer Generator. Moreover, the symbol time at the output of the RS Encoder is reduced by a factor of the code rate, because 63 symbols are output over the frame time of 53 seconds. The AWGN Channel block accounts for this by using the following parameters:

```
RS_EbNoCoded = RS_EbNoUncoded + 10*log10(RS_k/RS_n);
RS_TsymCoded = RS_TsUncoded * (RS_k/RS_n);
```

The receiver determines which symbols to erase by finding the 64-QAM symbols, per codeword, that are closest to a decision boundary. It deletes the six least reliable code symbols, which still allows the RS Decoder to correct (10-6)/2 = 2 errors per codeword.

We simulate the system, showing the received symbols and those symbols that were erased:


Received Signal Scatter Plot
File
Tools
View
Simulation
Help
Received 64-QAM Symbols
Quadrature Amplitude
In-phase Amplitude
8
6
4
2
0
-2
-4
-6
-8
-5
0
5
Ready
T=53000.000



BER Performance with Erasures Only

Now let's examine the BER performance at the output of the decoder. We set the stop time of the simulation to inf, then simulate until 100 bit errors are collected out of the RS Decoder. The 64-QAM BER is shown below, followed by the RS BER. This convention is followed for the remainder of this example.

```
BER_eras =
   1.7049e-03   2.5906e-06
```

Simulation with Erasures and Punctures

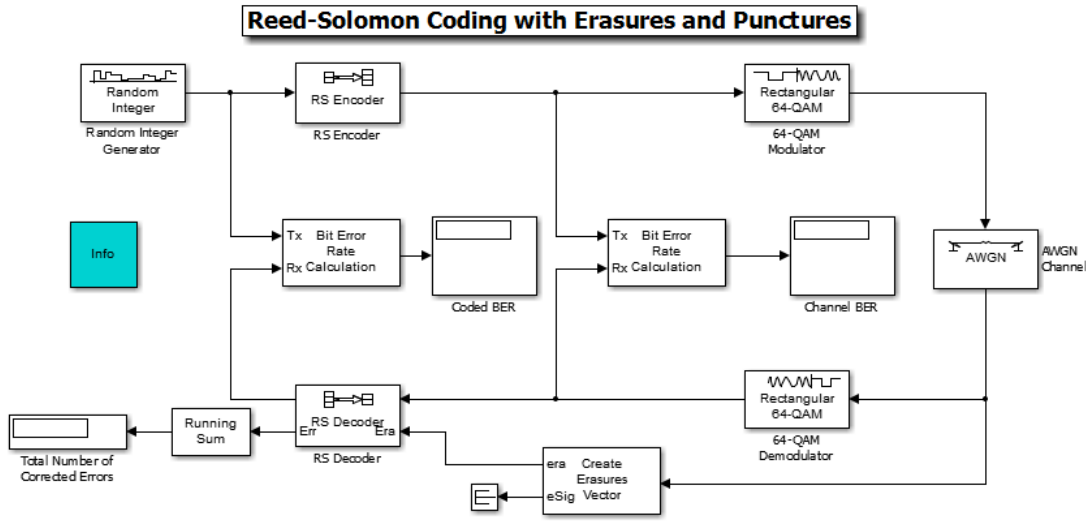
In addition to decoding receiver-generated erasures, the RS Decoder can correct encoder-generated punctures. The decoding algorithm is identical for the two cases, but the per-codeword sum of the punctures and erasures cannot exceed twice the error-correcting capability of the code. Consider the following model that performs decoding for both erasures and punctures.

The same puncture vector is specified in both the encoder and decoder blocks. This example punctures two symbols from each codeword. Vector values of "1" indicate nonpunctured symbols, while values of "0" indicate punctured symbols. In the erasures vector, however, values of "1" indicate erased symbols, while values of "0" indicate nonerased symbols.

Several of the parameters for the AWGN Channel block are now slightly different, because the length of the codeword is now different from the previous example. The block accounts for the size difference with the following code:

```
RS_EbNoCoded = RS_EbNoUncoded + 10*log10( RS_k / (RS_n - RS_numPuncs) );
RS_TsymCoded = RS_TsUncoded * ( RS_k / (RS_n - RS_numPuncs) );
```

We simulate the model, `RSCodingErasuresPunctExample.mdl`, collecting 1000 errors out of the RS Decoder block. Due to puncturing, the signal dimensions out of the encoder are 61-by-1, rather than 63-by-1 in the model with no puncturing. The Create Erasures Vector subsystem must also account for the size differences as it creates a 61-by-1 erasures vector. Open the model RSCodingErasuresPunctExample.



BER Performance with Erasures and Punctures

Let's compare the BERs for erasures decoding with and without puncturing.

The BER out of the 64-QAM Demodulator is slightly better in the punctured case, because the E_b/N_0 into the demodulator is slightly higher. However, the BER out of the RS Decoder is much worse in the punctured case, because the two punctures reduce the error correcting capability of the code by one, leaving it able to correct only (10-6-2)/2 = 1 error per codeword.

```
BER_eras =
   1.7049e-03   2.5906e-06
BER_eras_punc =
   1.4767e-03   6.1103e-05
```

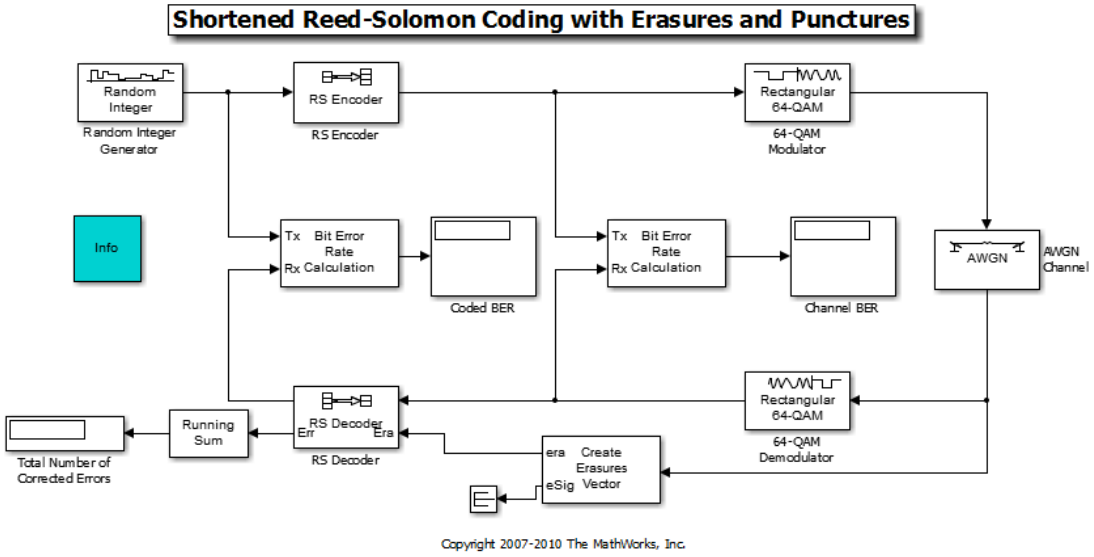
Specifying a Shortened Code

Shortening a block code removes symbols from its message portion, where puncturing removes symbols from its parity portion. You can incorporate both techniques with the RS encoder and decoder blocks.

For example, to shorten a (63,53) code to a (53,43) code, you can simply enter 53 and 43 for n and k, respectively, in the encoder and decoder block masks. However, if you

want to shorten it by 35 symbols to a (28,18) code, you must explicitly specify the field of GF(2^6). Otherwise, the RS blocks will assume that the code is shortened from a (31,21) code.

To specify the field of GF(2^6), you must use a nondefault primitive polynomial. The following model, `RSCodingErasuresPunctShortExample`, which incorporates erasures, punctures, and shortening, does precisely that. Examine how the polynomial is specified in the RS blocks. Open the model RSCodingErasuresPunctShortExample.



Simulation with Erasures, Punctures, and Shortening

Because shortening alters the code rate much like puncturing does, the AWGN parameters must be changed again. The AWGN Channel block accounts for this with the following code:

```
RS_EbNoCoded = RS_EbNoUncoded + 10*log10( (RS_k - RS_shortenLength) / (RS_n - RS_shorte
RS_TsymCoded = RS_TsUncoded * (RS_k - RS_shortenLength) / (RS_n - RS_shortenLength - RS
```

We simulate the model, once again collecting 1000 errors out of the RS Decoder block. Note that the signal dimensions out of the RS Encoder are 26x1, due to 35 symbols of shortening and 2 symbols of puncturing. Once again, the Create Erasures Vector subsystem must also account for the size difference caused by the shortened code.

BER Performance with Erasures, Punctures, and Shortening

Let's compare the BER performance for decoding with erasures only, with erasures and punctures, and with erasures, punctures, and shortening.

The BER out of the 64-QAM Demodulator is worse with shortening than it is without shortening. This is because the code rate of the shortened code is much lower than the code rate of the non-shortened code and therefore the coded E_b/N_0 into the demodulator is worse with shortening. A shortened code has the same error correcting capability as non-shortened code for the same E_b/N_0, but the reduction in E_b/N_0 manifests in the form of a higher BER out of the RS Decoder with shortening than without.

```
BER_eras =
   1.7049e-03   2.5906e-06
BER_eras_punc =
   1.4767e-03   6.1103e-05
BER_eras_punc_short =
   3.5975e-03   9.1940e-05
```

Further Exploration

You can experiment with these systems by running them over a loop of E_b/N_0 values and generating a BER curve for them. You can then compare their performance against a theoretical 64-QAM/RS system without erasures, punctures, or shortening. Use BERTool to generate the theoretical BER curves.

8

Galois Fields

Working with Galois Fields

In this section...

This example illustrates how to work with Galois fields.

Galois fields are used in error-control coding, where a Galois field is an algebraic field with a finite number of members. A Galois field that has 2^m members is denoted by GF(2^m), where m is an integer between 1 and 16 in this example.

Creating Galois Field Arrays

You create Galois field arrays using the GF function. To create the element 3 in the Galois field 2^2, you can use the following command:

```
A = gf(3,2)
```

```
A = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

           3
```

Using Galois Field Arrays

You can now use A as if it were a built-in MATLAB® data type. For example, here is how you can add two elements in a Galois field together:

```
A = gf(3,2);
B = gf(1,2);
```

```
C = A+B


C = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

           2
```

Arithmetic in Galois Fields

Note that 3 + 1 = 2 in this Galois field. The rules for arithmetic operations are different for Galois field elements compared to integers. To see some of the differences between Galois field arithmetic and integer arithmetic, first look at an addition table for integers 0 through 3:

```
  +__0__1__2__3
  0| 0  1  2  3
  1| 1  2  3  4
  2| 2  3  4  5
  3| 3  4  5  6
```

You can define such a table in MATLAB with the following commands:

```
A = ones(4,1)*[0 1 2 3];
B = [0 1 2 3]'*ones(1,4);
Table = A+B


Table =

     0     1     2     3
     1     2     3     4
     2     3     4     5
     3     4     5     6
```

Similarly, create an addition table for the field GF(2^2) with the following commands:

```
A = gf(ones(4,1)*[0 1 2 3],2);
B = gf([0 1 2 3]'*ones(1,4),2);
A+B
```

```
ans = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     1     2     3
     1     0     3     2
     2     3     0     1
     3     2     1     0
```

Using MATLAB® Functions with Galois Arrays

Many other MATLAB functions will work with Galois arrays. To see this, first create a couple of arrays.

```
A = gf([1 33],8);
B = gf([1 55],8);
```

Now you can multiply two polynomials.

```
C = conv(A,B)
```

```
C = GF(2^8) array. Primitive polynomial = D^8+D^4+D^3+D^2+1 (285 decimal)

Array elements =

     1    22   153
```

You can also find roots of a polynomial. (Note that they match the original values in A and B.)

```
roots(C)
```

```
ans = GF(2^8) array. Primitive polynomial = D^8+D^4+D^3+D^2+1 (285 decimal)

Array elements =

    33
    55
```

Hamming Code Example

The most important application of Galois field theory is in error-control coding. The rest of this example uses a simple error-control code, a Hamming code. An error-control code works by adding redundancy to information bits. For example, a (7,4) Hamming code maps 4 bits of information to 7-bit codewords. It does this by multiplying the 4-bit codeword by a 4 x 7 matrix. You can obtain this matrix with the HAMMGEN function:

```
[H,G] = hammgen(3)


H =

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


G =

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

H is the parity-check matrix and G is the generator matrix. To encode the information bits [0 1 0 0], multiply the information bits [0 1 0 0] by the generator matrix G:

```
A = gf([0 1 0 0],1)
Code = A*G


A = GF(2) array.

Array elements =

           0           1           0           0


Code = GF(2) array.

Array elements =

  Columns 1 through 6
```

```
           0           1           1           0           1           0

  Column 7

           0
```

Suppose somewhere along transmission, an error is introduced into this codeword. (Note that a Hamming code can correct only 1 error.)

```
Code(1) = 1   % Place a 1 where there should be a 0.


Code = GF(2) array.

Array elements =

  Columns 1 through 6

           1           1           1           0           1           0

  Column 7

           0
```

You can use the parity-check matrix H to determine where the error occurred, by multiplying the codeword by H:

```
H*Code'


ans = GF(2) array.

Array elements =

           1
           0
           0
```

To find the error, look at the parity-check matrix H. The column in H that matches [1 0 0]' is the location of the error. Looking at H, you can see that the first column is [1 0 0]'. This means that the first element of the vector Code contains the error.

```
H
```

```
H =

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

Convolutional Coding

Punctured Convolutional Coding

This example shows how to use the convolutional encoder and Viterbi decoder System objects to simulate a punctured coding system. The complexity of a Viterbi decoder increases rapidly with the code rate. Puncturing is a technique that allows the encoding and decoding of higher rate codes using standard rate 1/2 encoders and decoders.

Introduction

This example showcases the simulation of a communication system consisting of a random binary source, a convolutional encoder, a BPSK modulator, an additive white Gaussian noise (AWGN) channel, and a Viterbi decoder. The example shows how to run simulations to obtain bit error rate (BER) curves and compares these curves to a theoretical bound.

Initialization

Convolutional Encoding with Puncturing

Create a rate 1/2, constraint length 7 ConvolutionalEncoder System object. This encoder takes one-bit symbols as inputs and generates 2-bit symbols as outputs. If you assume 3-bit message words as inputs, then the encoder will generate 6-bit codeword outputs.

```
hConvEnc = comm.ConvolutionalEncoder(poly2trellis(7, [171 133]));
```

Specify a puncture pattern to create a rate 3/4 code from the previous rate 1/2 code using the puncture pattern vector [1;1;0;1;1;0]. The ones in the puncture pattern vector indicate that bits in positions 1, 2, 4, and 5 are transmitted, while the zeros indicate that bits in positions 3 and 6 are punctured or removed from the transmitted signal. The effect of puncturing is that now, for every 3 bits of input, the punctured code generates 4 bits of output (as opposed to the 6 bits produced before puncturing). This results in a rate 3/4 code. In the example at hand, the length of the puncture pattern vector must be an integer multiple of 6 since 3-bit inputs get converted into 6-bit outputs by the rate 1/2 convolutional encoder.

To set the desired puncture pattern in the convolutional encoder System object, `hConvEnc` , set the `PuncturePatternSource` property to `Property` and the `PuncturePattern` property to `[1;1;0;1;1;0]` .

```
hConvEnc.PuncturePatternSource = 'Property';
hConvEnc.PuncturePattern = [1;1;0;1;1;0];
```

Modulator and Channel

Create a BPSKModulator System object to transmit the encoded data using binary phase shift keying modulation over a channel.

```
hMod = comm.BPSKModulator;
```

Create an AWGNChannel System object. Set the `NoiseMethod` property of the channel to `Signal to noise ratio (Eb/No)` to specify the noise level using the energy per bit to noise power spectral density ratio (Eb/No). When running simulations, test the coding system for different values of Eb/No ratio by changing the `EbNo` property of the channel object. The output of the BPSK modulator generates unit power signals; set the `SignalPower` property to 1 Watt. The system at hand is at the symbol rate; set the `SamplesPerSymbol` property to 1.

```
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)',...
  'SignalPower', 1, 'SamplesPerSymbol', 1);
```

Viterbi Decoding with Depuncturing

Configure a ViterbiDecoder System object so it decodes the punctured code specified for the convolutional encoder. This example assumes unquantized inputs to the Viterbi decoder, so set the `InputFormat` property to `Unquantized` .

```
hVitDec = comm.ViterbiDecoder(poly2trellis(7, [171 133]), ...
  'InputFormat', 'Unquantized');
```

In general, the puncture pattern vectors you use for the convolutional encoder and Viterbi decoder must be the same. To specify the puncture pattern, set the `PuncturePatternSource` property of the Viterbi decoder System object, `hVitDec` , to `Property` . Set the `PuncturePattern` property to the same puncture pattern vector you use for the convolutional encoder.

Because the punctured bits are not transmitted, there is no information to indicate their values. As a result, the decoding process ignores them.

```
hVitDec.PuncturePatternSource =  'Property';
hVitDec.PuncturePattern = hConvEnc.PuncturePattern;
```

For a rate 1/2 code with no puncturing, you normally set the traceback depth of a Viterbi decoder to a value close to 40. Decoding punctured codes requires a higher value, in order to give the decoder enough data to resolve the ambiguities that the punctures introduce. This example uses a traceback depth of 96. Set this value using the `TraceBackDepth` property of the Viterbi decoder object, `hVitDec` .

```
hVitDec.TracebackDepth = 96;
```

Calculating the Error Rate

Create an ErrorRate calculator System object to compare decoded bits to the original transmitted bits. The output of the error rate calculator object is a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed. The Viterbi decoder creates a delay in the output decoded bit stream equal to the traceback length. To account for this delay set the `ReceiveDelay` property of the error rate calculator System object to 96.

```
hErrorCalc = comm.ErrorRate('ReceiveDelay', hVitDec.TracebackDepth);
```

Stream Processing Loop

Analyze the BER performance of the punctured coding system for different noise levels.

Uncoded and Coded Eb/No Ratio Values

Typically, you measure system performance according to the value of the energy per bit to noise power spectral density ratio (Eb/No) available at the input of the channel encoder. The reason for this is that this quantity is directly controlled by the systems engineer. Analyze the performance of the coding system for Eb/No values between 2 and 5 dB.

```
EbNoEncoderInput = 2:0.5:5; % in dB
```

The signal going into the AWGN channel is the encoded signal. Convert the Eb/No values so that they correspond to the energy ratio at the encoder output. If you input three bits to the encoder and obtain four bit outputs, then the energy relation is given by the 3/4 rate as follows:

```
EbNoEncoderOutput = EbNoEncoderInput + 10*log10(3/4);
```

Simulation loop

To obtain BER performance results, transmit frames of 3000 bits through the communications system. For each Eb/No value, stop simulations upon reaching a specific number of errors or transmissions. To improve the accuracy of the results, increase the target number of errors or the maximum number of transmissions.

```
frameLength = 3000;         % this value must be an integer multiple of 3
targetErrors = 300;
```

```
maxNumTransmissions = 5e6;
```

Loop through the encoded Eb/No values (the simulation will take a few seconds to complete).

```
BERVec = zeros(3,length(EbNoEncoderOutput)); % Allocate memory to store results
for n=1:length(EbNoEncoderOutput)
  reset(hErrorCalc)
  reset(hConvEnc)
  reset(hVitDec)
  hChan.EbNo = EbNoEncoderOutput(n); % Set the channel EbNo value for simulation
  while (BERVec(2,n) < targetErrors) && (BERVec(3,n) < maxNumTransmissions)
    % Generate binary frames of size specified by the frameLength variable
    data = randi([0 1], frameLength, 1);
    % Convolutionally encode the data
    encData = step(hConvEnc, data);
    % Modulate the encoded data
    modData = step(hMod, encData);
    % Pass the modulated signal through an AWGN channel
    channelOutput = step(hChan, modData);
    % Pass the real part of the channel complex outputs as the unquantized
    % input to the Viterbi decoder.
    decData = step(hVitDec, real(channelOutput));
    % Compute and accumulate errors
    BERVec(:,n) = step(hErrorCalc, data, decData);
  end
end
```

Compare Results to Theoretical Curves

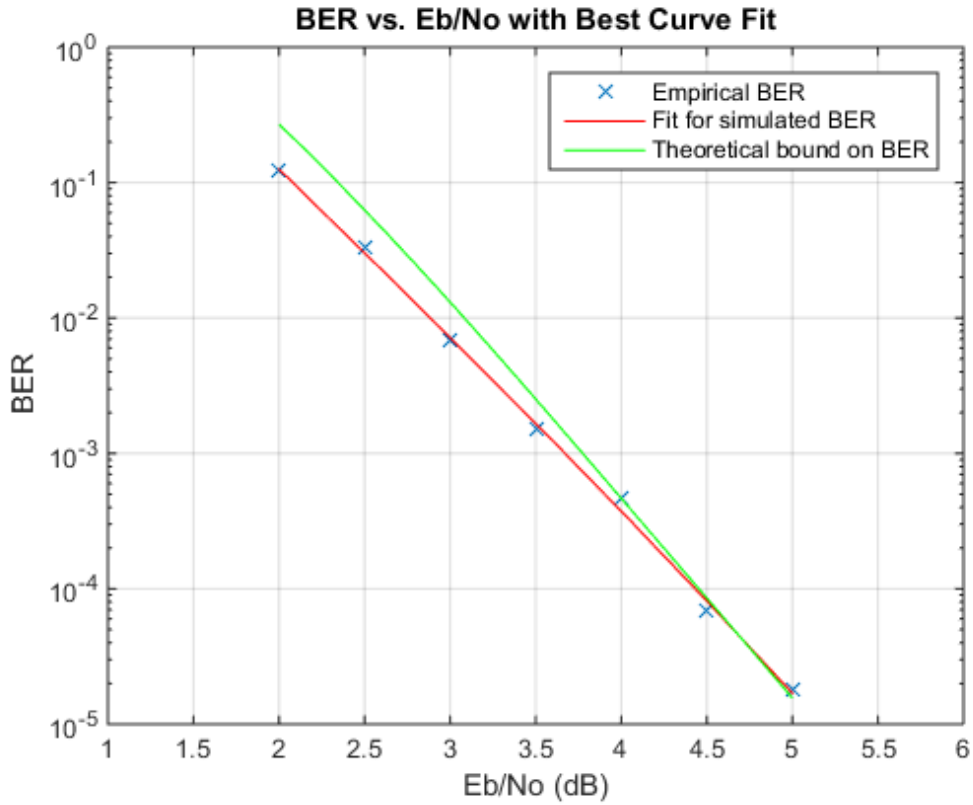
We compare the simulation results using an approximation of the bit error probability bound for a punctured code as per [1]. The following commands compute an approximation of this bound using the first seven terms of the summation for Eb/No values in `2:0.5:5` . The values used for nerr are found in Table 2 of [2].

```
dist = 5:11;
nerr = [42 201 1492 10469 62935 379644 2253373];
codeRate = 3/4;
bound = nerr*(1/6)*erfc(sqrt(codeRate*(10.0.^((2:.02:5)/10))'*dist))';
```

Plot results. If the target number of errors or maximum number of transmissions you specify for the simulation are too small, the curve fitting algorithm might fail.

```
berfit(EbNoEncoderInput,BERVec(1,:)); % Curve-fitted simulation results
hold on;
```

```
semilogy((2:.02:5),bound,'g'); % Theoretical results
legend('Empirical BER','Fit for simulated BER', 'Theoretical bound on BER')
axis([1 6 10^-5 1])
```



In some cases, at lower bit error rates, simulation results appear to indicate error rates slightly above the bound. This results from simulation variance (if fewer than 500 bit errors are observed) or from the finite traceback depth in the decoder.

Summary

We utilized several System objects to simulate a communications system with convolutional coding and puncturing. We simulated the system to obtain BER performance versus different Eb/No ratio values. The BER results were compared to theoretical bounds.

Selected Bibliography

1 Yasuda, Y., K. Kashiki, and Y. Hirata, "High Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding," IEEE® Transactions on Communications, Vol. COM-32, March, 1984, pp. 315-319

2 Begin, G., Haccoun, D., and Paquin, C., "Further results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding," IEEE Transactions on Communications, Vol. 38, No. 11, November, 1990, p. 1923

Iterative Decoding of a Serially Concatenated Convolutional Code

In this section...

This model shows how to use an iterative process to decode a serially concatenated convolutional code (SCCC).

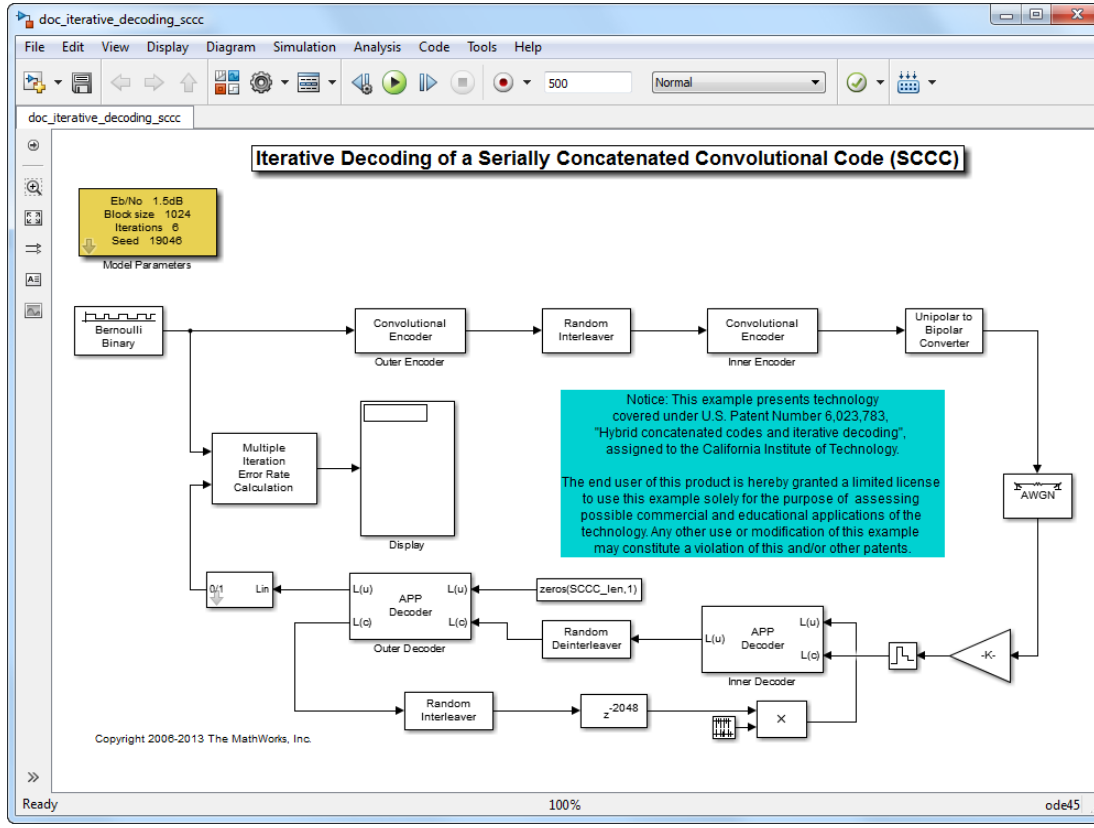
Note: This example presents technology covered under U.S. Patent Number 6,023,783, "Hybrid concatenated codes and iterative decoding," assigned to the California Institute of Technology. The end user of this product is hereby granted a limited license to use this example solely for the purpose of assessing possible commercial and educational applications of the technology. Any other use or modification of this example may constitute a violation of this and/or other patents.

Exploring the Example

The simulation generates information bits, encodes them using a serially concatenated convolutional code, and transmits the coded information along a noisy channel. The simulation then decodes the received coded information, using an iterative decoding process, and computes error statistics based on different numbers of iterations. Throughout the simulation, the error rates appear in a Display block.

Open the model, doc_iterative_decoding_sccc, by entering the following on the MATLAB command line.

```
doc_iterative_decoding_sccc
```



Variables in the Example

The Model Parameters block lets you vary the values of some quantities that the model uses. The table below indicates their names and meanings.

Name	Meaning
Eb/No	E_b/N_0 in channel noise, measured in dB; used to compute the variance of the channel noise
Block size	The number of bits in each frame of uncoded data

Name	Meaning
Number of iterations	The number of iterations to use when decoding
Seed	The initial seed in the Random Interleaver and Random Deinterleaver blocks

Creating a Serially Concatenated Code

The encoding portion of the example uses a Convolutional Encoder block to encode a data frame, a Random Interleaver block to shuffle the bits in the codewords, and another Convolutional Encoder block to encode the interleaved bits. Because these blocks are connected in series with each other, the resulting code is called a serially concatenated code.

Together, these blocks encode the 1024-bit data frame into a 3072-bit frame representing a concatenated code. These sizes depend on the model's **Block size** parameter (see the Model Parameters block). The code rate of the concatenated code is 1/3.

In general, the purpose of interleaving is to protect codewords from burst errors in a noisy channel. A burst error that corrupts interleaved data actually has a small effect on each of several codewords, rather than a large effect on any one codeword. The smaller the error in an individual codeword, the greater the chance that the decoder can recover the information correctly.

Convolutional Encoding Details

The two instances of the Convolutional Encoder block use their **Trellis structure** parameters to specify the convolutional codes. The table below lists the polynomials that define each of the two convolutional codes. The second encoder has two inputs and uses two rows of memory registers.

	Outer Convolutional Code	Inner Convolutional Code
Generator Polynomials	$1+D+D^2$ and $1+D^2$	First row: $1+D+D^2$, 0, and $1+D^2$
		Second row: 0, $1+D+D^2$, and $1+D$

	Outer Convolutional Code	Inner Convolutional Code
Feedback Polynomials	$1+D+D^2$	$1+D+D^2$ for each row
Constraint Lengths	3	3 for each row
Code rate	1/2	2/3

Decoding Using an Iterative Process

The decoding portion of this example consists of two APP Decoder blocks, a Random Deinterleaver block, and several other blocks. Together, these blocks form a loop and operate at a rate six times that of the encoding portion of the example. The loop structure and higher rate combine to make the decoding portion an iterative process. Using multiple iterations improves the decoding performance. You can control the number of iterations by setting the **Number of iterations** parameter in the model's Model Parameters block. The default number of iterations is six.

Computations in Each Iteration

In each iteration, the decoding portion of the example decodes the inner convolutional code, deinterleaves the result, and decodes the outer convolutional code. The outer decoder's `L(u)` output signal represents the updated likelihoods of original message bits (that is, input bits to the outer encoder).

The looping strategy in this example enables the inner decoder to benefit in the next iteration from the outer decoder's work. To understand how the loop works, first recall the meanings of these signals:

- The outer decoder's `L(c)` output signal represents the updated likelihoods of code bits from the outer encoder.
- The inner decoder's `L(u)` input represents the likelihoods of input bits to the inner encoder.

The feedback loop recognizes that the primary distinction between these two signals is in the interleaving operation that occurs between the outer and inner encoders. Therefore, the loop interleaves the `L(c)` output of the outer decoder to replicate that interleaving operation, delays the interleaved data to ensure that the inner decoder's two input ports represent data from the same time steps, and resets the `L(u)` input to the inner decoder to zero after every six iterations.

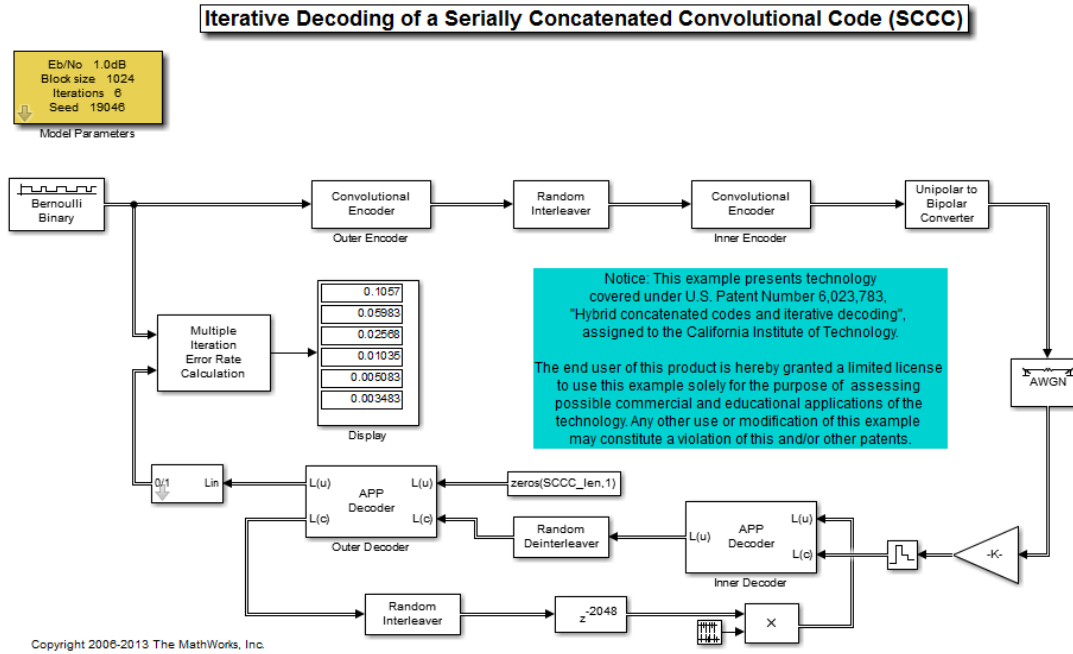
Results of the Iterative Loop

The result of decoding is a 1024-element frame whose elements indicate the likelihood that each of the 1024 message bits was a 0 or a 1. A nonnegative element indicates that the message bit was probably a 1, and a negative element indicates that the message bit was probably a 0. The Hard Decision block converts nonnegative and negative values to 1's and 0's, respectively, so that the results have the same form as the original uncoded binary data.

Results and Displays

The example includes a large Display block that shows error rates after comparing the received data with the transmitted data. The number of error rates in the display is the number of iterations in the decoding process. The first error rate reflects the performance of a decoding process that uses one iteration, the second error rate reflects the performance of a decoding process that uses two iterations, and so on. The series of error rates shows that the error rate generally decreases as the number of iterations increases.

Change the **Eb/No** to 1 dB and run the simulation. Observe that the bit error rates decrease with each iteration.



References

[1] Benedetto, S., D. Divsalar, G. Montorsi, and F. Pollara, "Serial Concatenation of Interleaved Codes: Performance Analysis, Design, and Iterative Decoding," JPL TDA Progress Report, Vol. 42-126, August 1996.

[2] Divsalar, Dariush, and Fabrizio Pollara, Hybrid Concatenated Codes and Iterative Decoding, U. S. Patent No. 6,023,783, Feb. 8, 2000.

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

Punctured Convolutional Encoding

In this section...

This model shows how to use the Convolutional Encoder and Viterbi Decoder blocks to simulate a punctured coding system. The complexity of a Viterbi decoder increases rapidly with the code rate. Puncturing is a technique that allows the encoding and decoding of higher rate codes using standard rate 1/2 encoders and decoders.

The example is somewhat similar to the one that appears in "Soft-Decision Decoding", which shows convolutional coding without puncturing.

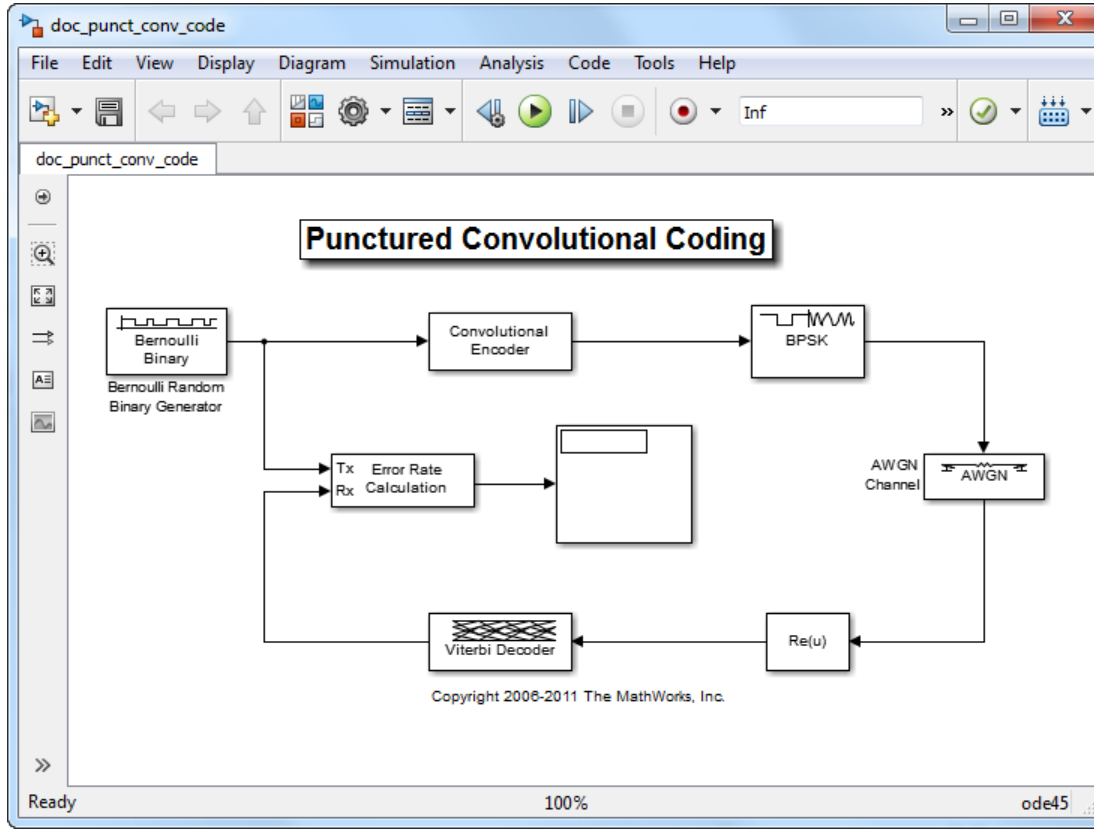
Structure of the Example

This example contains these blocks from the Communications System Toolbox.

- Bernoulli Binary Generator: Create a sequence of random bits to use as a message.
- Convolutional Encoder: Encode the message using the convolutional encoder.
- BPSK Modulator Baseband: Modulate the encoded message.
- AWGN Channel: Pass the modulated signal through a noisy channel.
- Error Rate Calculation: Compute the number of discrepancies between the original and recovered messages.

Open the example, doc_punct_conv_code, by entering the following at the MATLAB command prompt.

```
doc_punct_conv_code
```



Generating Random Data

The Bernoulli Binary Generator block produces the information source for this simulation. The block generates a frame of three random bits at each sample time. The **Samples per frame** parameter determines the number of rows of the output frame.

Convolutional Encoding with Puncturing

The Convolutional Encoder block encodes the data from the Bernoulli Binary Generator. This example uses the same code as described in “Soft-Decision Decoding”.

The puncture pattern is specified by the **Puncture vector** parameter in the mask. The puncture vector is a binary column vector. A 1 indicates that the bit in the corresponding

position of the input vector is sent to the output vector, while a 0 indicates that the bit is removed.

For example, to create a rate 3/4 code from the rate 1/2, constraint length 7 convolutional code, the optimal puncture vector is [1 1 0 1 1 0].' (where the .' after the vector indicates the transpose). Bits in positions 1, 2, 4, and 5 are transmitted, while bits in positions 3 and 6 are removed. Now, for every 3 bits of input, the punctured code generates 4 bits of output (as opposed to the 6 bits produced before puncturing). This makes the rate 3/4.

In this example, the output from the Bernoulli Binary Generator is a column vector of length 3. Because the rate 1/2 Convolutional Encoder doubles the length of each vector, the length of the puncture vector must divide 6.

Transmitting Data

The AWGN Channel block simulates transmission over a noisy channel. The parameters for the block are set in the mask as follows:

- The **Mode** parameter for this block is set to `Signal to noise ratio (Es/No)`.
- The **Es/No** parameter is set to 2 dB. This value typically is changed from one simulation run to the next.
- The preceding modulation block generates unit power signals so the **Input signal power** is set to 1 Watt.
- The **Symbol period** is set to 0.75 seconds because the code has rate 3/4.

Demodulating

In this simulation, the Viterbi Decoder block is set to accept unquantized inputs. As a result, the simulation passes the channel output through a Simulink Complex to Real-Imag block that extracts the real part of the complex samples.

Viterbi Decoding of Punctured Codes

The Viterbi Decoder block is configured to decode the same rate 1/2 code specified in the Convolutional Encoder block.

In this example, the decision type is set to `Unquantized`. For codes without puncturing, you would normally set the **Traceback depth** for this code to a value close to 40. However, for decoding punctured codes, a higher value is required to give the decoder enough data to resolve the ambiguities introduced by the punctures.

Since the punctured bits are not transmitted, there is no information to indicate their values. As a result they are ignored in the decoding process.

The **Puncture vector** parameter indicates the locations of the punctures or the bits to ignore in the decoding process. Each 1 in the puncture vector indicates a transmitted bit while each 0 indicates a puncture or the bit to ignore in the input to the decoder.

In general, the two **Puncture vector** parameters in the Convolutional Encoder and Viterbi Decoder must be the same.

Calculating the Error Rate

The Error Rate Calculation block compares the decoded bits to the original source bits. The output of the Error Rate Calculation block is a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

In the mask for this block, the **Receive delay** parameter is set to 96, because the **Traceback depth** value of 96 in the Viterbi Decoder block creates a delay of 96. If there were other blocks in the model that created delays, the **Receive delay** would equal the sum of all the delays.

BER simulations typically run until a minimum number of errors have occurred, or until the simulation processes a maximum number of bits. The Error Rate Calculation block uses its **Stop simulation** mode to set these limits and to control the duration of the simulation.

Evaluating Results

Generating a bit error rate curve requires multiple simulations. You can perform multiple simulations using the `sim` command. Follow these steps:

- In the model window, remove the Display block and the line connected to its port.
- In the AWGN Channel block, set the **Es/No** parameter to the variable name `EsNodB`.
- In the Error Rate Calculation block, set **Output data** to `Workspace` and then set **Variable name** to `BER_Data`.
- Save the model in your working directory under a different name, such as `my_punct_conv_code.slx`.

- Execute the following code, which runs the simulation multiple times and gathers results.

```
CodeRate = 0.75;
EbNoVec = [2:.5:5];
EsNoVec = EbNoVec + 10*log10(CodeRate);
BERVec = zeros(length(EsNoVec),3);
for n=1:length(EsNoVec),
    EsNodB = EsNoVec(n);
    sim('my_commpunctcnvcod');
    BERVec(n,:) = BER_Data;
end
```

To confirm the validity of the results, compare them to an established performance bound. The bit error rate performance of a rate $r = (n\text{-}1)/n$ punctured code is bounded above by the expression:

$$P_b \le \frac{1}{2(n-1)} \sum_{d=d_{free}}^{\infty} \omega_d \mathrm{erfc}\left(\sqrt{\mathrm{rd}(\mathrm{E_b}/\mathrm{N_0})}\right)$$

In this expression, `erfc` denotes the complementary error function, r is the code rate, and both d_{free} and ω_d are dependent on the particular code. For the rate 3/4 code of this example, $d_{free} = 5$, $\omega_5 = 42$, $\omega_6 = 201$, $\omega_7 = 1492$, and so on. See reference [1] for more details.

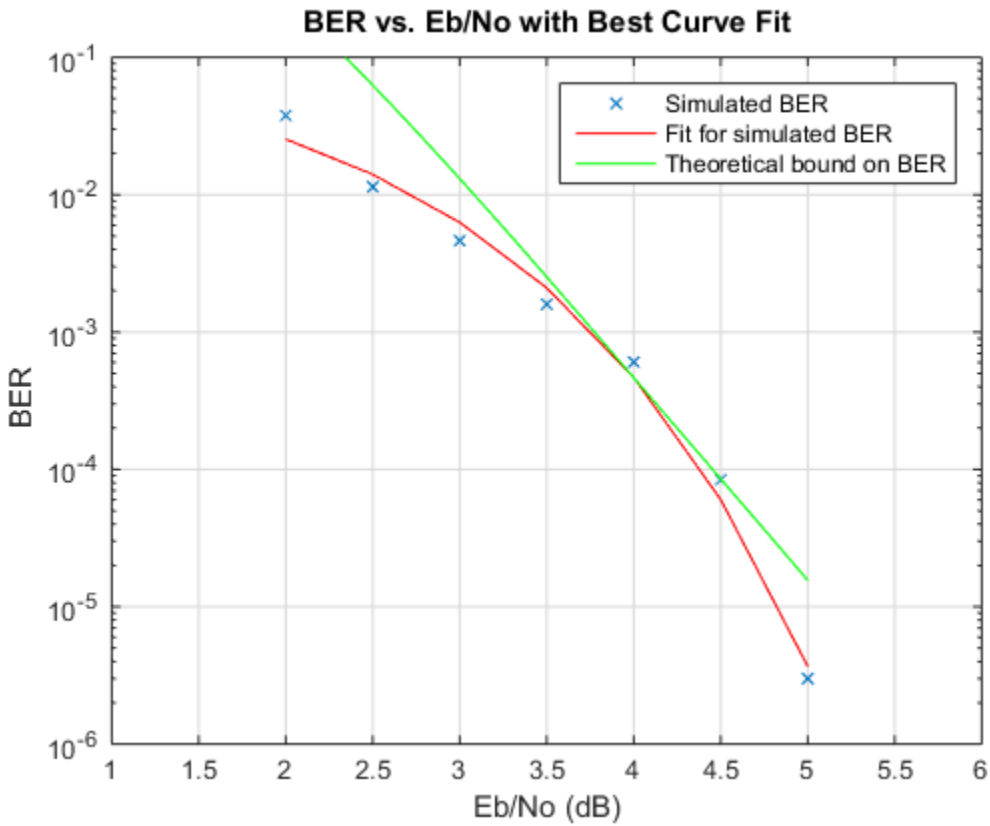
The following commands compute an approximation of this bound using the first seven terms of the summation (the values used for `nerr` are found in Table 2 of reference [2]:

```
dist = [5:11];
nerr = [42 201 1492 10469 62935 379644 2253373];
CodeRate = 3/4;
EbNo_dB = [2:.02:5];
EbNo = 10.0.^(EbNo_dB/10);
arg = sqrt(CodeRate*EbNo'*dist);
bound = nerr*(1/6)*erfc(arg)';
```

To plot the simulation and theoretical results in the same figure, use the commands below.

```
berfit(EbNoVec',BERVec(:,1)); % Curve-fitted simulation results
hold on;
semilogy(EbNo_dB,bound,'g'); % Theoretical results
```

```
legend('Simulated BER','Fit for simulated BER',...
    'Theoretical bound on BER')
```



In some cases, at the lower bit error rates, you might notice simulation results that appear to indicate error rates slightly above the bound. This can result from simulation variance (if fewer than 500 bit errors are observed) or from the finite traceback depth in the decoder.

References

[1] Yasuda, Y., K. Kashiki, and Y. Hirata, "High Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding," IEEE Transactions on Communications, Vol. COM-32, March, 1984, pp. 315-319.

[2] Begin, G., Haccoun, D., and Paquin, C., "Further results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding," IEEE Transactions on Communications, Vol. 38, No. 11, November, 1990, p. 1923.

10

Channel Modeling and RF Impairments

AWGN Channel

In this section...

Section Overview

An AWGN channel adds white Gaussian noise to the signal that passes through it. You can create an AWGN channel in a model using the `comm.AWGNChannel` System object, the AWGN Channel block, or the `awgn` function.

The following examples use an AWGN Channel: `QPSK Transmitter and Receiver` and `scattereyedemo`.

AWGN Channel Noise Level

The relative power of noise in an AWGN channel is typically described by quantities such as

- Signal-to-noise ratio (SNR) per sample. This is the actual input parameter to the `awgn` function.
- Ratio of bit energy to noise power spectral density (EbNo). This quantity is used by BERTool and performance evaluation functions in this toolbox.
- Ratio of symbol energy to noise power spectral density (EsNo)

Relationship Between EsNo and EbNo

The relationship between EsNo and EbNo, both expressed in dB, is as follows:

$$E_s / N_0 \text{ (dB)} = E_b / N_0 \text{ (dB)} + 10\log_{10}(k)$$

where k is the number of information bits per symbol.

In a communication system, k might be influenced by the size of the modulation alphabet or the code rate of an error-control code. For example, if a system uses a rate-1/2 code and 8-PSK modulation, then the number of information bits per symbol (k) is the product of the code rate and the number of coded bits per modulated symbol: (1/2) $\log_2(8)$ = 3/2.

In such a system, three information bits correspond to six coded bits, which in turn correspond to two 8-PSK symbols.

Relationship Between EsNo and SNR

The relationship between EsNo and SNR, both expressed in dB, is as follows:

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

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

where T_{sym} is the signal's symbol period and T_{samp} is the signal's sampling period.

For example, if a complex baseband signal is oversampled by a factor of 4, then EsNo exceeds the corresponding SNR by $10\log_{10}(4)$.

Derivation for Complex Input Signals

You can derive the relationship between EsNo and SNR for complex input signals as follows:

$$\begin{aligned} E_s / N_0 \text{ (dB)} &= 10\log_{10}\left((S \cdot T_{sym}) / (N / B_n)\right) \\ &= 10\log_{10}\left((T_{sym} F_s) \cdot (S / N)\right) \\ &= 10\log_{10}\left(T_{sym} / T_{samp}\right) + SNR \text{ (dB)} \end{aligned}$$

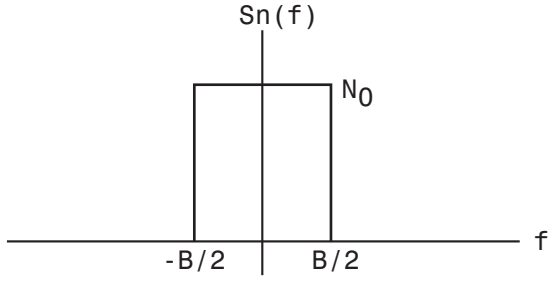
where

- S = Input signal power, in watts
- N = Noise power, in watts
- B_n = Noise bandwidth, in Hertz
- F_s = Sampling frequency, in Hertz

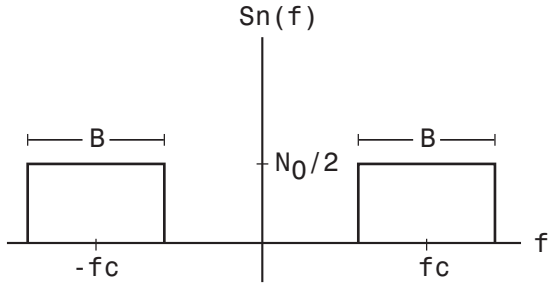
Note that $B_n = F_s = 1/T_{samp}$.

Behavior for Real and Complex Input Signals

The following figures illustrate the difference between the real and complex cases by showing the noise power spectral densities $S_n(f)$ of a real bandpass white noise process and its complex lowpass equivalent.



Complex Lowpass Noise Power Spectral Density



Real Bandpass Noise Power Spectral Density

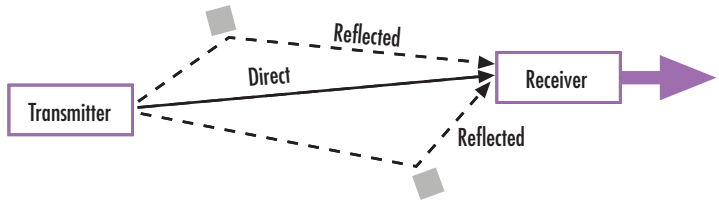
Fading Channels

In this section...

Overview of Fading Channels

Using Communications System Toolbox you can implement fading channels using objects or blocks. Rayleigh and Rician fading channels are useful models of real-world phenomena in wireless communications. These phenomena include multipath scattering effects, time dispersion, and Doppler shifts that arise from relative motion between the transmitter and receiver. This section gives a brief overview of fading channels and describes how to implement them using the toolbox.

The figure below depicts direct and major reflected paths between a stationary radio transmitter and a moving receiver. The shaded shapes represent reflectors such as buildings.



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

near the mobile. These irresolvable components combine at the receiver and give rise to the phenomenon known as *multipath fading*. Due to this phenomenon, each major path behaves as a discrete fading path. Typically, the fading process is characterized by a Rayleigh distribution for a nonline-of-sight path and a Rician distribution for a line-of-sight path.

The relative motion between the transmitter and receiver causes Doppler shifts. Local scattering typically comes from many angles around the mobile. This scenario causes a range of Doppler shifts, known as the *Doppler spectrum*. The *maximum* Doppler shift corresponds to the local scattering components whose direction exactly opposes the mobile's trajectory.

Implement Fading Channel Using an Object

A baseband channel model for multipath propagation scenarios that you implement using objects includes:

- N discrete fading paths, each with its own delay and average power gain. A channel for which N = 1 is called a *frequency-flat fading channel*. A channel for which N > 1 is experienced as a *frequency-selective fading channel* by a signal of sufficiently wide bandwidth.
- A Rayleigh or Rician model for each path.
- Default channel path modeling using a Jakes Doppler spectrum, with a maximum Doppler shift that can be specified. Other types of Doppler spectra allowed (identical or different for all paths) include: flat, restricted Jakes, asymmetrical Jakes, Gaussian, bi-Gaussian, and rounded.

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

Some additional information about typical values for delays and gains is in “Choose Realistic Channel Property Values” on page 10-20

Implement Fading Channel Using a Block

The Channels block library includes Rayleigh and Rician fading blocks that can simulate real-world phenomena in mobile communications. These phenomena include multipath scattering effects, as well as Doppler shifts that arise from relative motion between the transmitter and receiver.

Note To model a channel that involves both fading and additive white Gaussian noise, use a fading channel block connected in series with the AWGN Channel block, where the fading channel block comes first.

The table below indicates the situations in which each fading channel block is appropriate.

Signal Path	Channel Block
Direct line-of-sight path from transmitter to receiver	Multipath Rician Fading Channel
One or more major reflected paths from transmitter to receiver	Multipath Rayleigh Fading Channel

In the case of multiple major reflected paths, a single instance of the Multipath Rayleigh Fading Channel block can model all of them simultaneously. The number of paths that the block uses is the length of either the **Delay vector** or the **Gain vector** parameter, whichever length is larger. (If both of these parameters are vectors, they must have the same length; if exactly one of these parameters is a scalar, the block expands it into a vector whose size matches that of the other **vector** parameter.)

Choosing appropriate block parameters for your situation is important. For more details about the parameters of fading channel blocks, see

- The reference pages for the Multipath Rayleigh Fading Channel block and the Multipath Rician Fading Channel block
- The “Choose Realistic Channel Property Values” section under “Configuring Channel Objects” in the Communications System Toolbox documentation

Compensate for Fading Response

A communication system involving a fading channel usually requires component(s) that compensate for the fading response. Typical approaches to compensate for fading include:

- Differential modulation or a one-tap equalizer helps compensate for a frequency-flat fading channel. See the M-DPSK Modulator Baseband block Help page or the example in “Compare Empirical Results to Theoretical Results” on page 10-25 for information about implementing differential modulation.
- An equalizer with multiple taps helps compensate for a frequency-selective fading channel. See “Equalization” for more information.

The Communications Link with Adaptive Equalization example illustrates why compensating for a fading channel is necessary.

Visualize a Fading Channel

You can plot a fading channel's characteristics using channel visualization tools.

For communication systems that you implement using objects, see “Channel Visualization”.

For communication systems that you implement using blocks, there are two ways to visualize fading channel response. One way is to double-click the block during a simulation. The second way is to select **Open channel visualization at start of simulation** in the block dialog box.

Methodology for Simulating Multipath Fading Channels:

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

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

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

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

In the equations above:

- T_s is the input sample period to the channel.

- $\{\tau_k\}$, where $1 \le k \le K$, is the set of path delays. K is the total number of paths in the multipath fading channel.
- $\{a_k\}$, where $1 \le k \le K$, is the set of complex path gains of the multipath fading channel. These path gains are uncorrelated with each other.
- N_1 and N_2 are chosen so that $|g_n|$ is small when n is less than $-N_1$ or greater than N_2.

Two techniques, filtered Gaussian noise and sum-of-sinusoids, are used to generate the set of complex path gains, a_k.

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

Filtered Gaussian Noise Technique

1. A complex uncorrelated (white) Gaussian process with zero mean and unit variance is generated in discrete time.
2. The complex Gaussian process is filtered by a Doppler filter with frequency response $H(f) = \sqrt{S(f)}$, where $S(f)$ denotes the desired Doppler power spectrum.
3. The filtered complex Gaussian process is interpolated so that its sample period is consistent with that of the input signal. A combination of linear and polyphase interpolation is used.

Sum-of-sinusoids Technique

1. Mutually uncorrelated Rayleigh fading waveforms are generated using the method described in [2], where $i = 1$ corresponds to the in-phase component and $i = 2$ corresponds to the quadrature component.

$$z_k(t) = \mu_k^{(1)}(t) + j\mu_k^{(2)}(t), \quad k = 1,2,\ldots,K$$

$$\mu_k^{(i)}(t) = \sqrt{\frac{2}{N_k}} \sum_{n=1}^{N_k} \cos\left(2\pi f_{k,n}^{(i)} t + \theta_{k,n}^{(i)}\right), \quad i = 1,2$$

Where

- N_k specifies the number of sinusoids used to model a single path.

- $f_{k,n}^{(i)}$ is the discrete Doppler frequency and is calculated for each sinusoid component within a single path.
- $\theta_{k,n}^{(i)}$ is the phase of the n^{th} component of $\mu_k^{(i)}$ and is an i.i.d. random variable having a uniform distribution over the interval $(0, 2\pi]$.
- t is the fading process time.

The discrete Doppler frequencies, $f_{k,n}^{(i)}$, with maximum shift $f_{\max}$ are given by

$$\begin{aligned} f_{k,n}^{(i)} &= f_{\max}\cos\left(\alpha_{k,n}^{(i)}\right) \\ &= f_{\max}\cos\left[\frac{\pi}{2N_k}\left(n-\frac{1}{2}\right)+\alpha_{k,0}^{(i)}\right] \end{aligned}$$

where

$$\alpha_{k,0}^{(i)} \triangleq (-1)^{i-1}\frac{\pi}{4N_k}\cdot\frac{k}{K+2}, \quad i=1,2 \text{ and } k=1,2,\ldots,K$$

2 In order to advance the fading process in time, an initial time parameter, t_{init}, is introduced. The fading waveforms become

$$\mu_k^{(i)}(t) = \sqrt{\frac{2}{N_k}}\sum_{n=1}^{N_k}\cos\left(2\pi f_{k,n}^{(i)}\left(t+t_{init}\right)+\theta_{k,n}^{(i)}\right), \quad i=1,2$$

When $t_{init} = 0$, the fading process starts at time zero. A positive value of t_{init} advances the fading process relative to time zero while maintaining its continuity.

3 Channel fading samples are generated using the $GMEDS_1$ [2] algorithm.

Calculate Complex Coefficients

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

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

where

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

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

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

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

At the input to the band-limited multipath channel model, the transmitted symbols must be oversampled by a factor at least equal to the bandwidth expansion factor introduced by pulse shaping. For example, if sinc pulse shaping is used, for which the bandwidth of the pulse-shaped signal is equal to the symbol rate, then the bandwidth expansion factor is 1, and at least one sample per symbol is required at the input to the channel. If a raised cosine (RC) filter with a factor in excess of 1 is used, for which the bandwidth of the pulse-shaped signal is equal to twice the symbol rate, then the bandwidth expansion factor is 2, and at least two samples per symbol are required at the input to the channel.

For additional information, see the article A Matlab-based Object-Oriented Approach to Multipath Fading Channel Simulation, located on MATLABCentral.

References

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

[2] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. “Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated

Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Specify Fading Channels

Communications System Toolbox models a fading channel as a linear FIR filter. Filtering a signal using a fading channel involves these steps:

1. Create a channel object that describes the channel that you want to use. A channel object is a type of MATLAB variable that contains information about the channel, such as the maximum Doppler shift.
2. Adjust properties of the channel object, if necessary, to tailor it to your needs. For example, you can change the path delays or average path gains.

   Note: Setting the maximum path delay greater than 100 samples may generate an 'Out of memory' error.
3. Apply the channel object to your signal using the `filter` function.

This section describes how to define, inspect, and manipulate channel objects. The topics are:

Creating Channel Objects

The `rayleighchan` and `ricianchan` functions create fading channel objects. The table below indicates the situations in which each function is suitable.

Function	Object	Situation Modeled
`rayleighchan`	Rayleigh fading channel object	One or more major reflected paths
`ricianchan`	Rician fading channel object	One direct line-of-sight path, possibly combined with one or more major reflected paths

For example, the command below creates a channel object representing a Rayleigh fading channel that acts on a signal sampled at 100,000 Hz. The maximum Doppler shift of the channel is 130 Hz.

```
c1 = rayleighchan(1/100000,130); % Rayleigh fading channel object
```

The object `c1` is a valid input argument for the `filter` function. To learn how to use the `filter` function to filter a signal using a channel object, see "Use Fading Channels" on page 10-23.

Duplicate and Copy Objects

Another way to create an object is to duplicate an existing object and then adjust the properties of the new object, if necessary. If you do this, it is important to use a `copy` command such as

```
c2 = copy(c1); % Copy c1 to create an independent c2.
```

instead of `c2 = c1`. The `copy` command creates a copy of `c1` that is independent of `c1`. By contrast, the command `c2 = c1` creates `c2` as merely a reference to `c1`, so that `c1` and `c2` always have indistinguishable content.

Display Object Properties

A channel object has numerous properties that record information about the channel model, about the state of a channel that has already filtered a signal, and about the channel's operation on a future signal. You can view the properties in these ways:

- To view all properties of a channel object, enter the object's name in the Command Window.
- To view a specific property of a channel object or to assign the property's value to a variable, enter the object's name followed by a dot (period), followed by the name of the property.

In the example below, entering `c1` causes MATLAB to display all properties of the channel object `c1`. Some of the properties have values from the `rayleighchan` command that created `c1`, while other properties have default values.

```
c1 = rayleighchan(1/100000,130); % Create object.
c1 % View all properties of c1.
g = c1.PathGains % Retrieve the PathGains property of c1.
```

The output is

```
c1 =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-005
         DopplerSpectrum: [1x1 doppler.jakes]
         MaxDopplerShift: 130
              PathDelays: 0
           AvgPathGaindB: 0
      NormalizePathGains: 1
            StoreHistory: 0
          StorePathGains: 0
               PathGains: -0.0428 + 0.4732i
      ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0

g =

   -0.0428 + 0.4732i
```

A Rician fading channel object has an additional property that does not appear above, namely, a scalar `KFactor` property.

For more information about what each channel property means, see the reference page for the `rayleighchan` or `ricianchan` function.

Change Object Properties

To change the value of a writeable property of a channel object, issue an assignment statement that uses dot notation on the channel object. More specifically, dot notation means an expression that consists of the object's name, followed by a dot, followed by the name of the property.

The example below illustrates how to change the `ResetBeforeFiltering` property, indicating you do not want to reset the channel before each filtering operation.

```
c1 = rayleighchan(1/100000,130) % Create object.
c1.ResetBeforeFiltering = 0 % Do not reset before filtering.
```

The output below displays all the properties of the channel object before and after the change in the value of the `ResetBeforeFiltering` property. In the second listing of properties, the `ResetBeforeFiltering` property has the value `0`.

```
c1 =

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

c1 =

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

Note: Some properties of a channel object are read-only. For example, you cannot assign a new value to the `NumSamplesProcessed` property because the channel automatically counts the number of samples it has processed since the last reset.

Relationships Among Channel Object Properties

Some properties of a channel object are related to each other such that when one property's value changes, another property's value must change in some corresponding way to keep the channel object consistent. For example, if you change the vector length of `PathDelays`, then the value of `AvgPathGaindB` must change so that its vector length

equals that of the new value of `PathDelays`. This is because the length of each of the two vectors equals the number of discrete paths of the channel. For details about linked properties and an example, see the reference page for `rayleighchan` or `ricianchan`.

Specify Doppler Spectrum of Fading Channel

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

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

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

Create a Doppler Object

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

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

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

Duplicate Doppler Objects

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

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

creates a Doppler object `d2` with the same properties as that of `d1`. `d1` and `d2` are then separate instances of a Doppler object, in that modifying either one will not affect the

other. Using `d1 = d2` instead will cause both `d1` and `d2` to reference the same instance of a Doppler object, in that modifying either one will cause the same modification to the other.

View and Change Doppler Object Properties

The syntax for viewing and changing Doppler object properties is the same as for the case of channel objects (see "Display Object Properties" on page 10-13 and "Change Object Properties" on page 10-14). The function `disp` can be used with Doppler objects to display their properties.

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

```
dr = doppler.rounded

dr =

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

dr.CoeffRounded(3) = 0.8250

dr =

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

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

Use Doppler Objects Within Channel Objects

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

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

h =
```

```
             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0417e-004
         DopplerSpectrum: [1x1 doppler.jakes]
         MaxDopplerShift: 100
              PathDelays: 0
           AvgPathGaindB: 0
     NormalizePathGains: 1
           StoreHistory: 0
          StorePathGains: 0
               PathGains: -0.4007 - 0.2748i
     ChannelFilterDelay: 0
   ResetBeforeFiltering: 1
    NumSamplesProcessed: 0

>> dop_flat = doppler.flat

dop_flat =

    SpectrumType: 'Flat'

>> h.DopplerSpectrum = dop_flat

h =

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

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

```
>> h = ricianchan(1/9600, 100, 2);
```

```
>> h.DopplerSpectrum = doppler.gaussian(0.3)

h =

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

>> h.DopplerSpectrum

ans =

     SpectrumType: 'Gaussian'
    SigmaGaussian: 0.3000

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

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

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

h =

            ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0417e-004
         DopplerSpectrum: [3x1 doppler.baseclass]
         MaxDopplerShift: 100
              PathDelays: [0 1.0000e-004 2.1000e-004]
```

```
          AvgPathGaindB: [0 0 0]
      NormalizePathGains: 1
            StoreHistory: 0
        StorePathGains: 0
             PathGains: [0.4233 - 0.1113i -0.0785 + 0.1667i
                         -0.2064 + 0.3531i]
      ChannelFilterDelay: 3
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0

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

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

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

Configure Channel Objects

Before you filter a signal using a channel object, make sure that the properties of the channel have suitable values for the situation you want to model. This section offers some guidelines to help you choose realistic values that are appropriate for your modeling needs. The topics are

- "Choose Realistic Channel Property Values" on page 10-20
- "Configure Channel Objects Based on Simulation Needs" on page 10-22

The syntaxes for viewing and changing values of properties of channel objects are described in "Specify Fading Channels" on page 10-12.

Choose Realistic Channel Property Values

Here are some tips for choosing property values that describe realistic channels:

Path Delays

- By convention, the first delay is typically set to zero. The first delay corresponds to the first arriving path.

- For indoor environments, path delays after the first are typically between 1 ns and 100 ns (that is, between 1e-9 s and 1e-7 s).
- For outdoor environments, path delays after the first are typically between 100 ns and 10 μs (that is, between 1e-7 s and 1e-5 s). Very large delays in this range might correspond, for example, to an area surrounded by mountains.

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

- The ability of a signal to resolve discrete paths is related to its bandwidth. If the difference between the largest and smallest path delays is less than about 1% of the symbol period, then the signal experiences the channel as if it had only one discrete path.

Average Path Gains

- The average path gains in the channel object indicate the average power gain of each fading path. In practice, an average path gain value is a large negative dB value. However, computer models typically use average path gains between -20 dB and 0 dB.
- The dB values in a vector of average path gains often decay roughly linearly as a function of delay, but the specific delay profile depends on the propagation environment.
- To ensure that the expected value of the path gains' total power is 1, you can normalize path gains via the channel object's `NormalizePathGains` property.

Maximum Doppler Shifts

- Some wireless applications, such as standard GSM (Global System for Mobile Communication) systems, prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed v (m/s), then the maximum Doppler shift is calculated as follows, where f is the transmission carrier frequency in Hertz and c is the speed of light (3e8 m/s).

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

- Based on this formula in terms of the speed of the mobile, a signal from a moving car on a freeway might experience a maximum Doppler shift of about 80 Hz, while a signal from a moving pedestrian might experience a maximum Doppler shift of about 4 Hz. These figures assume a transmission carrier frequency of 900 MHz.

- A maximum Doppler shift of 0 corresponds to a static channel that comes from a Rayleigh or Rician distribution.

K-Factor for Rician Fading Channels

- The Rician K-factor specifies the ratio of specular-to-diffuse power for a direct line-of-sight path. The ratio is expressed linearly, not in dB.
- For Rician fading, the K-factor is typically between 1 and 10.
- A K-factor of 0 corresponds to Rayleigh fading.

Doppler Spectrum Parameters

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

Configure Channel Objects Based on Simulation Needs

Here are some tips for configuring a channel object to customize the filtering process:

- If your data is partitioned into a series of vectors (that you process within a loop, for example), you can invoke the `filter` function multiple times while automatically saving the channel's state information for use in a subsequent invocation. The state information is visible to you in the channel object's `PathGains` and `NumSamplesProcessed` properties, but also involves properties that are internal rather than visible.

  Note: To maintain continuity from one invocation to the next, you must set the `ResetBeforeFiltering` property of the channel object to `0`.

- If you set the `ResetBeforeFiltering` property of the channel object to `0` and want the randomness to be repeatable, use the `reset` function before filtering any signals to reset both the channel and the state of the internal random number generator.
- If you want to reset the channel before a filtering operation so that it does not use any previously stored state information, either use the `reset` function or set the `ResetBeforeFiltering` property of the channel object to `1`. The former method resets the channel object once, while the latter method causes the `filter` function to reset the channel object each time you invoke it.
- If you want to normalize the fading process so that the expected value of the path gains' total power is 1, set the `NormalizePathGains` property of the channel object to `1`.

Use Fading Channels

After you have created a channel object as described in "Specify Fading Channels" on page 10-12, you can use the `filter` function to pass a signal through the channel. The arguments to `filter` are the channel object and the signal. At the end of the filtering operation, the channel object retains its state so that you can find out the final path gains or the total number of samples that the channel has processed since it was created or reset. If you configured the channel to avoid resetting its state before each new filtering operation (`ResetBeforeFiltering` is `0`), then the retention of state information is important for maintaining continuity between successive filtering operations.

For an example that illustrates the basic syntax and state retention, see "Power of a Faded Signal" on page 10-23.

If you want to use the channel visualization tool to plot the characteristics of a channel object, you need to set the `StateHistory` property of the channel object to `1` so that it is populated with plot information. See "Channel Visualization" for details.

Rayleigh Fading Channel

The following examples use fading channels:

- "Power of a Faded Signal" on page 10-23
- "Compare Empirical Results to Theoretical Results" on page 10-25
- "Work with Delays" on page 10-26
- "Filter Using a Loop" on page 10-27
- "Store Channel State History" on page 10-28
- "Use the Channel Visualization Tool" on page 10-29

Power of a Faded Signal

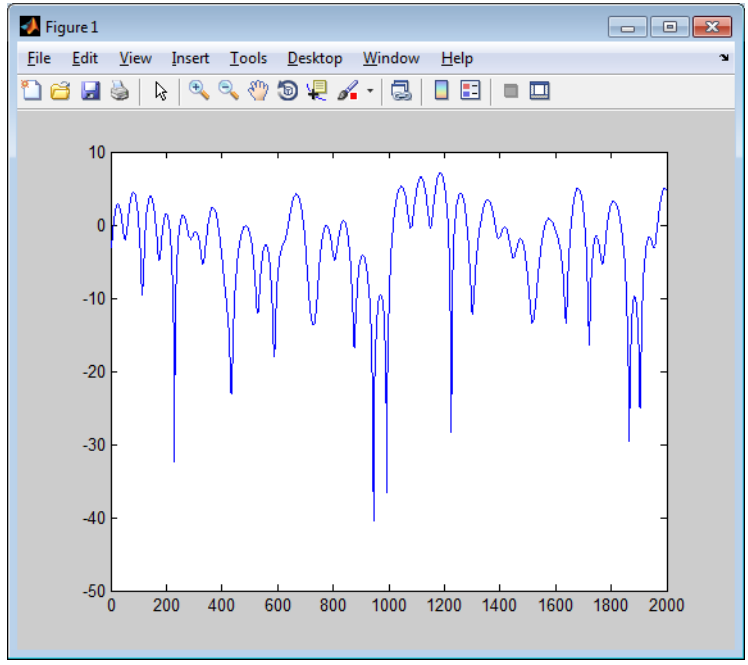
The code below plots a faded signal's power (versus sample number). The code also illustrates the syntax of the `filter` and `rayleighchan` functions and the state retention of the channel object. Notice from the output that `NumSamplesProcessed` equals the number of elements in `sig`, the signal.

```
c = rayleighchan(1/10000,100);
sig = 1i*ones(2000,1);  % Generate signal
y = filter(c,sig);      % Pass signal through channel
c                       % Display all properties of the channel
```

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

The output and the plot follow.

```
c =

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



Compare Empirical Results to Theoretical Results

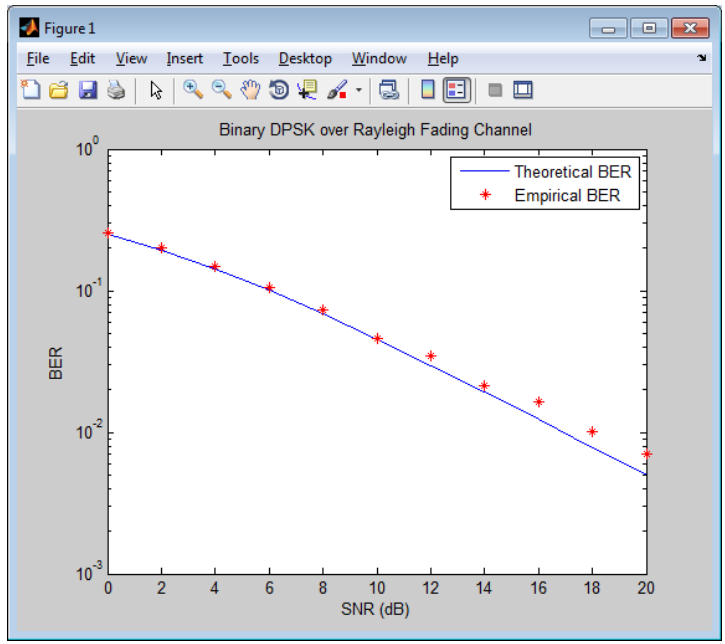
The code below creates a frequency-flat Rayleigh fading channel object and uses it to process a DBPSK signal consisting of a single vector. The example continues by computing the bit error rate of the system for different values of the signal-to-noise ratio. Notice that the example uses `filter` before `awgn`; this is the recommended sequence to use when you combine fading with AWGN.

```
% Create Rayleigh fading channel object.
chan = rayleighchan(1/10000,100);

% Generate data and apply fading channel.
M = 2; % DBPSK modulation order
hMod = comm.DBPSKModulator;       % Create a DPSK modulator
hDemod = comm.DBPSKDemodulator;     % Create a DPSK demodulator
tx = randi([0 M-1],50000,1);        % Generate a random bit stream
dpskSig = step(hMod, tx);       % DPSK modulate the signal
fadedSig = filter(chan,dpskSig);    % Apply the channel effects

% Compute error rate for different values of SNR.
SNR = 0:2:20; % Range of SNR values, in dB.
numSNR = length(SNR);
berVec = zeros(3, numSNR);

% Create an AWGNChannel and ErrorRate calculator System object
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');
hErrorCalc = comm.ErrorRate;

for n = 1:numSNR
    hChan.SNR = SNR(n);
   rxSig = step(hChan,fadedSig);   % Add Gaussian noise
   rx = step(hDemod, rxSig);  % Demodulate
   reset(hErrorCalc)
   % Compute error rate.
   berVec(:,n) = step(hErrorCalc,tx,rx);
end
BER = berVec(1,:);
% Compute theoretical performance results, for comparison.
BERtheory = berfading(SNR,'dpsk',M,1);

% Plot BER results.
semilogy(SNR,BERtheory,'b-',SNR,BER,'r*');
legend('Theoretical BER','Empirical BER');
xlabel('SNR (dB)'); ylabel('BER');
```

```
title('Binary DPSK over Rayleigh Fading Channel');
```

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

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



Work with Delays

The value of a channel object's `ChannelFilterDelay` property is the number of samples by which the output of the channel lags the input. If you compare the input and output data sets directly, you must take the delay into account by using appropriate truncating or padding operations.

The example illustrates one way to account for the delay before computing a bit error rate.

```
M = 2;                          % DQPSK modulation order
bitRate = 50000;
```

```
hMod = comm.DBPSKModulator;       % Create a DPSK modulator
hDemod = comm.DBPSKDemodulator;     % Create a DPSK demodulator

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

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

% Compute bit error rate, taking delay into account.
hErrorCalc = comm.ErrorRate('ReceiveDelay', delay);
berVec = step(hErrorCalc,tx,rx);
ber = berVec(1)
num = berVec(2)
```

The output below shows that the error rate is small. If the example had not compensated for the channel delay, the error rate would have been close to 1/2.

```
num =

   845


ber =

    0.0170
```

Filter Using a Loop

The section "Configure Channel Objects Based on Simulation Needs" on page 10-22 indicates how to invoke the `filter` function multiple times while maintaining continuity from one invocation to the next. The example below invokes `filter` within a loop and uses the small data sets from successive iterations to create an animated effect. The particular channel in this example is a Rayleigh fading channel with two discrete major paths.

```
% Set up parameters.
M = 4;              % QPSK modulation order
hMod = comm.QPSKModulator;
bitRate = 50000;    % Data rate is 50 kb/s
numTrials = 125;    % Number of iterations of loop
```

```
% Create Rayleigh fading channel object.
ch = rayleighchan(1/bitRate,4,[0 2e-5],[0 -9]);
% Indicate that FILTER should not reset the channel
% in each iteration below.
ch.ResetBeforeFiltering = 0;

% Initialize scatter plot.
hConst = comm.ConstellationDiagram;

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

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

The scatter plot changes with each iteration of the loop, and the exact content varies because the fading process involves random numbers.

Store Channel State History

By default, the `PathGains` property of a channel object stores the current complex path gain vector.

Setting the `StoreHistory` property of a channel to true makes it store the last N path gain vectors, where N is the length of the vector processed through the channel. The following code illustrates this property

```
h = rayleighchan(1/100000, 130);  % Rayleigh channel
tx = randi([0 1],10,1);           % Random bit stream
hmod = comm.DBPSKModulator;       % Create DBPSK Modulator
dpskSig = step(hmod,tx);          % Process data by calling the step method
h.StoreHistory = true;            % Allow states to be stored
y = filter(h, dpskSig);           % Run signal through channel
h.PathGains                       % Display the stored path gains data
```

This example generates an output similar to the following:

```
 -0.7601 - 1.1853i
```

```
  -0.7540 - 1.1822i
  -0.7480 - 1.1791i
  -0.7419 - 1.1759i
  -0.7358 - 1.1728i
  -0.7298 - 1.1696i
  -0.7237 - 1.1665i
  -0.7177 - 1.1634i
  -0.7115 - 1.1599i
  -0.7053 - 1.1565i

ans =

   0.0788 - 0.5305i
```

The last element is the current path gain of the channel.

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

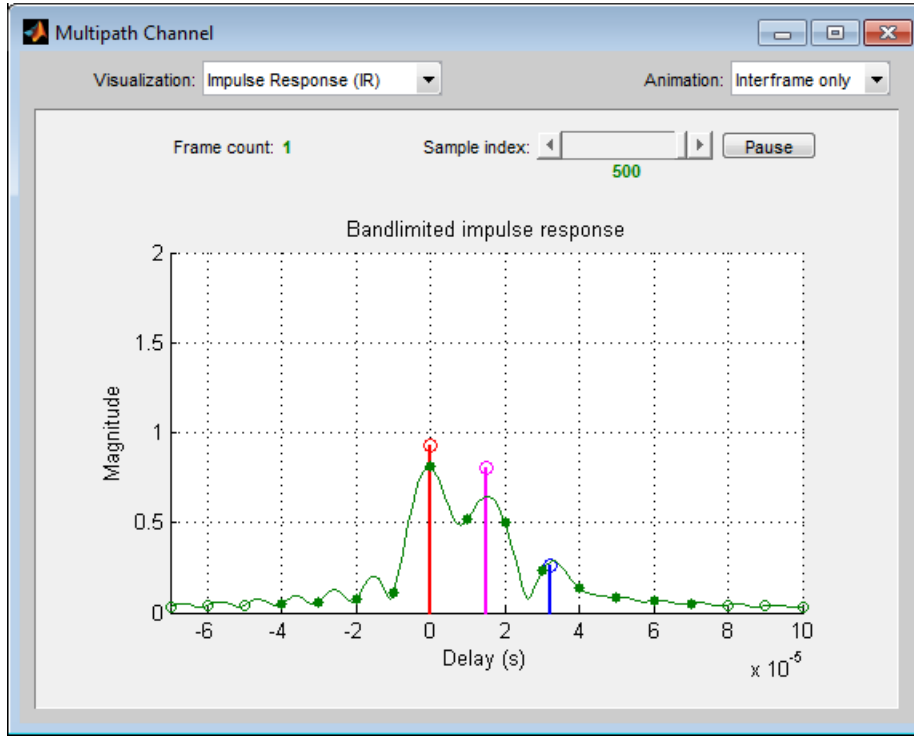
Use the Channel Visualization Tool

Communications System Toolbox software provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See “Fading Channels” on page 10-5 for a description of fading channels and objects.

To open the channel visualization tool, type `plot(h)` at the command line, where `h` is a channel object that contains plot information. To populate a channel object with plot information, run a signal through it after setting its `StoreHistory` property to `true`.

For example, the following code opens the channel visualization tool showing a three-path Rayleigh channel through which a random signal is passed:

```
% Three-Path Rayleigh channel
h = rayleighchan(1/100000, 130, [0 1.5e-5 3.2e-5], [0, -3, -3]);
tx = randi([0 1],500,1);          % Random bit stream
hmod = comm.DBPSKModulator;       % Create DBPSKModulator
dpskSig = step(hmod,tx);          % DPSK signal
h.StoreHistory = true;            % Allow states to be stored
y = filter(h, dpskSig);           % Run signal through channel
plot(h);                          % Call Channel Visualization Tool
```



See “Examples of Using the Channel Visualization Tool” on page 10-39 for the basic usage cases of the channel visualization tool.

This tool can also be accessed from Communications System Toolbox software.

Parts of the GUI

The **Visualization** pull-down menu allows you to choose the visualization method.

The **Frame count** counter shows the index of the current frame. It shows the number of frames processed by the filter method since the channel object was constructed or reset. A *frame* is a vector of M elements, interpreted to be M successive samples that are uniformly spaced in time, with a sample period equal to that specified for the channel.

The **Sample index** slider control indicates which channel snapshot is currently being displayed, while the **Pause** button pauses a running animation until you click it again. The slider control and **Pause** button apply to all visualizations except the `Doppler Spectrum`.

The **Animation** pull-down menu allows you to select how you want to display the channel snapshots within each frame. Setting this to `Slow` makes the tool show channel snapshots in succession, starting at the sample set by the **Sample index** slider control. Selecting `Medium` or `Fast` makes the tool show fewer uniformly spaced snapshots, allowing you to go through the channel snapshots more rapidly. Selecting `Interframe only` (the default selection) prevents automatic animation of snapshots within the same frame. The **Animation** menu applies to all visualizations except the `Doppler Spectrum`.

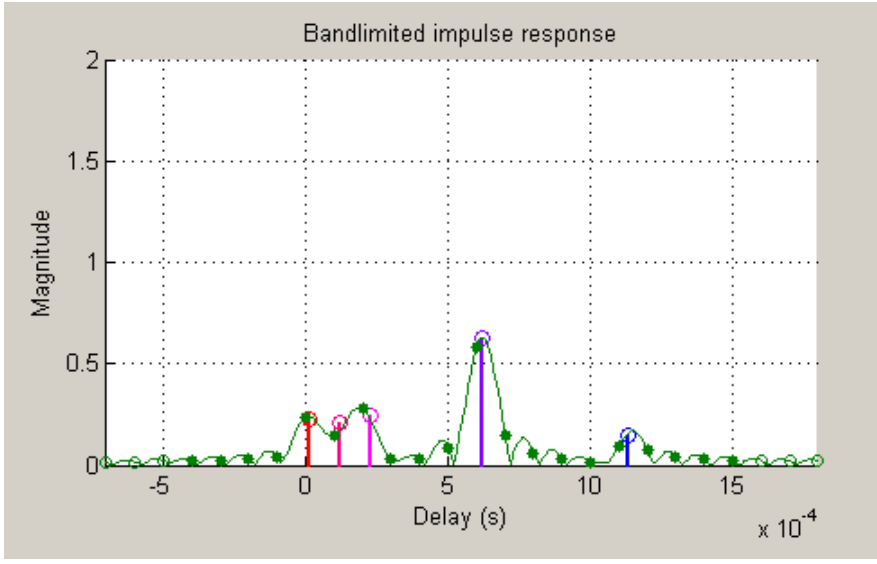
Visualization Options

The channel visualization tool plots the characteristics of a filter in various ways. Simply choose the visualization method from the **Visualization** menu, and the plot updates itself automatically.

The following visualization methods are currently available:

Impulse Response (IR)

This plot shows the magnitudes of two impulse responses: the multipath response (infinite bandwidth) and the bandlimited channel response.



The multipath response is represented by stems, each corresponding to one multipath component. The component with the smallest delay value is shown in red, and the

component with the largest delay value is shown in blue. Components with intermediate delay values are shades between red and blue, becoming more blue for larger delays.

The bandlimited channel response is represented by the green curve. This response is the result of convolving the multipath impulse response, described above, with a sinc pulse of period, T, equal to the input signal's sample period.

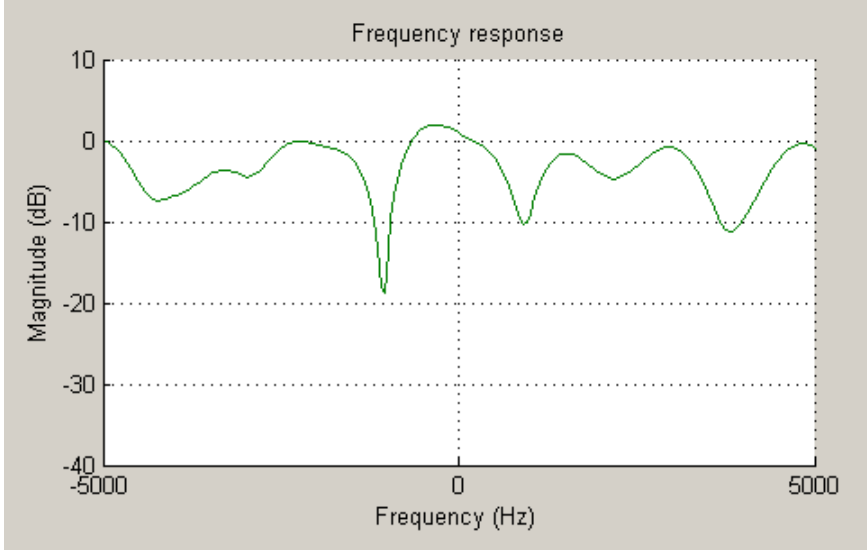
The solid green circles represent the channel filter response sampled at rate 1/T. The output of the channel filter is the convolution of the input signal (sampled at rate 1/T) with this discrete-time FIR channel filter response. For computational speed, the response is truncated.

The hollow green circles represent sample values not captured in the channel filter response that is used for processing the input signal.

Note that these impulse responses vary over time. You can use the slider to visualize how the impulse response changes over time for the current frame (i.e., input signal vector over time).

Frequency Response (FR)

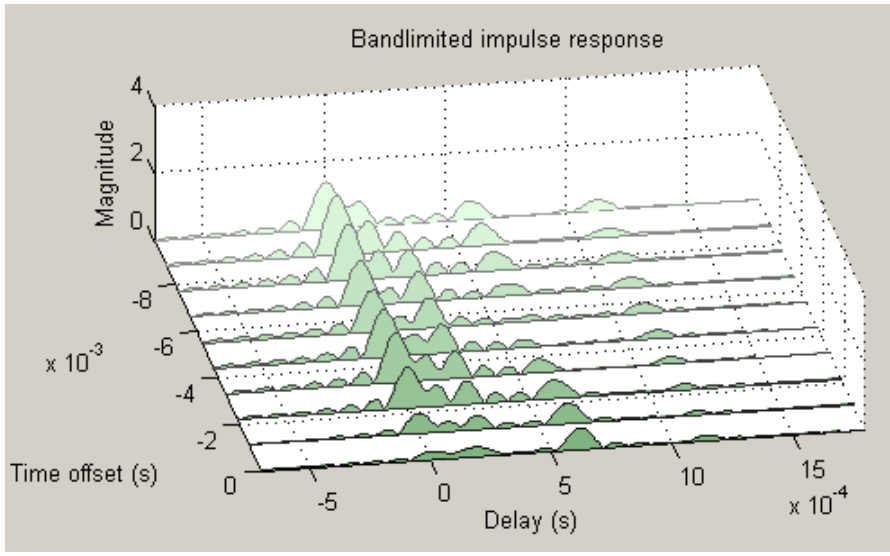
This plot shows the magnitude (in dB) of the frequency response of the multipath channel over the signal bandwidth.



As with the impulse response visualization, you can visualize how this frequency response changes over time.

IR Waterfall

This plot shows the evolution of the magnitude impulse response over time.

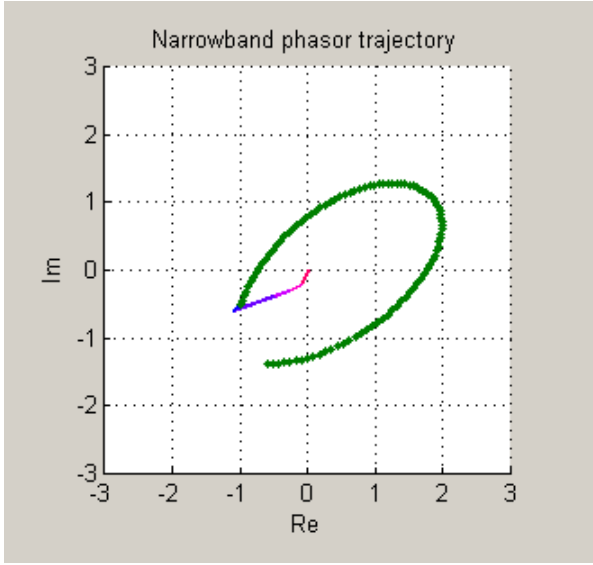


It shows 10 snapshots of the bandlimited channel impulse response within the last frame, with the darkest green curve showing the current response.

The time offset is the time of the channel snapshot relative to the current response time.

Phasor Trajectory

This plot shows phasors (vectors representing magnitude and phase) for each multipath component, using the same color code that was used for the impulse response plot.

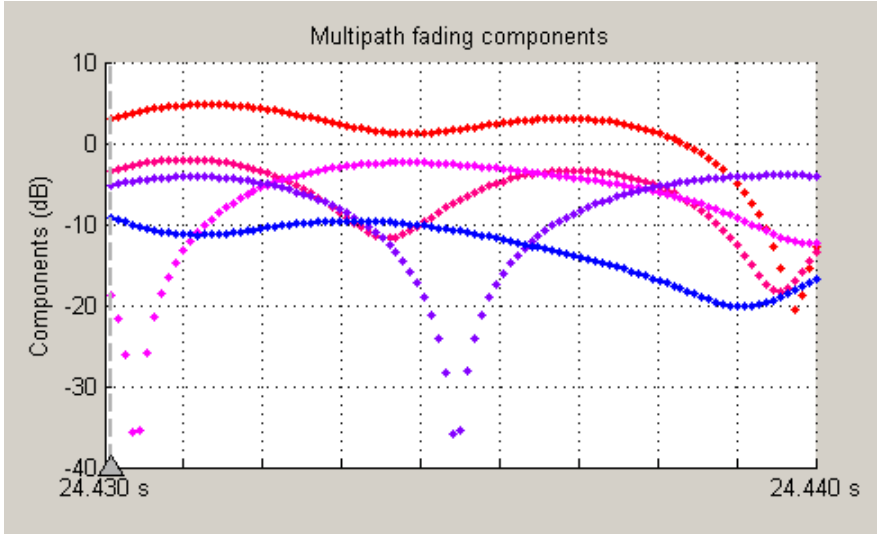


The phasors are connected end to end in order of path delay, and the trajectory of the resultant phasor is plotted as a green line. This resultant phasor is referred to as the *narrowband phasor*.

This plot can be used to determine the impact of the multipath channel on a narrowband signal. A narrowband signal is defined here as having a sample period much greater than the span of delays of the multipath channel (alternatively, a signal bandwidth much smaller than the coherence bandwidth of the channel). Thus, the multipath channel can be represented by a single complex gain, which is the sum of all the multipath component gains. When the narrowband phasor trajectory passes through or near the origin, it corresponds to a deep narrowband fade.

Multipath Components

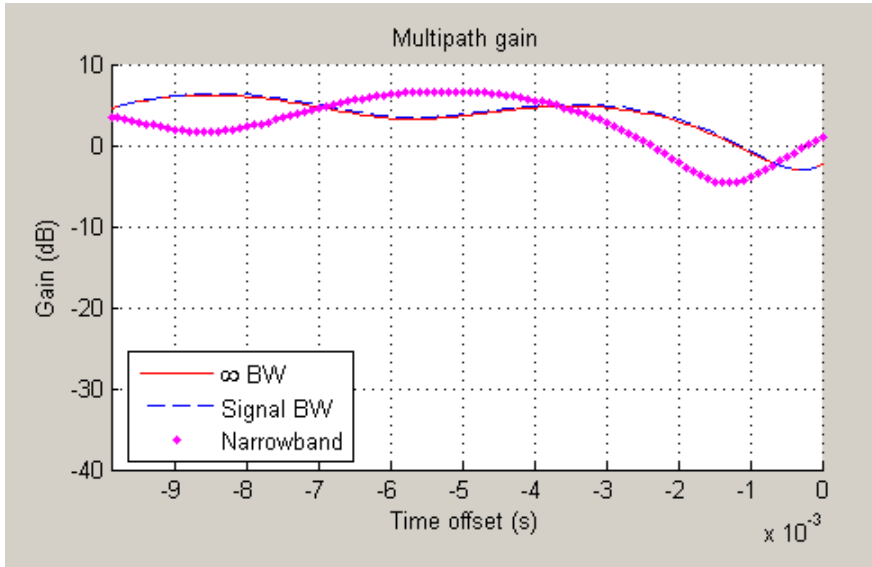
This plot shows the magnitudes of the multipath gains over time, using the same color code as that used for the multipath impulse response.



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

Multipath Gain

This plot shows the collective gains for the multipath channel for three signal bandwidths.



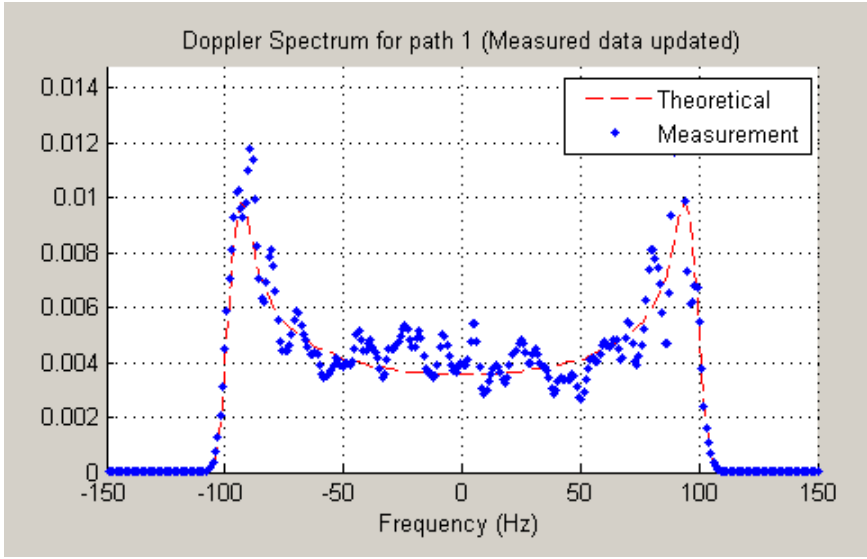
A collective gain is the sum of component magnitudes, as explained in the following:

- Narrowband (magenta dots): This is the magnitude of the narrowband phasor in the above trajectory plot. This curve is sometimes referred to as the *narrowband fading envelope*.
- Current signal bandwidth (dashed blue line): This is the sum of the magnitudes of the channel filter impulse response samples (the solid green dots in the impulse response plot). This curve represents the maximum signal energy that can be captured using a RAKE receiver. Its value (or metrics, such as theoretical BER, derived from it) is sometimes referred to as the *matched filter bound*.
- Infinite bandwidth (solid red line): This is the sum of the magnitudes of the multipath component gains.

In general, the variability of this multipath gain, or of the signal fading, decreases as signal bandwidth is increased, because multipath components become more resolvable. If the signal bandwidth curve roughly follows the narrowband curve, you might describe the signal as narrowband. If the signal bandwidth curve roughly follows the infinite bandwidth curve, you might describe the signal as wideband. With the right receiver, a wideband signal exploits the path diversity inherent in a multipath channel.

Doppler Spectrum

This plot shows up to two Doppler spectra.



The first Doppler spectrum, represented by the dashed red line, is a theoretical spectrum based on the Doppler filter response used in the multipath channel model. In the preceding plot, the theoretical Doppler spectrum used for the multipath channel model is known as the *Jakes spectrum*. Note that the plotted Doppler spectrum is normalized to have a total power of 1. This Doppler spectrum is used to determine a Doppler filter response. For practical purposes, the Doppler filter response is truncated, which has the effect of modifying the Doppler spectrum, as shown in the plot.

The second Doppler spectrum, represented by the blue dots, is determined by measuring the power spectrum of the multipath fading channel as the model generates path gains. This measurement is meaningful only after enough path gains have been generated. The title above the plot reports how many samples need to be processed through the channel before either the first Doppler spectrum or an updated spectrum can be plotted.

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

new Doppler spectrum will appear as soon as the processing of the current frame has ended.

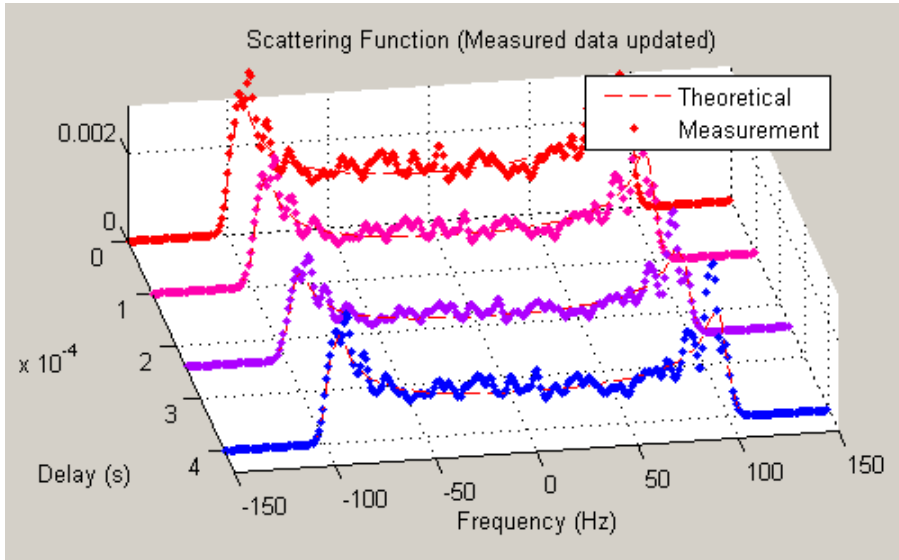
If the measured Doppler spectrum is a good approximation of the theoretical Doppler spectrum, the multipath channel model has generated enough fading gains to yield a reasonable representation of the channel statistics. For instance, if you want to determine the average BER of a communications link with a multipath channel and you want a statistically accurate measure of this average, you may want to ensure that the channel has processed enough samples to yield at least one Doppler spectrum measurement.

It is possible that a multipath channel (e.g., a Rician channel) can have both specular (line-of-sight) and diffuse components. In such a case, the Doppler spectrum would have both a line component and a wideband component. The channel visualization tool only shows the wideband component for the Doppler spectrum.

Unlike other visualizations, the Doppler spectrum visualization does not support animation. Because there is no intraframe data to plot, the visualization tool only updates the channel statistics at the end of each frame and therefore cannot pause in the middle of a frame. If you switch to the Doppler spectrum visualization from a different visualization that is in pause mode, the **Pause** button is subsequently disabled. Disabling pause avoids interaction problems between the Doppler spectrum visualization and other animation-style visualizations.

Scattering Function

This plot shows the Doppler spectra of each path versus the path delays, using the same color code as that used for the multipath impulse response.



The principle of operation of the Scattering Function plot is similar to that of the Doppler Spectrum plot. The main difference is that the Doppler spectra on this plot are not normalized as they are on the Doppler Spectrum plot, in order to better visualize the power delay profile.

Composite Plots

Several composite plots are also available. These are chosen by selecting the following from the **Visualization** pull-down menu:

- `IR and FR` for impulse response and frequency response plots.
- `Components and Gain` for multipath components and multipath gain plots.
- `Components and IR` for multipath components and impulse response plots.
- `Components, IR, and Phasor` for multipath components, impulse response, and phasor trajectory plots.

Examples of Using the Channel Visualization Tool

Here are two examples that show how you might interact with the GUI.

Visualize Samples Within a Frame

This example shows how to visualize samples within a frame through animation. The following lines of code create a Rayleigh channel and open the channel visualization tool:

```
% Create a fast fading channel
h = rayleighchan(1e-4, 100, [0 1.1e-4], [0 0]);

h.StoreHistory = 1;          % Allow states to be stored
y = filter(h, ones(100,1));  % Process samples through channel
plot(h);                     % Open channel visualization tool
```

After selecting a visualization option and a speed in the **Animation** menu, move the **Sample index** slider control all the way to the left and click **Resume**. The slider control moves by itself during animation. The sample index increments automatically to show which snapshot you are visualizing.

You can also move the slider control and glance through the samples of the frame as you like.

Animate Snapshots Across Frames

This example shows how to animate snapshots across frames. The following lines of code call the filter and plot methods within a loop to accomplish this:

```
Ts = 1e-4;   % Sample period (s)
fd = 100;    % Maximum Doppler shift

% Initialize DPSK modulator for M=4
hMod = comm.DPSKModulator(4);

% Path delay and gains
tau = [0.1 1.2 2.3 6.2 11.3]*Ts;
PdB = linspace(0, -10, length(tau)) - length(tau)/20;

nTrials = 10000;   % Number of trials
N = 100;           % Number of samples per frame

h = rayleighchan(Ts, fd, tau, PdB);  % Create channel object
h.NormalizePathGains = false;
h.ResetBeforeFiltering = false;
h.StoreHistory = 1;
h % Show channel object

% Channel fading simulation
for trial = 1:nTrials
    x = randi([0 3],10000,1);   % Random symbols
    dpskSig = step(hMod, x);    % Modulated symbols
    y = filter(h, dpskSig);     % Channel filter
    plot(h);                    % Plot channel response
```

```
    % The line below returns control to the command line in case
    % the GUI is closed while this program is still running
    if isempty(findobj('name', 'Multipath Channel')), break; end;
end
```

While the animation is running, you can move the slider control and change the sample index (which also makes the animation pause). After clicking **Resume**, the plot continues to animate.

The property `ResetBeforeFiltering` needs to be set to false so that the state information in the channel is not reset after the processing of each frame.

Rician Fading Channel

Quasi-Static Channel Modeling

Typically, a path gain in a fading channel changes insignificantly over a period of $1/(100f_d)$ seconds, where f_d is the maximum Doppler shift. Because this period corresponds to a very large number of bits in many modern wireless data applications, assessing performance over a statistically significant range of fading entails simulating a prohibitively large amount of data. Quasi-static channel modeling provides a more tractable approach, which you can implement using these steps:

1. Generate a random channel realization using a maximum Doppler shift of 0.
2. Process some large number of bits.
3. Compute error statistics.
4. Repeat these steps many times to produce a distribution of the performance metric.

The example below illustrates the quasi-static channel modeling approach.

```
M = 4;                          % DQPSK modulation order
hMod = comm.DQPSKModulator;       % Create a DPSK modulator
hDemod = comm.DQPSKDemodulator;     % Create a DPSK demodulator

numBits = 10000;                % Each trial uses 10000 bits.
numTrials = 20;                 % Number of BER computations

% Note: In reality, numTrials would be a large number
% to get an accurate estimate of outage probabilities
% or packet error rate.
% Use 20 here just to make the example run more quickly.
```

```
% Create Rician channel object.
chan = ricianchan;      % Static Rician channel
chan.KFactor = 3;       % Rician K-factor
% Because chan.ResetBeforeFiltering is 1 by default,
% FILTER resets the channel in each trial below.

% Create an AWGNChannel and ErrorRate calculator System object
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');
hChan.SNR = 20;
hErrorCalc = comm.ErrorRate;
serVec = zeros(3,numTrials);

% Compute error rate once for each independent trial.
for n = 1:numTrials
    reset(hErrorCalc)
   tx = randi([0 M-1],numBits,1);       % Generate random bit stream
   dpskSig = step(hMod, tx);        % DPSK modulate signal
   fadedSig = filter(chan, dpskSig);    % Apply channel effects
   rxSig = step(hChan,fadedSig);           % Add Gaussian noise.
   rx = step(hDemod,rxSig);       % Demodulate.

   % Compute number of symbol errors.
   % Ignore first sample because of DPSK initial condition.
   serVec(:,n) = step(hErrorCalc,tx(2:end),rx(2:end));
end
nErrors = serVec(2,:)
per = mean(nErrors > 0) % Proportion of packets that had errors
```

While the example runs, the Command Window displays the growing list of symbol error counts in the vector `nErrors`. It also displays the packet error rate at the end. The sample output below shows a final value of `nErrors` and omits intermediate values. Your results might vary because of randomness in the example.

```
nErrors =

  Columns 1 through 9

     0     0     0     0     0     0     0     0     0

  Columns 10 through 18

     0     0     0     0     7     0     0     0     0
```

```
  Columns 19 through 20

     0   216


per =

    0.1000
```

More About the Quasi-Static Technique

As an example to show how the quasi-static channel modeling approach can save computation, consider a wireless local area network (LAN) in which the carrier frequency is 2.4 GHz, mobile speed is 1 m/s, and bit rate is 10 Mb/s. The following expression shows that the channel changes insignificantly over 12,500 bits:

$$\begin{aligned}\left(\frac{1}{100 f_d}\ \text{s}\right)(10\ \text{Mb/s}) &= \left(\frac{c}{100 v f}\ \text{s}\right)(10\ \text{Mb/s}) \\ &= \frac{3\times 10^8\ \text{m/s}}{100(1\ \text{m/s})(2.4\ \text{GHz})}(10\ \text{Mb/s}) \\ &= 12{,}500\ \text{b}\end{aligned}$$

A traditional Monte Carlo approach for computing the error rate of this system would entail simulating thousands of times the number of bits shown above, perhaps tens of millions of bits. By contrast, a quasi-static channel modeling approach would simulate a few packets at each of about 100 locations to arrive at a spatial distribution of error rates. From this distribution one could determine, for example, how reliable the communication link is for a random location within the indoor space. If each simulation contains 5,000 bits, 100 simulations would process half a million bits in total. This is substantially fewer bits compared to the traditional Monte Carlo approach.

Additional Examples Using Fading Channels

The following models include the use of fading channels:

- Rayleigh Fading Channel, which illustrates the channel's effect on a QPSK modulated signal
- IEEE 802.11a WLAN Physical Layer
- Defense Communications: US MIL-STD-188-110B

- WCDMA End-to-End Physical Layer

MIMO Channel

The Communications System Toolbox software provides a Multiple Input Multiple Output (MIMO) Multipath Fading Channel System object. Multipath MIMO fading channels allow for design of communication systems with multiple antenna elements at the transmitter and receiver.

For more information, see the `comm.MIMOChannel` Help page.

The product also includes an LTE MIMO Multipath Fading Channel System object. This object allows for design of communication systems with multiple antenna elements at the transmitter and receiver using the 3GPP Long Term Evolution (LTE) standard.

For more information, see the `comm.LTEMIMOChannel` Help page.

The following demos illustrate MIMO fading channel techniques using MATLAB System objects:

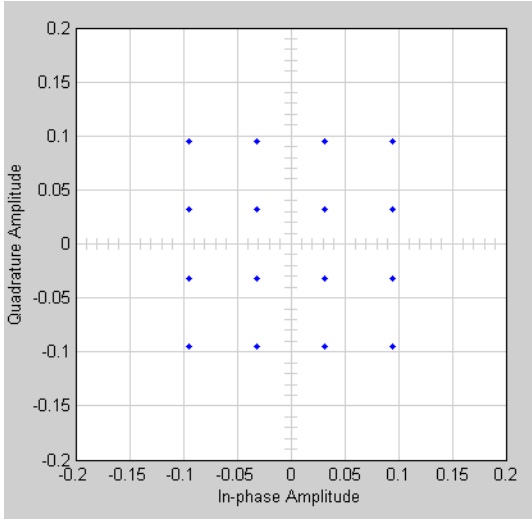
- Introduction to MIMO Systems
- IEEE 802.11n Channel Models
- IEEE 802.16 Channel Models
- LTE PHY Downlink with Spatial Multiplexing

RF Impairments

In this section...

Illustrate RF Impairments That Distort a Signal

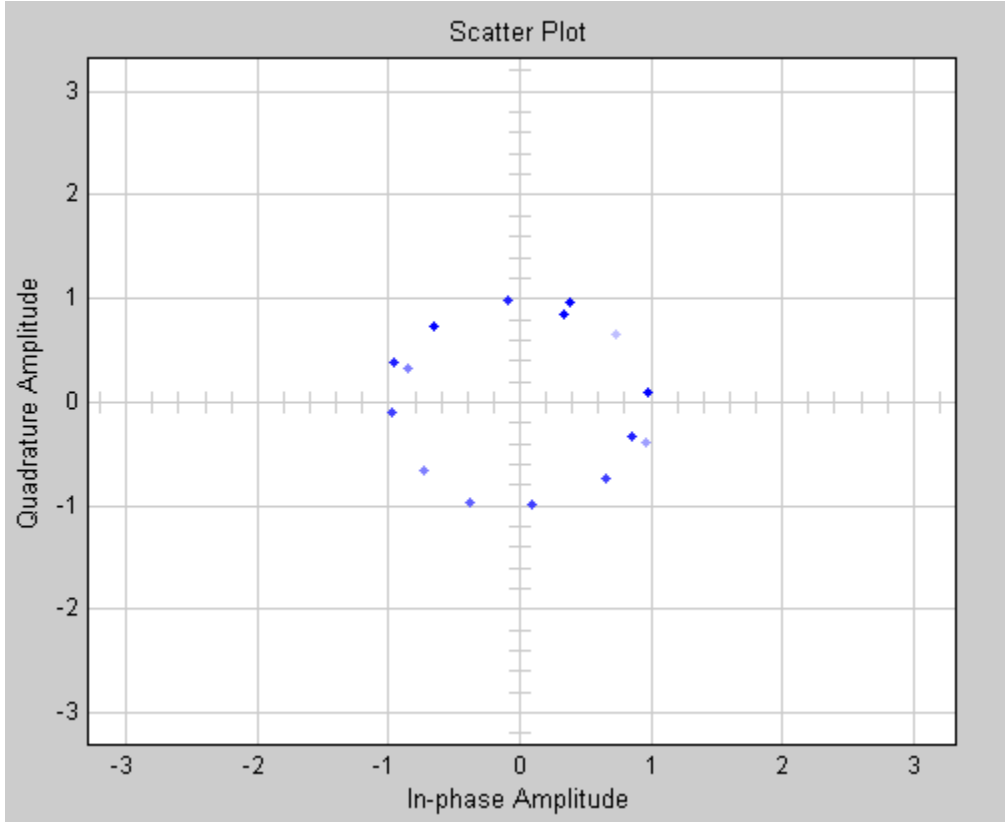
This section presents scatter plots that illustrate how blocks in the RF Impairments library distort a signal modulated by 16-ary quadrature amplitude modulation (QAM). The usual 16-ary QAM constellation without distortion is shown in the following figure.



As the scatter plots show, the first two blocks distort both the magnitude and angle of points in the constellation, while the last two alter just the angle.

You can create these scatter plots with models similar to the following, which produces the scatter plot for the Memoryless Nonlinearity block:

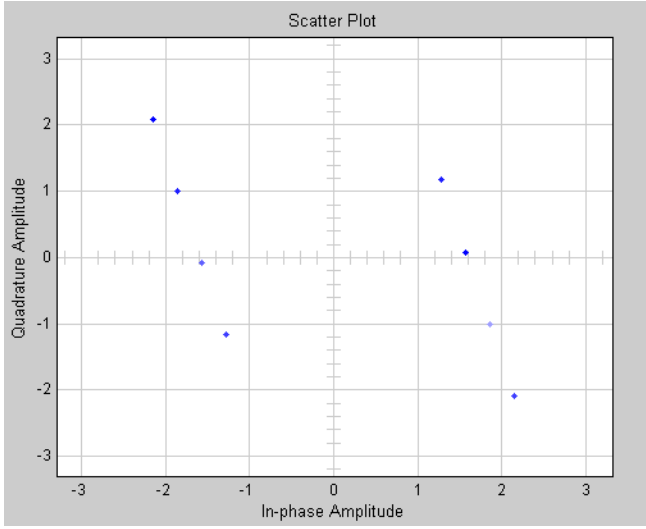
16-ary QAM Model



The model uses the Rectangular QAM Modulator Baseband block, from AM in the Digital Baseband Modulation sublibrary of the Modulation library. You control the power of the block's output signal with the **Normalization method** parameter. To open this model, enter `doc_16qam_plot` at the MATLAB command line.

I/Q Imbalance Block

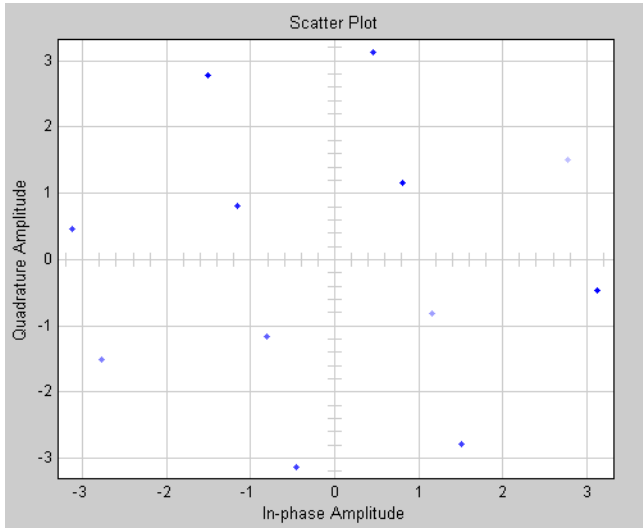
You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model with the I/Q Imbalance block. Set the block's **I/Q amplitude imbalance (dB)** parameter to 10 and the **I/Q phase imbalance (deg)** parameter to 30.



For more examples of scatter plots produced using this block, see the reference page for the I/Q Imbalance block.

Phase/Frequency Offset Block

You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model with the Phase/Frequency Offset block. Set the block's **Frequency offset (Hz)** parameter to 0 and the **Phase offset (deg)** parameter to 70.

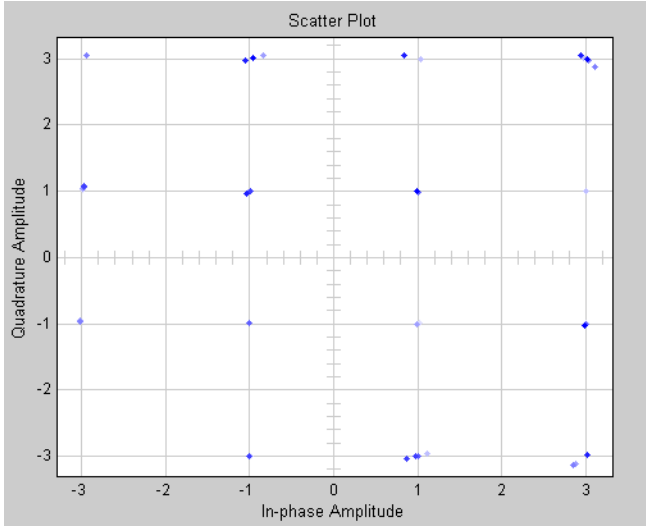


The **Frequency offset (Hz)** parameter adds a constant to the phase of the signal. The scatter plot corresponds to the standard constellation rotated by a fixed angle of 70 degrees.

The **Frequency offset (Hz)** parameter determines the rate of change of the signal's phase. In this example, **Frequency offset (Hz)** is set to `0`, so the scatter plot always falls on the grid shown in the preceding figure. If you set **Frequency offset (Hz)** to a positive number, the points on the scatter plot fall on a rotating grid, corresponding to the standard constellation, which revolves at a constant rate in the counterclockwise direction. For an example, see the reference page for the Phase/Frequency Offset block.

Phase Noise Block

You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model with the Phase Noise block. Set the **Phase noise level (dBc/Hz)** parameter to `-60` and the **Frequency offset (Hz)** parameter to `100`.



The phase noise adds a random error to the signal's phase, so that the points in the scatter plot are spread in a radial pattern around the constellation points.

Phase/Frequency Offsets and Phase Noise

The RF Impairments library contains two blocks that simulate phase/frequency offsets and phase noise:

- The Phase/Frequency Offset block applies phase and frequency offsets to a signal.
- The Phase Noise block applies phase noise to a signal.

The Phase/Frequency Offset block and the Phase Noise block alter only the phase and frequency of the signal.

Receiver Thermal Noise and Free Space Path Loss

The RF Impairments Library contains two blocks that simulate signal impairments due to thermal noise and signal attenuation due to the distance from the transmitter to the receiver:

- The Receiver Thermal Noise block simulates the effects of thermal noise on a complex baseband signal.

- The Free Space Path Loss block simulates the loss of signal power due to the distance from the transmitter and signal frequency.

Nonlinearity and I/Q Imbalance

The following two blocks model signal impairments due to nonlinear devices or imbalances between the in-phase and quadrature components of a modulated signal:

- The Memoryless Nonlinearity block models the AM-to-AM and AM-to-PM distortion in nonlinear amplifiers.
- The I/Q Imbalance block models imbalances between the in-phase and quadrature components of a signal caused by differences in the physical channels carrying the separate components.

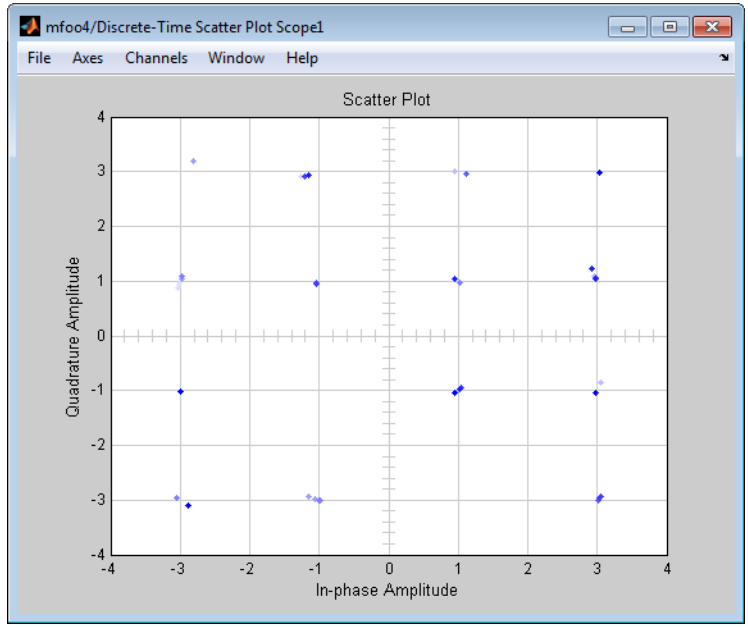
These blocks distort both the phase and amplitude of the signal.

Apply Nonlinear Distortion to Input Signal

The Memoryless Nonlinearity block applies a nonlinear distortion to the input signal. This distortion models the AM-to-AM and AM-to-PM conversions in nonlinear amplifiers. The block provides several methods, which you specify by the **Method** parameter, for modeling the nonlinear characteristics of amplifiers:

- Cubic polynomial
- Hyperbolic tangent
- Saleh model
- Ghorbani model
- Rapp model

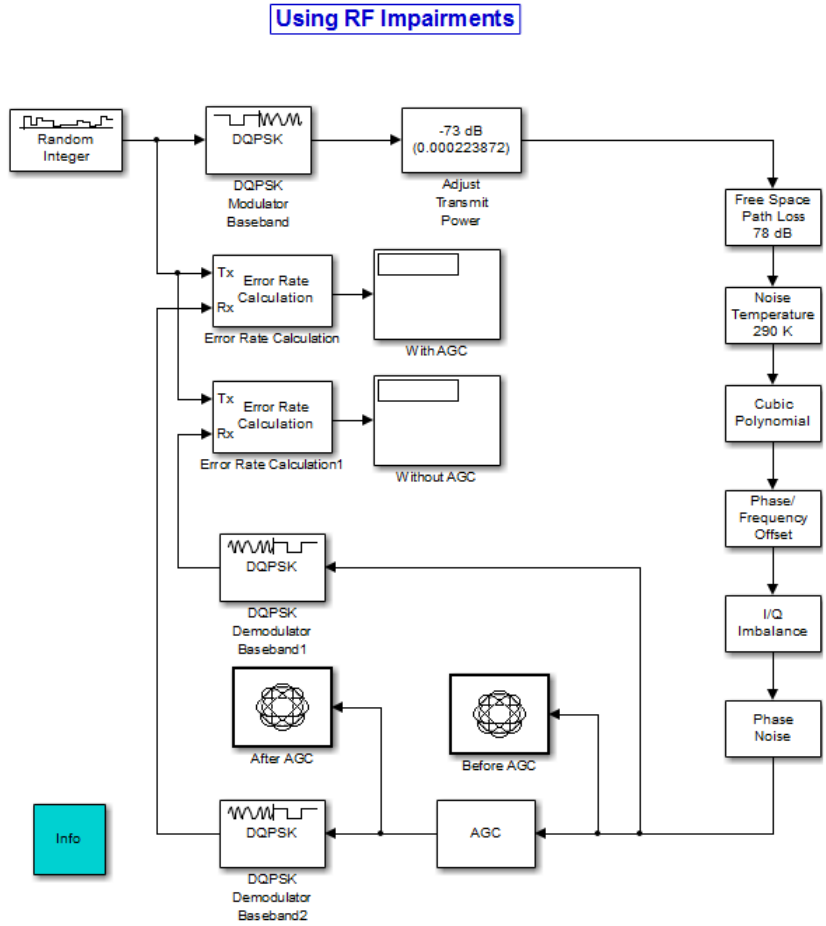
In the model shown in the preceding figure, the **Method** parameter is set to `Ghorbani model`. The following figure shows the scatter plot the model generates.



For another example of a scatter plot produced using this block, see the reference page for the Memoryless Nonlinearity block.

Simulate RF Impairments to a DQPSK Signal

The model shown in the following figure simulates RF impairments to a signal modulated by differential quaternary phase shift keying (DQPSK).



You can open this model by typing `doc_receiverimpairments_dqpsk` at the MATLAB command line.

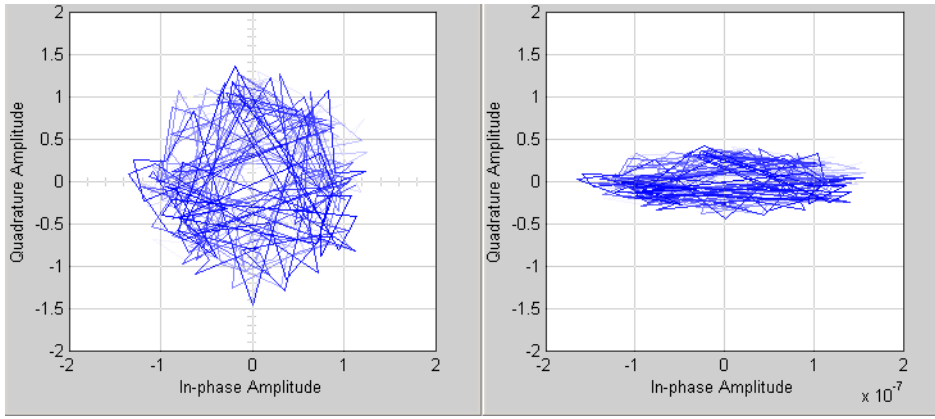
Overview of the Model

The model does the following:

- Modulates a random signal using DQPSK modulation.
- Applies impairments to the signal using the blocks from the RF Impairments library.

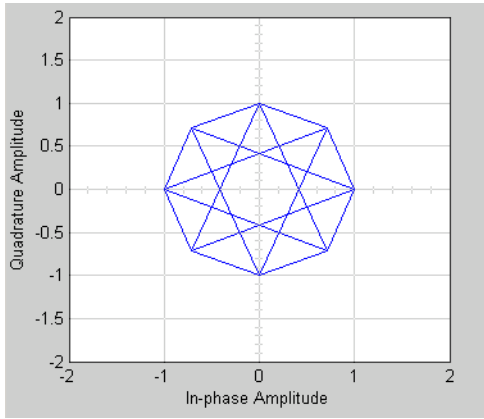
- Forks the signal into two paths, and processes one path with an automatic gain control (AGC) to compensate for the free space path loss and the I/Q imbalance.
- Displays the trajectory of the signal with AGC and the trajectory of the signal without AGC.
- Demodulates both signals and calculates their error rates.

You can see the effect of the automatic gain by comparing the trajectories of the signals with and without AGC, as shown in the following figure.



Signal With (Left) and Without (Right) AGC

The trajectory of the signal with AGC more closely matches the undistorted trajectory for DQPSK, shown in the following figure, than does than the signal without AGC. Consequently, the error rate for the signal with AGC is much lower than the error rate for the signal without AGC.



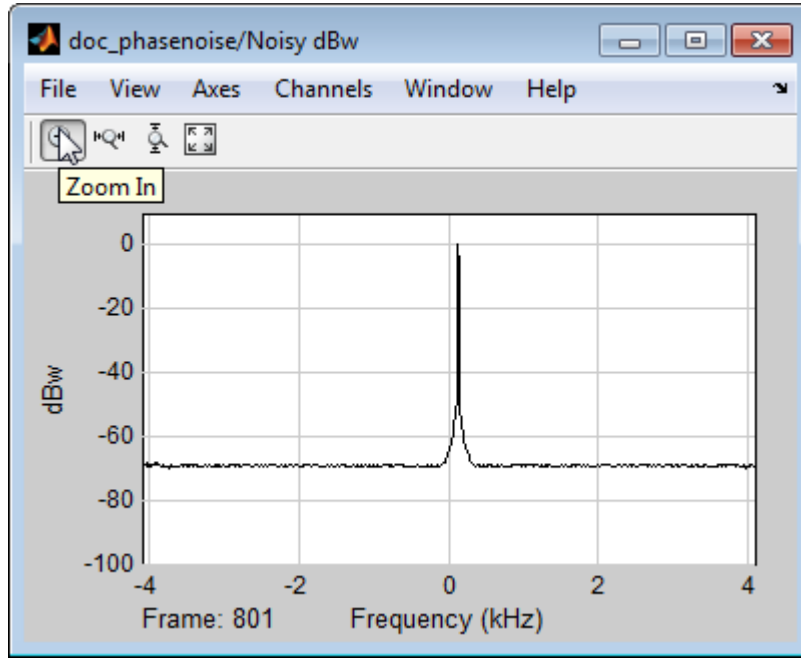
In this example, the error rate for the demodulated signal without AGC is primarily caused by free space path loss and I/Q imbalance. The QPSK modulation minimizes the effects of the other impairments.

View Phase Noise Effects on Signal Spectrum

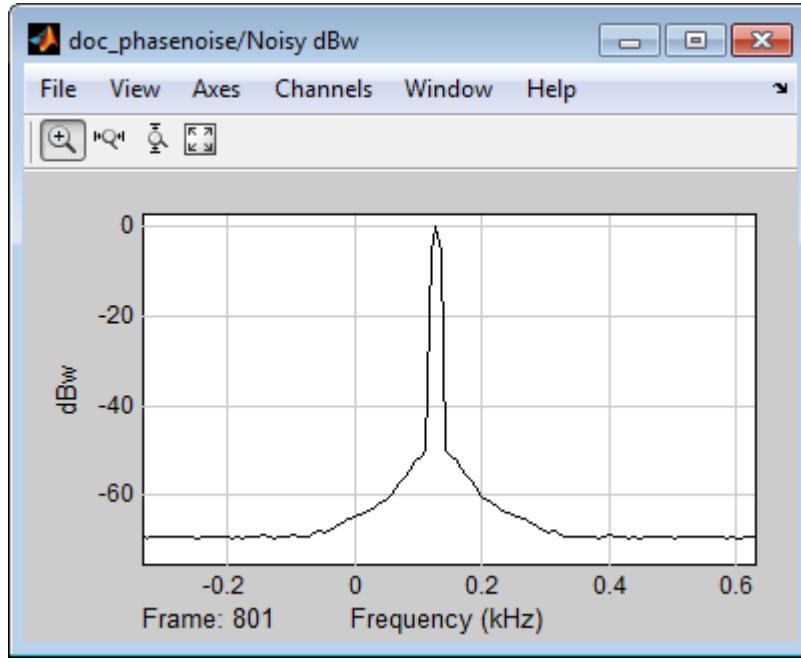
This example shows the effects spectral and phase noise have on a 128 Hz carrier frequency.

1. Type `doc_phasenoise` at the MATLAB command line to open the model.
2. Click **Simulation** > **Run**.

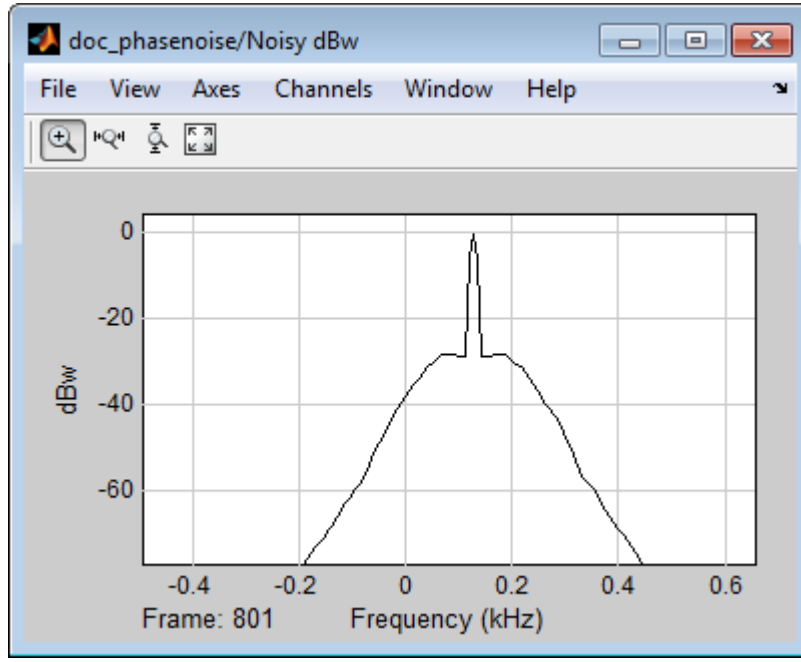
   The model generates four figure windows. Notice the position of the 128 Hertz signal, and the respective noise floor on the different plots. Take note of the numeric value that the RMS Phase Noise block displays.
3. In the Noisy dBw figure window, click **Zoom In**.



4 Move the mouse pointer to the figure window and then click-and-drag to zoom in on the 128 Hz signal.



5 In the Simulink model, double-click the Phase Noise block mask.
6 Change the value of the **Phase noise level** block parameter to `[-40 -100]`
7 Change the value of the **Frequency offset** block parameter to `[100 400]`
8 Click **OK**.
9 Click **Run**.
10 Observe how changing the phase noise and frequency offset vectors effects the 128 Hz signal.



As you add noise, the spectrum shape changes. With more noise, 128 Hz signal becomes less distinct, as the side lobes increase in amplitude. Similarly, as you add phase noise, the measured value in the RMS Phase Noise block also increases.

Selected Bibliography for Channel Modeling

[1] Simon, M. K., and Alouini, M. S.,
Digital Communication over Fading Channels – A Unified Approach to Performance Analysis, 1st Ed., Wiley, 2000.

[2] 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.

[3] 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.

Measurements

Bit Error Rate (BER)

In this section...

Theoretical Results

Common Notation

The following notation is used throughout this Appendix:

Quantity or Operation	Notation
Size of modulation constellation	M
Number of bits per symbol	$k = \log_2 M$
Energy per bit-to-noise power-spectral-density ratio	$\frac{E_b}{N_0}$
Energy per symbol-to-noise power-spectral-density ratio	$\frac{E_s}{N_0} = k\frac{E_b}{N_0}$
Bit error rate (BER)	P_b
Symbol error rate (SER)	P_s
Real part	$\mathrm{Re}[\cdot]$
Largest integer smaller than	$\lfloor \cdot \rfloor$

The following mathematical functions are used:

Function	Mathematical Expression
Q function	$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-t^2/2)dt$$
Marcum Q function	$$Q(a,b) = \int_b^\infty t \exp\left(-\frac{t^2+a^2}{2}\right) I_0(at)dt$$
Modified Bessel function of the first kind of order ν	$$I_\nu(z) = \sum_{k=0}^{\infty} \frac{(z/2)^{\upsilon+2k}}{k!\Gamma(\nu+k+1)}$$ where $$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$ is the gamma function.
Confluent hypergeometric function	$$_1F_1(a,c;x) = \sum_{k=0}^{\infty} \frac{(a)_k}{(c)_k} \frac{x^k}{k!}$$ where the Pochhammer symbol, $(\lambda)_k$, is defined as $(\lambda)_0 = 1$, $(\lambda)_k = \lambda(\lambda+1)(\lambda+2)\cdots(\lambda+k-1)$.

The following acronyms are used:

Acronym	Definition
M-PSK	*M*-ary phase-shift keying
DE-M-PSK	Differentially encoded *M*-ary phase-shift keying

Acronym	Definition
BPSK	Binary phase-shift keying
DE-BPSK	Differentially encoded binary phase-shift keying
QPSK	Quaternary phase-shift keying
DE-QPSK	Differentially encoded quaternary phase-shift keying
OQPSK	Offset quaternary phase-shift keying
DE-OQPSK	Differentially encoded offset quaternary phase-shift keying
M-DPSK	*M*-ary differential phase-shift keying
M-PAM	*M*-ary pulse amplitude modulation
M-QAM	*M*-ary quadrature amplitude modulation
M-FSK	*M*-ary frequency-shift keying
MSK	Minimum shift keying
M-CPFSK	*M*-ary continuous-phase frequency-shift keying

Analytical Expressions Used in berawgn

M-PSK

From equation 8.22 in [2]

$$P_s = \frac{1}{\pi}\int_0^{(M-1)\pi/M} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[\pi/M]}{\sin^2\theta}\right)d\theta$$

The following expression is very close, but not strictly equal, to the exact BER (from [4] and equation 8.29 from [2]):

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')P_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$P_i = \frac{1}{2\pi}\int_0^{\pi(1-(2i-1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[(2i-1)\pi/M]}{\sin^2\theta}\right)d\theta - \frac{1}{2\pi}\int_0^{\pi(1-(2i+1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[(2i+1)\pi/M]}{\sin^2\theta}\right)d\theta$$

Special case of $M = 2$, e.g., BPSK (equation 5.2-57 from [1]):

$$P_s = P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

Special case of $M = 4$, e.g., QPSK (equations 5.2-59 and 5.2-62 from [1]):

$$P_s = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1-\frac{1}{2}Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

DE-M-PSK

$M = 2$, e.g., DE-BPSK (equation 8.36 from [2]):

$$P_s = P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)-2Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

$M = 4$, e.g., DE-QPSK (equation 8.38 from [2]):

$$P_s = 4Q\left(\sqrt{\frac{2E_b}{N_0}}\right)-8Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right)+8Q^3\left(\sqrt{\frac{2E_b}{N_0}}\right)-4Q^4\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

From equation 5 in [3]:

$$P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1-Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$

OQPSK

Same BER/SER as QPSK [2].

DE-OQPSK

Same BER/SER as DE-QPSK [3].

M-DPSK

From equation 8.84 in [2]:

$$P_s = \frac{\sin(\pi / M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{\exp\left(-(kE_b / N_0)(1-\cos(\pi / M)\cos\theta)\right)}{1-\cos(\pi / M)\cos\theta}d\theta$$

The following expression is very close, but not strictly equal, to the exact BER [4]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2} (w_i^{'}) A_i \right)$$

where $w_i^{'} = w_i + w_{M-i}$, $w_{M/2}^{'} = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$A_i = F\left((2i+1)\frac{\pi}{M} \right) - F\left((2i-1)\frac{\pi}{M} \right)$$

$$F(\psi) = -\frac{\sin\psi}{4\pi} \int_{-\pi/2}^{\pi/2} \frac{\exp\left(-kE_b / N_0 (1-\cos\psi\cos t)\right)}{1-\cos\psi\cos t} dt$$

Special case of $M = 2$ (equation 8.85 from [2]):

$$P_b = \frac{1}{2}\exp\left(-\frac{E_b}{N_0} \right)$$

M-PAM

From equations 8.3 and 8.7 in [2], and equation 5.2-46 in [1]:

$$P_s = 2\left(\frac{M-1}{M} \right) Q\left(\sqrt{\frac{6}{M^2-1}\frac{kE_b}{N_0}} \right)$$

From [5]:

$$P_b = \frac{2}{M\log_2 M} \times \sum_{k=1}^{\log_2 M} \sum_{i=0}^{(1-2^{-k})M-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{M} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{M} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1)\sqrt{\frac{6\log_2 M}{M^2-1}\frac{E_b}{N_0}} \right) \right\}$$

M-QAM

For square M-QAM, $k = \log_2 M$ is even (equation 8.10 from [2], and equations 5.2-78 and 5.2-79 from [1]):

$$P_s = 4\frac{\sqrt{M}-1}{\sqrt{M}}Q\left(\sqrt{\frac{3}{M-1}\frac{kE_b}{N_0}}\right) - 4\left(\frac{\sqrt{M}-1}{\sqrt{M}}\right)^2 Q^2\left(\sqrt{\frac{3}{M-1}\frac{kE_b}{N_0}}\right)$$

From [5]:

$$P_b = \frac{2}{\sqrt{M}\log_2\sqrt{M}} \times \sum_{k=1}^{\log_2\sqrt{M}} \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{\sqrt{M}} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1)\sqrt{\frac{6\log_2 M}{2(M-1)}\frac{E_b}{N_0}} \right) \right\}$$

For rectangular (non-square) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$, $I = 2^{\frac{k-1}{2}}$, and $J = 2^{\frac{k+1}{2}}$:

$$P_s = \frac{4IJ - 2I - 2J}{M} \times Q\left(\sqrt{\frac{6\log_2(IJ)}{(I^2+J^2-2)}\frac{E_b}{N_0}}\right) - \frac{4}{M}(1+IJ-I-J)Q^2\left(\sqrt{\frac{6\log_2(IJ)}{(I^2+J^2-2)}\frac{E_b}{N_0}}\right)$$

From [5]:

$$P_b = \frac{1}{\log_2(IJ)}\left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l) \right)$$

where

$$P_I(k) = \frac{2}{I} \sum_{i=0}^{(1-2^{-k})I-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{I} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{I} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1)\sqrt{\frac{6\log_2(IJ)}{I^2+J^2-2}\frac{E_b}{N_0}} \right) \right\}$$

and

$$P_J(k) = \frac{2}{J} \sum_{j=0}^{(1-2^{-l})J-1} \left\{ (-1)^{\left\lfloor \frac{j2^{l-1}}{J} \right\rfloor} \left(2^{l-1} - \left\lfloor \frac{j2^{l-1}}{J} + \frac{1}{2} \right\rfloor \right) Q\left((2j+1) \sqrt{\frac{6\log_2(IJ)}{I^2+J^2-2} \frac{E_b}{N_0}} \right) \right\}$$

Orthogonal M-FSK with Coherent Detection

From equation 8.40 in [2] and equation 5.2-21 in [1]:

$$P_s = 1 - \int_{-\infty}^{\infty} \left[Q\left(-q - \sqrt{\frac{2kE_b}{N_0}} \right) \right]^{M-1} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{q^2}{2} \right) dq$$

$$P_b = \frac{2^{k-1}}{2^k - 1} P_s$$

Nonorthogonal 2-FSK with Coherent Detection

For $M = 2$ (from equation 5.2-21 in [1] and equation 8.44 in [2]):

$$P_s = P_b = Q\left(\sqrt{\frac{E_b(1 - \mathrm{Re}[\rho])}{N_0}} \right)$$

ρ is the complex correlation coefficient:

$$\rho = \frac{1}{2E_b} \int_0^{T_b} \tilde{s}_1(t) \tilde{s}_2^*(t) dt$$

where $\tilde{s}_1(t)$ and $\tilde{s}_2(t)$ are complex lowpass signals, and

$$E_b = \frac{1}{2} \int_0^{T_b} \left| \tilde{s}_1(t) \right|^2 dt = \frac{1}{2} \int_0^{T_b} \left| \tilde{s}_2(t) \right|^2 dt$$

For example:

$$\tilde{s}_1(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_1 t}, \; \tilde{s}_2(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_2 t}$$

$$\rho = \frac{1}{2E_b}\int_0^{T_b}\sqrt{\frac{2E_b}{T_b}}e^{j2\pi f_1 t}\sqrt{\frac{2E_b}{T_b}}e^{-j2\pi f_2 t}dt = \frac{1}{T_b}\int_0^{T_b}e^{j2\pi(f_1-f_2)t}dt$$
$$= \frac{\sin(\pi\Delta f T_b)}{\pi\Delta f T_b}e^{j\pi\Delta f t}$$

where $\Delta f = f_1 - f_2$.

$$\mathrm{Re}[\rho] = \mathrm{Re}\left[\frac{\sin(\pi\Delta f T_b)}{\pi\Delta f T_b}e^{j\pi\Delta f t}\right] = \frac{\sin(\pi\Delta f T_b)}{\pi\Delta f T_b}\cos(\pi\Delta f T_b) = \frac{\sin(2\pi\Delta f T_b)}{2\pi\Delta f T_b}$$
$$\Rightarrow P_b = Q\left(\sqrt{\frac{E_b(1-\sin(2\pi\Delta f T_b)/(2\pi\Delta f T_b))}{N_0}}\right)$$

(from equation 8.44 in [2], where $h = \Delta f T_b$)

Orthogonal M-FSK with Noncoherent Detection

From equation 5.4-46 in [1] and equation 8.66 in [2]:

$$P_s = \sum_{m=1}^{M-1}(-1)^{m+1}\binom{M-1}{m}\frac{1}{m+1}\exp\left[-\frac{m}{m+1}\frac{kE_b}{N_0}\right]$$
$$P_b = \frac{1}{2}\frac{M}{M-1}P_s$$

Nonorthogonal 2-FSK with Noncoherent Detection

For $M = 2$ (from equation 5.4-53 in [1] and equation 8.69 in [2]):

$$P_s = P_b = Q(\sqrt{a},\sqrt{b}) - \frac{1}{2}\exp\left(-\frac{a+b}{2}\right)I_0(\sqrt{ab})$$

where

$$a = \frac{E_b}{2N_0}(1-\sqrt{1-|\rho|^2}),\ b = \frac{E_b}{2N_0}(1+\sqrt{1-|\rho|^2})$$

Precoded MSK with Coherent Detection

Same BER/SER as BPSK.

Differentially Encoded MSK with Coherent Detection

Same BER/SER as DE-BPSK.

MSK with Noncoherent Detection (Optimum Block-by-Block)

Upper bound (from equations 10.166 and 10.164 in [6]):

$$P_s = P_b$$

$$\le \frac{1}{2}\left[1 - Q\left(\sqrt{b_1}, \sqrt{a_1}\right) + Q\left(\sqrt{a_1}, \sqrt{b_1}\right)\right] + \frac{1}{4}\left[1 - Q\left(\sqrt{b_4}, \sqrt{a_4}\right) + Q\left(\sqrt{a_4}, \sqrt{b_4}\right)\right] + \frac{1}{2}e^{-\frac{E_b}{N_0}}$$

where

$$a_1 = \frac{E_b}{N_0}\left(1 - \sqrt{\frac{3 - 4/\pi^2}{4}}\right), \quad b_1 = \frac{E_b}{N_0}\left(1 + \sqrt{\frac{3 - 4/\pi^2}{4}}\right)$$

$$a_4 = \frac{E_b}{N_0}\left(1 - \sqrt{1 - 4/\pi^2}\right), \quad b_4 = \frac{E_b}{N_0}\left(1 + \sqrt{1 - 4/\pi^2}\right)$$

CPFSK Coherent Detection (Optimum Block-by-Block)

Lower bound (from equation 5.3-17 in [1]):

$$P_s > K_{\delta_{\min}} Q\left(\sqrt{\frac{E_b}{N_0}\delta_{\min}^2}\right)$$

Upper bound:

$$\delta_{\min}^2 > \min_{1 \le i \le M-1}\left\{2i\left(1 - \operatorname{sinc}(2ih)\right)\right\}$$

where h is the modulation index, and $K_{\delta_{\min}}$ is the number of paths having the minimum distance.

$$P_b \cong \frac{P_s}{k}$$

Analytical Expressions Used in berfading

Notation

The following notation is used for the expressions found in `berfading`.

Value	Notation
Power of the fading amplitude r	$\Omega = E\left[r^2\right]$, where $E[\cdot]$ denotes statistical expectation
Number of diversity branches	L
SNR per symbol per branch	$\bar{\gamma}_l = \left(\Omega_l \frac{E_s}{N_0}\right) / L = \left(\Omega_l \frac{kE_b}{N_0}\right) / L$ For identically-distributed diversity branches: $\bar{\gamma} = \left(\Omega \frac{kE_b}{N_0}\right) / L$
Moment generating functions for each diversity branch	Rayleigh fading: $M_{\gamma_l}(s) = \frac{1}{1 - s\bar{\gamma}_l}$

Value	Notation
	Rician fading: $M_{\gamma_l}(s) = \frac{1+K}{1+K-s\bar{\gamma}_l} e^{\left[\frac{Ks\bar{\gamma}_l}{(1+K)-s\bar{\gamma}_l}\right]}$ where K is the ratio of energy in the specular component to the energy in the diffuse component (linear scale). For identically-distributed diversity branches: $M_{\gamma_l}(s) = M_{\gamma}(s)$ for all l.

The following acronyms are used:

Acronym	Definition
MRC	maximal-ratio combining
EGC	equal-gain combining

M-PSK with MRC

From equation 9.15 in [2]:

$$P_s = \frac{1}{\pi} \int_0^{(M-1)\pi/M} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{\sin^2(\pi/M)}{\sin^2\theta}\right) d\theta$$

From [4] and [2]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2} (w_i^{'})\bar{P}_i\right)$$

where $w_i^{'} = w_i + w_{M-i}$, $w_{M/2}^{'} = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$\bar{P}_i = \frac{1}{2\pi}\int_0^{\pi(1-(2i-1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i-1)\pi}{M}\right)d\theta - \frac{1}{2\pi}\int_0^{\pi(1-(2i+1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i+1)\pi}{M}\right)d\theta$$

For the special case of Rayleigh fading with $M = 2$ (from equations C-18, C-21, and Table C-1 in [6]):

$$P_b = \frac{1}{2}\left[1 - \mu\sum_{i=0}^{L-1}\binom{2i}{i}\left(\frac{1-\mu^2}{4}\right)^i\right]$$

where

$$\mu = \sqrt{\frac{\bar{\gamma}}{\bar{\gamma}+1}}$$

If $L = 1$:

$$P_b = \frac{1}{2}\left[1 - \sqrt{\frac{\bar{\gamma}}{\bar{\gamma}+1}}\right]$$

DE-M-PSK with MRC

For $M = 2$ (from equations 8.37 and 9.8-9.11 in [2]):

$$P_s = P_b = \frac{2}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta - \frac{2}{\pi}\int_0^{\pi/4}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta$$

M-PAM with MRC

From equation 9.19 in [2]:

$$P_s = \frac{2(M-1)}{M\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(M^2-1)}{\sin^2\theta}\right)d\theta$$

From [5] and [2]:

$$P_b = \frac{2}{\pi M \log_2 M} \times \sum_{k=1}^{\log_2 M} \sum_{i=0}^{(1-2^{-k})M-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{M} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{M} + \frac{1}{2} \right\rfloor \right) \int_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{(2i+1)^2 3/(M^2-1)}{\sin^2\theta} \right) d\theta \right\}$$

M-QAM with MRC

For square M-QAM, $k = \log_2 M$ is even (equation 9.21 in [2]):

$$P_s = \frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right) \int_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{3/(2(M-1))}{\sin^2\theta} \right) d\theta - \frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right)^2 \int_0^{\pi/4} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{3/(2(M-1))}{\sin^2\theta} \right) d\theta$$

From [5] and [2]:

$$P_b = \frac{2}{\pi\sqrt{M} \log_2 \sqrt{M}} \times \sum_{k=1}^{\log_2 \sqrt{M}} \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{\sqrt{M}} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) \int_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{(2i+1)^2 3/(2(M-1))}{\sin^2\theta} \right) d\theta \right\}$$

For rectangular (nonsquare) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$, $I = 2^{\frac{k-1}{2}}$, $J = 2^{\frac{k+1}{2}}$, $\overline{\gamma}_l = \Omega_l \log_2(IJ) \frac{E_b}{N_0}$, and

$$P_s = \frac{4IJ - 2I - 2J}{M\pi} \int_0^{\pi/2} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{3/(I^2+J^2-2)}{\sin^2\theta} \right) d\theta - \frac{4}{M\pi}(1 + IJ - I - J) \int_0^{\pi/4} \prod_{l=1}^{L} M_{\gamma_l} \left(-\frac{3/(I^2+J^2-2)}{\sin^2\theta} \right) d\theta$$

From [5] and [2]:

$$P_b = \frac{1}{\log_2(IJ)}\left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l)\right)$$

$$P_I(k) = \frac{2}{I\pi}\sum_{i=0}^{(1-2^{-k})I-1}\left\{(-1)^{\left\lfloor \frac{i2^{k-1}}{I} \right\rfloor}\left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{I} + \frac{1}{2} \right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$

$$P_J(k) = \frac{2}{J\pi}\sum_{j=0}^{(1-2^{-l})J-1}\left\{(-1)^{\left\lfloor \frac{j2^{l-1}}{J} \right\rfloor}\left(2^{l-1} - \left\lfloor \frac{j2^{l-1}}{J} + \frac{1}{2} \right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2j+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$

M-DPSK with Postdetection EGC

From equation 8.165 in [2]:

$$P_s = \frac{\sin(\pi/M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{[1-\cos(\pi/M)\cos\theta]}\prod_{l=1}^{L} M_{\gamma_l}\left(-[1-\cos(\pi/M)\cos\theta]\right)d\theta$$

From [4] and [2]:

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')\bar{A}_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i, and

$$\bar{A}_i = \bar{F}\left((2i+1)\frac{\pi}{M}\right) - \bar{F}\left((2i-1)\frac{\pi}{M}\right)$$

$$\bar{F}(\psi) = -\frac{\sin\psi}{4\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{(1-\cos\psi\cos t)}\prod_{l=1}^{L} M_{\gamma_l}\left(-(1-\cos\psi\cos t)\right)dt$$

For the special case of Rayleigh fading with $M = 2$, and $L = 1$ (equation 8.173 from [2]):

$$P_b = \frac{1}{2(1+\bar{\gamma})}$$

Orthogonal 2-FSK, Coherent Detection with MRC

From equation 9.11 in [2]:

$$P_s = P_b = \frac{1}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1/2}{\sin^2\theta}\right)d\theta$$

For the special case of Rayleigh fading (equations 14.4-15 and 14.4-21 in [1]):

$$P_s = P_b = \frac{1}{2^L}\left(1-\sqrt{\frac{\bar{\gamma}}{2+\bar{\gamma}}}\right)^L \sum_{k=0}^{L-1}\binom{L-1+k}{k}\frac{1}{2^k}\left(1+\sqrt{\frac{\bar{\gamma}}{2+\bar{\gamma}}}\right)^k$$

Nonorthogonal 2-FSK, Coherent Detection with MRC

Equations 9.11 and 8.44 in [2]:

$$P_s = P_b = \frac{1}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(1-\mathrm{Re}[\rho])/2}{\sin^2\theta}\right)d\theta$$

For the special case of Rayleigh fading with $L = 1$ (equation 20 in [8] and equation 8.130 in [2]):

$$P_s = P_b = \frac{1}{2}\left[1-\sqrt{\frac{\bar{\gamma}(1-\mathrm{Re}[\rho])}{2+\bar{\gamma}(1-\mathrm{Re}[\rho])}}\right]$$

Orthogonal M-FSK, Noncoherent Detection with EGC

Rayleigh fading (equation 14.4-47 in [1]):

$$P_s = 1-\int_0^{\infty}\frac{1}{(1+\bar{\gamma})^L(L-1)!}U^{L-1}e^{-\frac{U}{1+\bar{\gamma}}}\left(1-e^{-U}\sum_{k=0}^{L-1}\frac{U^k}{k!}\right)^{M-1}dU$$

$$P_b = \frac{1}{2}\frac{M}{M-1}P_s$$

Rician fading (equation 41 in [8]):

$$P_s = \sum_{r=1}^{M-1} \frac{(-1)^{r+1} e^{-LK\overline{\gamma}_r/(1+\overline{\gamma}_r)}}{\left(r(1+\overline{\gamma}_r)+1\right)^L} \binom{M-1}{r} \sum_{n=0}^{r(L-1)} \beta_{nr} \frac{\Gamma(L+n)}{\Gamma(L)} \left[\frac{1+\overline{\gamma}_r}{r+1+r\overline{\gamma}_r}\right]^n {}_1F_1\left(L+n, L; \frac{LK\overline{\gamma}_r/(1+\overline{\gamma}_r)}{r(1+\overline{\gamma}_r)+1}\right)$$

$$P_b = \frac{1}{2}\frac{M}{M-1} P_s$$

where

$$\overline{\gamma}_r = \frac{1}{1+K}\overline{\gamma}$$

$$\beta_{nr} = \sum_{i=n-(L-1)}^{n} \frac{\beta_{i(r-1)}}{(n-i)!} I_{[0,(r-1)(L-1)]}(i)$$

$$\beta_{00} = \beta_{0r} = 1$$

$$\beta_{n1} = 1/n!$$

$$\beta_{1r} = r$$

and $I_{[a,b]}(i) = 1$ if $a \le i \le b$ and 0 otherwise.

Nonorthogonal 2-FSK, Noncoherent Detection with No Diversity

From equation 8.163 in [2]:

$$P_s = P_b = \frac{1}{4\pi} \int_{-\pi}^{\pi} \frac{1-\varsigma^2}{1+2\varsigma\sin\theta+\varsigma^2} M_\gamma\left(-\frac{1}{4}(1+\sqrt{1-\rho^2})(1+2\varsigma\sin\theta+\varsigma^2)\right) d\theta$$

where

$$\varsigma = \sqrt{\frac{1-\sqrt{1-\rho^2}}{1+\sqrt{1-\rho^2}}}$$

Analytical Expressions Used in bercoding and BERTool

Common Notation for This Section

Description	Notation
Energy-per-information bit-to-noise power-spectral-density ratio	$\gamma_b = \frac{E_b}{N_0}$
Message length	K
Code length	N
Code rate	$R_c = \frac{K}{N}$

Block Coding

Specific notation for block coding expressions: $d_{\min}$ is the minimum distance of the code.

Soft Decision

BPSK, QPSK, OQPSK, PAM-2, QAM-4, and precoded MSK (equation 8.1-52 in [1]):

$$P_b \le \frac{1}{2}(2^K - 1)Q\left(\sqrt{2\gamma_b R_c d_{\min}}\right)$$

DE-BPSK, DE-QPSK, DE-OQPSK, and DE-MSK:

$$P_b \le \frac{1}{2}(2^K - 1)\left[2Q\left(\sqrt{2\gamma_b R_c d_{\min}}\right)\left[1 - Q\left(\sqrt{2\gamma_b R_c d_{\min}}\right)\right]\right]$$

BFSK, coherent detection (equations 8.1-50 and 8.1-58 in [1]):

$$P_b \le \frac{1}{2}(2^K - 1)Q\left(\sqrt{\gamma_b R_c d_{\min}}\right)$$

BFSK, noncoherent square-law detection (equations 8.1-65 and 8.1-64 in [1]):

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp\left(-\frac{1}{2}\gamma_b R_c d_{\min}\right)\sum_{i=0}^{d_{\min}-1}\left(\frac{1}{2}\gamma_b R_c d_{\min}\right)^i \frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

DPSK:

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp\left(-\gamma_b R_c d_{\min}\right)\sum_{i=0}^{d_{\min}-1}\left(\gamma_b R_c d_{\min}\right)^i \frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

Hard Decision

General linear block code (equations 4.3, 4.4 in [9], and 12.136 in [6]):

$$P_b \le \frac{1}{N}\sum_{m=t+1}^{N}(m+t)\binom{N}{m}p^m(1-p)^{N-m}$$

$$t = \left\lfloor \frac{1}{2}(d_{\min}-1) \right\rfloor$$

Hamming code (equations 4.11, 4.12 in [9], and 6.72, 6.73 in [7]):

$$P_b \approx \frac{1}{N}\sum_{m=2}^{N} m\binom{N}{m}p^m(1-p)^{N-m} = p - p(1-p)^{N-1}$$

(24, 12) extended Golay code (equation 4.17 in [9], and 12.139 in [6]):

$$P_b \le \frac{1}{24}\sum_{m=4}^{24}\beta_m\binom{24}{m}p^m(1-p)^{24-m}$$

where β_m is the average number of channel symbol errors that remain in corrected N-tuple when the channel caused m symbol errors (table 4.2 in [9]).

Reed-Solomon code with $N = Q - 1 = 2^q - 1$:

$$P_b \approx \frac{2^{q-1}}{2^q - 1}\frac{1}{N}\sum_{m=t+1}^{N} m\binom{N}{m}(P_s)^m(1-P_s)^{N-m}$$

for FSK (equations 4.25, 4.27 in [9], 8.1-115, 8.1-116 in [1], 8.7, 8.8 in [7], and 12.142, 12.143 in [6]), and

$$P_b \approx \frac{1}{q}\frac{1}{N}\sum_{m=t+1}^{N} m\binom{N}{m}(P_s)^m(1-P_s)^{N-m}$$

otherwise.

If $\log_2 Q / \log_2 M = q / k = h$ where h is an integer (equation 1 in [10]):

$$P_s = 1 - (1 - s)^h$$

where s is the symbol error rate (SER) in an uncoded AWGN channel.

For example, for BPSK, $M = 2$ and $P_s = 1 - (1 - s)^q$

Otherwise, P_s is given by table 1 and equation 2 in [10].

Convolutional Coding

Specific notation for convolutional coding expressions: d_{free} is the free distance of the code, and a_d is the number of paths of distance d from the all-zero path that merge with the all-zero path for the first time.

Soft Decision

From equations 8.2-26, 8.2-24, and 8.2-25 in [1], and equations 13.28 and 13.27 in [6]:

$$P_b < \sum_{d=d_{free}}^{\infty} a_d f(d) P_2(d)$$

with transfer function

$$T(D,N) = \sum_{d=d_{free}}^{\infty} a_d D^d N^{f(d)}$$

$$\left.\frac{dT(D,N)}{dN}\right|_{N=1} = \sum_{d=d_{free}}^{\infty} a_d f(d) D^d$$

where $f(d)$ is the exponent of N as a function of d.

Results for BPSK, QPSK, OQPSK, PAM-2, QAM-4, precoded MSK, DE-BPSK, DE-QPSK, DE-OQPSK, DE-MSK, DPSK, and BFSK are obtained as:

$$P_2(d) = P_b\Big|_{\frac{E_b}{N_0}=\gamma_b R_c d}$$

where P_b is the BER in the corresponding uncoded AWGN channel. For example, for BPSK (equation 8.2-20 in [1]):

$$P_2(d) = Q\left(\sqrt{2\gamma_b R_c d}\right)$$

Hard Decision

From equations 8.2-33, 8.2-28, and 8.2-29 in [1], and equations 13.28, 13.24, and 13.25 in [6]:

$$P_b < \sum_{d=d_{free}}^{\infty} a_d f(d) P_2(d)$$

where

$$P_2(d) = \sum_{k=(d+1)/2}^{d} \binom{d}{k} p^k (1-p)^{d-k}$$

when d is odd, and

$$P_2(d) = \sum_{k=d/2+1}^{d} \binom{d}{k} p^k (1-p)^{d-k} + \frac{1}{2}\binom{d}{d/2} p^{d/2}(1-p)^{d/2}$$

when d is even (p is the bit error rate (BER) in an uncoded AWGN channel).

Performance Results via Simulation

Section Overview

One way to compute the bit error rate or symbol error rate for a communication system is to simulate the transmission of data messages and compare all messages before and after transmission. The simulation of the communication system components using Communications Toolbox is covered in other parts of this guide. This section describes how to compare the data messages that enter and leave the simulation.

Another example of computing performance results via simulation is in "Curve Fitting for Error Rate Plots" on page 11-33 in the discussion of curve fitting.

Using Simulated Data to Compute Bit and Symbol Error Rates

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

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

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

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

If the error rate is very small (for example, 10^{-6} or smaller), the semianalytic technique might compute the result more quickly than a simulation-only approach. See "Performance Results via the Semianalytic Technique" on page 11-25 for more information on how to use this technique.

Example: Computing Error Rates

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

the resulting noisy code to the original code. Then it decodes and compares the decoded message to the original one.

```
m = 3; n = 2^m-1; k = n-m; % Prepare to use Hamming code.
msg = randi([0 1],k*200,1); % 200 messages of k bits each
code = encode(msg,n,k,'hamming');
codenoisy = rem(code+(rand(n*200,1)>.95),2); % Add noise.
% Decode and correct some errors.
newmsg = decode(codenoisy,n,k,'hamming');
% Compute and display symbol error rates.
noisyVec = step(comm.ErrorRate,code,codenoisy);
decodedVec = step(comm.ErrorRate,msg,newmsg);
disp(['Error rate in the received code: ',num2str(noisyVec(1))])
disp(['Error rate after decoding: ',num2str(decodedVec(1))])
```

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

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

Comparing Symbol Error Rate and Bit Error Rate

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

```
a = [1 2 3]'; b = [1 4 4]';
format rat % Display fractions instead of decimals.
% Create ErrorRate Calculator System object
serVec = step(comm.ErrorRate,a,b);
srate = serVec(1)
snum = serVec(2)
% Convert integers to bits
hIntToBit = comm.IntegerToBit(3);
a_bit = step(hIntToBit, a);
b_bit = step(hIntToBit, b);
% Calculate BER
berVec = step(comm.ErrorRate,a_bit,b_bit);
brate = berVec(1)
bnum = berVec(2)
```

The output is below.

```
snum =

      2


srate =

     2/3


bnum =

      5


brate =

     5/9
```

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

Performance Results via the Semianalytic Technique

The technique described in "Performance Results via Simulation" on page 11-22 works well for a large variety of communication systems, but can be prohibitively time-consuming if the system's error rate is very small (for example, 10^{-6} or smaller). This section describes how to use the semianalytic technique as an alternative way to compute error rates. For certain types of systems, the semianalytic technique can produce results much more quickly than a nonanalytic method that uses only simulated data.

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

When to Use the Semianalytic Technique

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

- Any effects of multipath fading, quantization, and amplifier nonlinearities must *precede* the effects of noise in the actual channel being modeled.
- The receiver is perfectly synchronized with the carrier, and timing jitter is negligible. Because phase noise and timing jitter are slow processes, they reduce the applicability of the semianalytic technique to a communication system.
- The noiseless simulation has no errors in the received signal constellation. Distortions from sources other than noise should be mild enough to keep each signal point in its correct decision region. If this is not the case, the calculated BER is too low. For instance, if the modeled system has a phase rotation that places the received signal points outside their proper decision regions, the semianalytic technique is not suitable to predict system performance.

Furthermore, the `semianalytic` function assumes that the noise in the actual channel being modeled is Gaussian. For details on how to adapt the semianalytic technique for non-Gaussian noise, see the discussion of generalized exponential distributions in [11].

Procedure for the Semianalytic Technique

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

1. Generate a message signal containing *at least* M^L symbols, where M is the alphabet size of the modulation and L is the length of the impulse response of the channel in symbols. A common approach is to start with an augmented binary pseudonoise (PN) sequence of total length $(\log_2 M)M^L$. An *augmented* PN sequence is a PN sequence with an extra zero appended, which makes the distribution of ones and zeros equal.
2. Modulate a carrier with the message signal using baseband modulation. Supported modulation types are listed on the reference page for `semianalytic`. Shape the resultant signal with rectangular pulse shaping, using the oversampling factor that you will later use to filter the modulated signal. Store the result of this step as `txsig` for later use.
3. Filter the modulated signal with a transmit filter. This filter is often a square-root raised cosine filter, but you can also use a Butterworth, Bessel, Chebyshev type 1 or 2, elliptic, or more general FIR or IIR filter. If you use a square-root raised cosine filter, use it on the nonoversampled modulated signal and specify the oversampling factor in the filtering function. If you use another filter type, you can apply it to the rectangularly pulse shaped signal.
4. Run the filtered signal through a *noiseless* channel. This channel can include multipath fading effects, phase shifts, amplifier nonlinearities, quantization, and

additional filtering, but it must not include noise. Store the result of this step as `rxsig` for later use.

5 Invoke the `semianalytic` function using the `txsig` and `rxsig` data from earlier steps. Specify a receive filter as a pair of input arguments, unless you want to use the function's default filter. The function filters `rxsig` and then determines the error probability of each received signal point by analytically applying the Gaussian noise distribution to each point. The function averages the error probabilities over the entire received signal to determine the overall error probability. If the error probability calculated in this way is a symbol error probability, the function converts it to a bit error rate, typically by assuming Gray coding. The function returns the bit error rate (or, in the case of DQPSK modulation, an upper bound on the bit error rate).

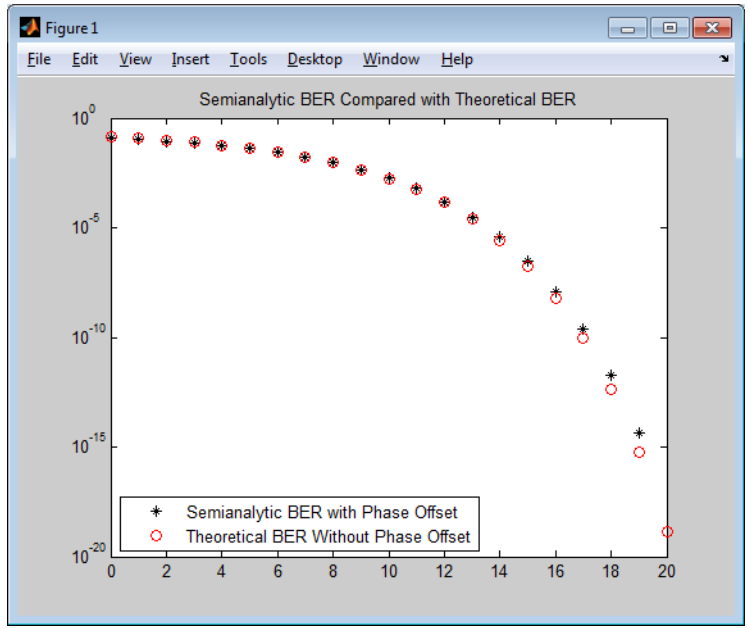
Example: Using the Semianalytic Technique

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

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

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

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

% Step 4. Run txsig through a noiseless channel.
rxsig = txsig*exp(1i*pi/180); % Static phase offset of 1 degree
% Step 5. Use the semianalytic function.
% Specify the receive filter as a pair of input arguments.
% In this case, num and den describe an ideal integrator.
num = ones(Nsamp,1)/Nsamp;
```

```
den = 1;
EbNo = 0:20; % Range of Eb/No values under study
ber = semianalytic(txsig,rxsig,'qam',M,Nsamp,num,den,EbNo);

% For comparison, calculate theoretical BER.
bertheory = berawgn(EbNo,'qam',M);

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

This example creates a figure like the one below.



Theoretical Performance Results

- "Computing Theoretical Error Statistics" on page 11-29
- "Plotting Theoretical Error Rates" on page 11-29

- "Comparing Theoretical and Empirical Error Rates" on page 11-30

Computing Theoretical Error Statistics

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

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

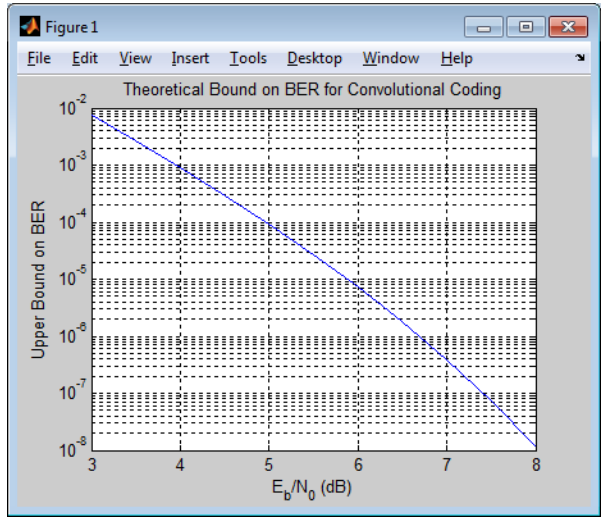
Each function's reference page lists one or more books containing the closed-form expressions that the function implements.

Plotting Theoretical Error Rates

The example below uses the `bercoding` function to compute upper bounds on bit error rates for convolutional coding with a soft-decision decoder. The data used for the generator and distance spectrum are from [1] and [12], respectively.

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

This example produces the following plot.



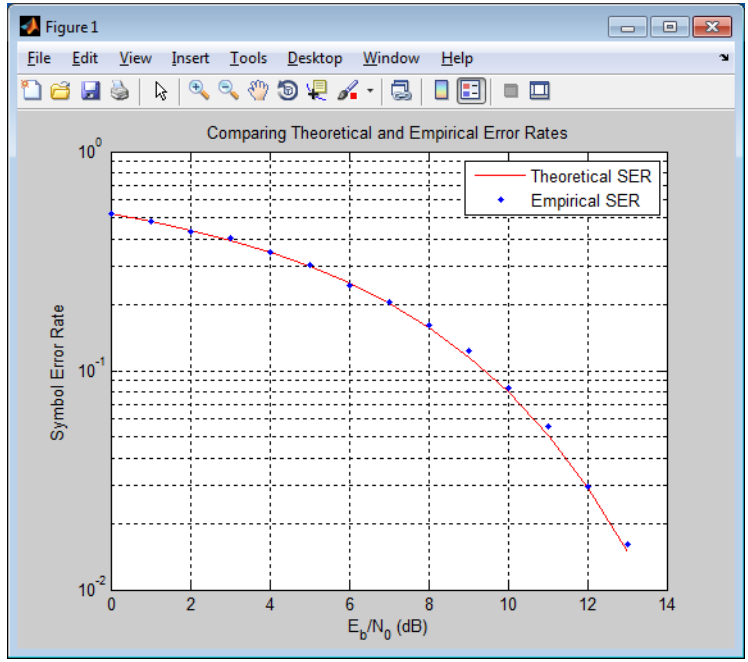
Comparing Theoretical and Empirical Error Rates

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

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

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

```
% "Selected Bibliography for Performance Evaluation."
snr = EbNo+3+10*log10(k);
% Preallocate variables to save time.
ynoisy = zeros(n,length(snr));
z      = zeros(n,length(snr));
berVec = zeros(3,length(EbNo));

% PAM modulation and demodulation system objects
h  = comm.PAMModulator(M);
h2 = comm.PAMDemodulator(M);

% AWGNChannel System object
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');


% ErrorRate calculator System object to compare decoded symbols to the
% original transmitted symbols.
hErrorCalc = comm.ErrorRate;

% Main steps in the simulation
x = randi([0 M-1],n,1); % Create message signal.
y = step(h,x); % Modulate.
hChan.SignalPower = (real(y)' * real(y))/ length(real(y));

% Loop over different SNR values.
for jj = 1:length(snr)
    reset(hErrorCalc)
    hChan.SNR = snr(jj); % Assign Channel SNR
    ynoisy(:,jj) = step(hChan,real(y)); % Add AWGN
    z(:,jj) = step(h2,complex(ynoisy(:,jj))); % Demodulate.

    % Compute symbol error rate from simulation.
    berVec(:,jj) = step(hErrorCalc, x, z(:,jj));
end

% 3. Plot empirical results, in same figure.
hold on; semilogy(EbNo,berVec(1,:),'b.');
legend('Theoretical SER','Empirical SER');
title('Comparing Theoretical and Empirical Error Rates');
hold off;
```

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



Error Rate Plots

Section Overview

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

Creating Error Rate Plots Using `semilogy`

In many error rate plots, the horizontal axis indicates E_b/N_0 values in dB and the vertical axis indicates the error rate using a logarithmic (base 10) scale. To see an example of such a plot, as well as the code that creates it, see "Comparing Theoretical and Empirical Error Rates" on page 11-30. The part of that example that creates the plot uses the `semilogy` function to produce a logarithmic scale on the vertical axis and a linear scale on the horizontal axis.

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

- "Example: Using the Semianalytic Technique" on page 11-27, which also illustrates
  - Plotting two sets of data on one pair of axes
  - Adding a title
  - Adding a legend
- "Plotting Theoretical Error Rates" on page 11-29, which also illustrates
  - Adding axis labels
  - Adding grid lines

Curve Fitting for Error Rate Plots

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

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

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

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

Example: Curve Fitting for an Error Rate Plot

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

Setting Up Parameters for the Simulation

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

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

```
% Set up initial parameters.
siglen = 100000; % Number of bits in each trial
M = 2; % DBPSK is binary.
% DBPSK modulation and demodulation System objects
hMod  = comm.DBPSKModulator;
hDemod = comm.DBPSKDemodulator;
% AWGNChannel System object
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');
% ErrorRate calculator System object to compare decoded symbols to the
% original transmitted symbols.
hErrorCalc = comm.ErrorRate;
```

```
EbNomin = 0; EbNomax = 9; % EbNo range, in dB
numerrmin = 5; % Compute BER only after 5 errors occur.
EbNovec = EbNomin:1:EbNomax; % Vector of EbNo values
numEbNos = length(EbNovec); % Number of EbNo values
% Preallocate space for certain data.
ber = zeros(1,numEbNos); % final BER values
berVec = zeros(3,numEbNos); % Updated BER values
intv = cell(1,numEbNos); % Cell array of confidence intervals
```

Simulating the System Using a Loop

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

The communication system simulation uses these toolbox functions:

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

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

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

```
% Loop over the vector of EbNo values.
berVec = zeros(3,numEbNos); % Reset
for jj = 1:numEbNos
    EbNo = EbNovec(jj);
    snr = EbNo; % Because of binary modulation
    reset(hErrorCalc)
    hChan.SNR = snr; % Assign Channel SNR
```

```
    % Simulate until numerrmin errors occur.
    while (berVec(2,jj) < numerrmin)
        msg = randi([0,M-1], siglen, 1); % Generate message sequence.
        txsig = step(hMod, msg); % Modulate.
        hChan.SignalPower = (txsig'*txsig)/length(txsig);  % Calculate and
        % assign signal power
        rxsig = step(hChan,txsig); % Add noise.
        decodmsg = step(hDemod, rxsig); % Demodulate.
        if (berVec(2,jj)==0)
            % The first symbol of a differentially encoded transmission
            % is discarded.
            berVec(:,jj) = step(hErrorCalc, msg(2:end),decodmsg(2:end));
        else
            berVec(:,jj) = step(hErrorCalc, msg, decodmsg);
        end
    end
    % Error rate and 98% confidence interval for this EbNo value
    [ber(jj), intv1] = berconfint(berVec(2,jj),berVec(3,jj)-1,.98);
    intv{jj} = intv1; % Store in cell array for later use.
    disp(['EbNo = ' num2str(EbNo) ' dB, ' num2str(berVec(2,jj)) ...
        ' errors, BER = ' num2str(ber(jj))])
end
```

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

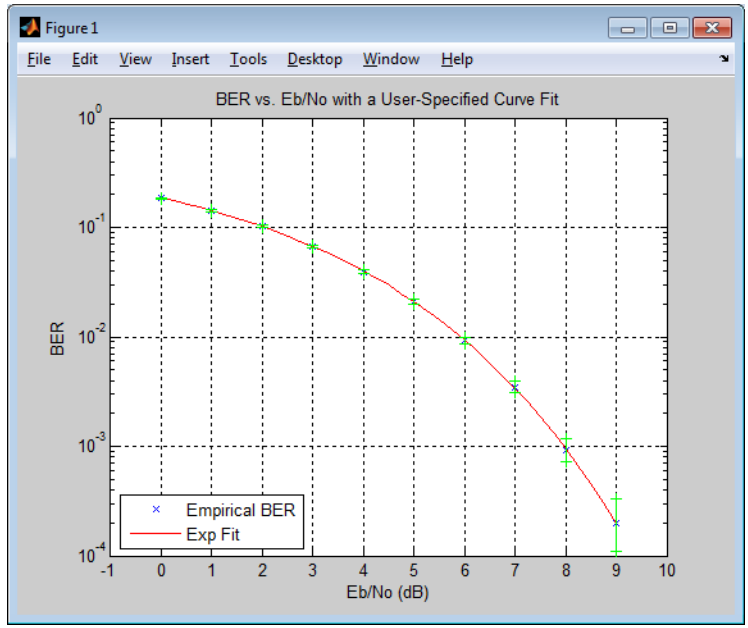
```
EbNo = 0 dB, 189 errors, BER = 0.18919
EbNo = 1 dB, 139 errors, BER = 0.13914
EbNo = 2 dB, 105 errors, BER = 0.10511
EbNo = 3 dB, 66 errors, BER = 0.066066
EbNo = 4 dB, 40 errors, BER = 0.04004
EbNo = 5 dB, 18 errors, BER = 0.018018
EbNo = 6 dB, 6 errors, BER = 0.006006
EbNo = 7 dB, 11 errors, BER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

Plotting the Empirical Results and the Fitted Curve

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

```
% Use BERFIT to plot the best fitted curve,
```

```
% interpolating to get a smooth plot.
fitEbNo = EbNomin:0.25:EbNomax; % Interpolation values
berfit(EbNovec,ber,fitEbNo,[],'exp');

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



BERTool

The command `bertool` launches the Bit Error Rate Analysis Tool (BERTool) application.

The application enables you to analyze the bit error rate (BER) performance of communications systems. BERTool computes the BER as a function of signal-to-noise ratio. It analyzes performance either with Monte-Carlo simulations of MATLAB functions and Simulink models or with theoretical closed-form expressions for selected types of communication systems.

Using BERTool you can:

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

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

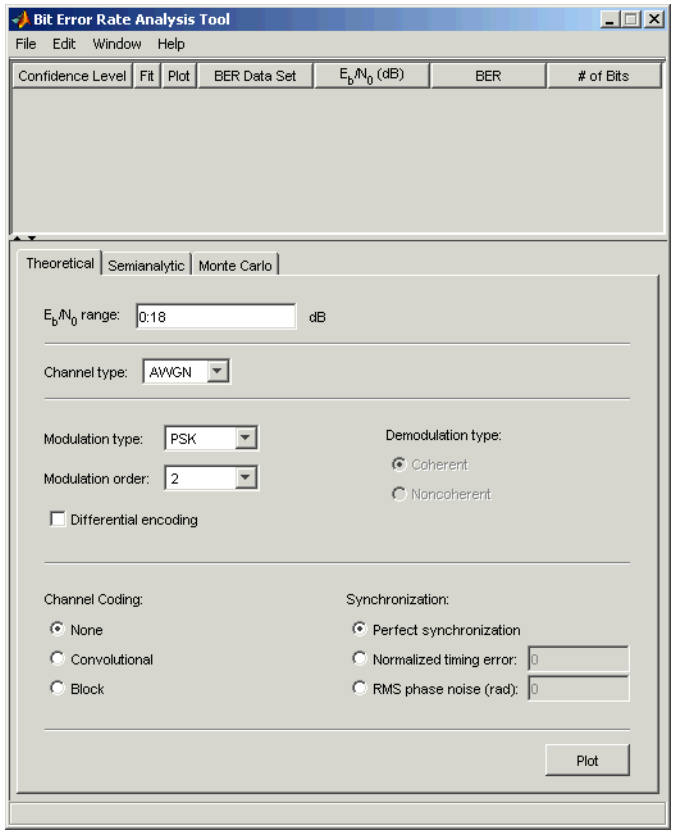
Note: BERTool is designed for analyzing bit error rates only, not symbol error rates, word error rates, or other types of error rates. If, for example, your simulation computes a symbol error rate (SER), convert the SER to a BER before using the simulation with BERTool.

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

Start BERTool

To open BERTool, type

```
bertool
```

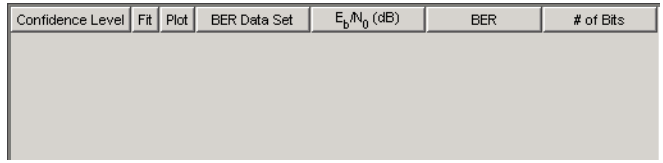


The BERTool Environment

- "Components of BERTool" on page 11-39
- "Interaction Among BERTool Components" on page 11-41

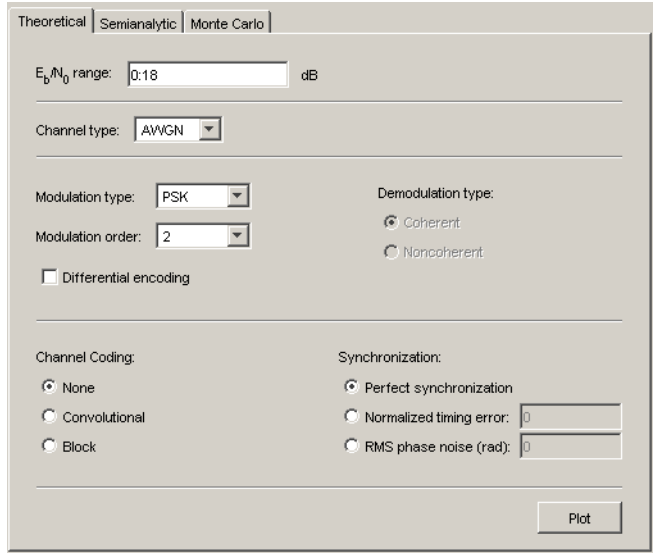
Components of BERTool

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



After you instruct BERTool to generate one or more BER data sets, they appear in the data viewer. An example that shows how data sets look in the data viewer is in "Example: Using a MATLAB Simulation with BERTool" on page 11-53.

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



To learn more about each of the methods, see

- "Computing Theoretical BERs" on page 11-42
- "Using the Semianalytic Technique to Compute BERs" on page 11-48
- "Run MATLAB Simulations" on page 11-53 or "Run Simulink Simulations" on page 11-66

- A separate BER Figure window, which displays some or all of the BER data sets that are listed in the data viewer. BERTool opens the BER Figure window after it has at least one data set to display, so you do not see the BER Figure window when you first open BERTool. For an example of how the BER Figure window looks, see "Example: Using the Theoretical Tab in BERTool" on page 11-43.

Interaction Among BERTool Components

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

- If you select a data set in the data viewer, BERTool reconfigures the tabs to reflect the parameters associated with that data set and also highlights the corresponding data in the BER Figure window. This is useful if the data viewer displays multiple data sets and you want to recall the meaning and origin of each data set.
- If you click data plotted in the BER Figure window, BERTool reconfigures the tabs to reflect the parameters associated with that data and also highlights the corresponding data set in the data viewer.

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

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

Note: If you want to observe the integration yourself but do not yet have any data sets in BERTool, then first try the procedure in "Example: Using the Theoretical Tab in BERTool" on page 11-43.

Note: If you save the BER Figure window using the window's **File** menu, the resulting file contains the contents of the window but not the BERTool data that led to the plot. To save an entire BERTool session, see "Saving a BERTool Session" on page 11-84.

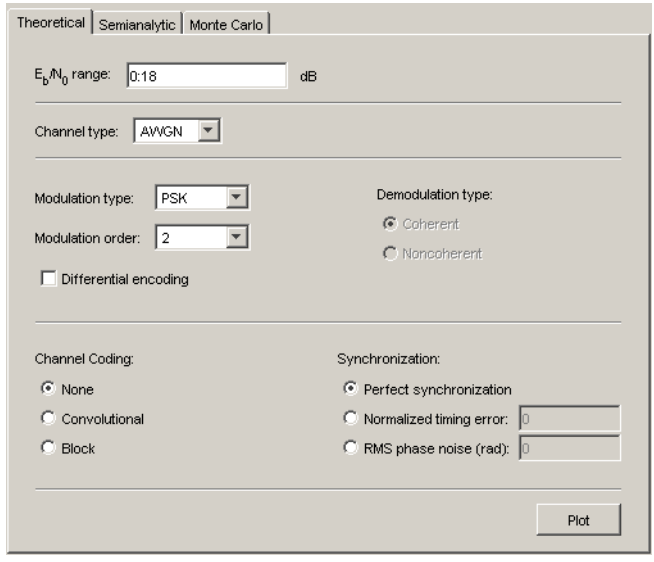
Computing Theoretical BERs

Section Overview

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

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

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



2 Set the parameters to reflect the system whose performance you want to analyze. Some parameters are visible and active only when other parameters have specific values. See “Available Sets of Theoretical BER Data” on page 11-45 for details.

3 Click **Plot**.

For an example that shows how to generate and analyze theoretical BER data via BERTool, see “Example: Using the Theoretical Tab in BERTool” on page 11-43.

Also, “Available Sets of Theoretical BER Data” on page 11-45 indicates which combinations of parameters are available on the **Theoretical** tab and which underlying functions perform computations.

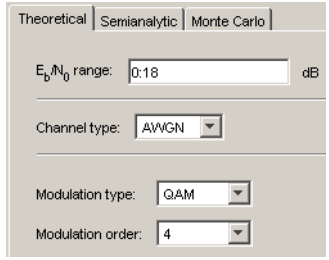
Example: Using the Theoretical Tab in BERTool

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

Running the Theoretical Example

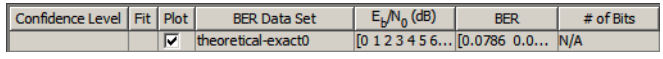
1 Open BERTool, and go to the **Theoretical** tab.

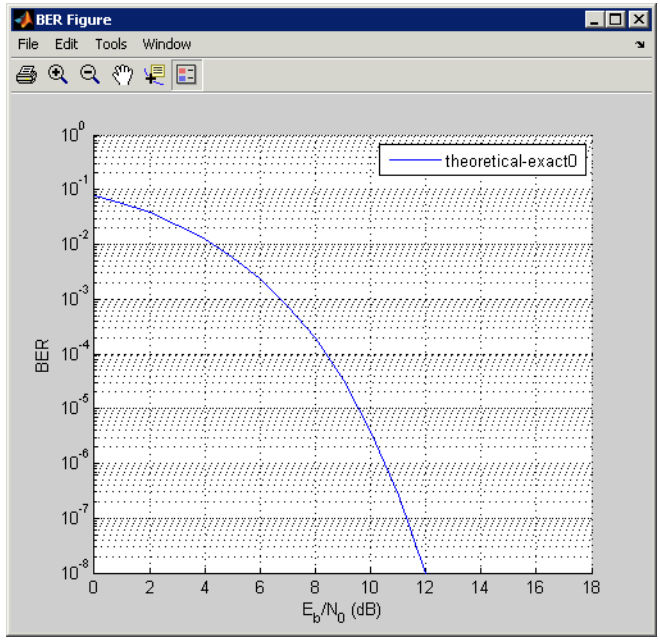
2 Set the parameters as shown in the following figure.



3 Click **Plot**.

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





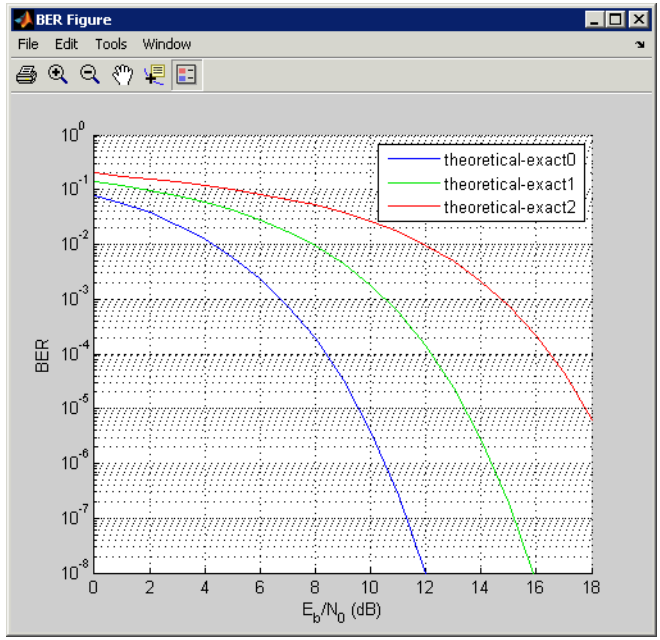
4 Change the **Modulation order** parameter to 16, and click **Plot**.

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

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

BERTool creates another entry in the data viewer and plots the new data in the same BER Figure window, as shown in the following figures.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	theoretical-exact0	[0 1 2 3 4 5 6...	[0.0786 0.0...	N/A
		☑	theoretical-exact1	[0 1 2 3 4 5 6...	[0.1409 0.1...	N/A
		☑	theoretical-exact2	[0 1 2 3 4 5 6...	[0.1998 0.1...	N/A



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

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

Available Sets of Theoretical BER Data

BERTool can generate a large set of theoretical bit-error rates, but not all combinations of parameters are currently supported. The **Theoretical** tab adjusts itself to your choices, so that the combination of parameters is always valid. You can set the **Modulation order** parameter by selecting a choice from the menu or by typing a value in the field. The **Normalized timing error** must be between 0 and 0.5.

BERTool assumes that Gray coding is used for all modulations.

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

Combinations of Parameters for AWGN Channel Systems

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

Modulation	Modulation Order	Other Choices
PSK	2, 4	Differential or nondifferential encoding.
	8, 16, 32, 64, or a higher power of 2	
OQPSK	4	Differential or nondifferential encoding.
DPSK	2, 4, 8, 16, 32, 64, or a higher power of 2	
PAM	2, 4, 8, 16, 32, 64, or a higher power of 2	
QAM	4, 8, 16, 32, 64, 128, 256, 512, 1024, or a higher power of 2	
FSK	2	Orthogonal or nonorthogonal; `Coherent` or `Noncoherent` demodulation.
	4, 8, 16, 32, or a higher power of 2	Orthogonal; `Coherent` demodulation.
	4, 8, 16, 32, or 64	Orthogonal; `Noncoherent` demodulation.
MSK	2	`Coherent` conventional or precoded MSK; `Noncoherent` precoded MSK.
CPFSK	2, 4, 8, 16, or a higher power of 2	`Modulation index` > 0.

BER results are also available for the following:

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

- convolutional coding with soft-decision decoding for all binary modulations (including 4-PSK and 4-QAM) except CPFSK
- uncoded nondifferentially-encoded 2-PSK with synchronization errors

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

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

Combinations of Parameters for Rayleigh and Rician Channel Systems

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

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

Modulation	Modulation Order	Other Choices
PSK	2	Differential or nondifferential encoding `Diversity order` #1 In the case of nondifferential encoding, diversity order being 1, and Rician fading, a value for RMS phase noise (in radians) can be specified.
	4, 8, 16, 32, 64, or a higher power of 2	`Diversity order` #1
OQPSK	4	`Diversity order` #1
DPSK	2, 4, 8, 16, 32, 64, or a higher power of 2	`Diversity order` #1
PAM	2, 4, 8, 16, 32, 64, or a higher power of 2	`Diversity order` #1

Modulation	Modulation Order	Other Choices
QAM	4, 8, 16, 32, 64, 128, 256, 512, 1024, or a higher power of 2	`Diversity order` #1
FSK	2	Correlation coefficient $\in [-1, 1]$. `Coherent` or `Noncoherent` demodulation `Diversity order` #1 In the case of a nonzero correlation coefficient and noncoherent demodulation, the diversity order is 1 only.
	4, 8, 16, 32, or a higher power of 2	`Noncoherent` demodulation only. `Diversity order` #1

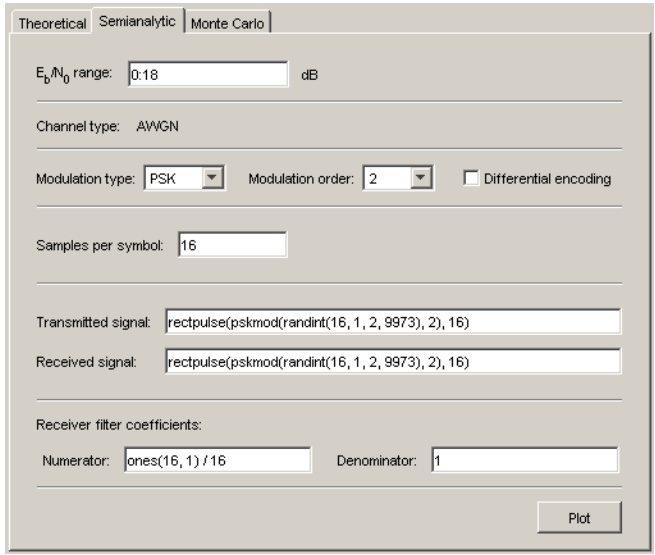
For more information about specific combinations of parameters, including bibliographic references that contain closed-form expressions, see the reference page for the `berfading` function.

Using the Semianalytic Technique to Compute BERs

Section Overview

You can use BERTool to generate and analyze BER data via the semianalytic technique. The semianalytic technique is discussed in "Performance Results via the Semianalytic Technique" on page 11-25, and "When to Use the Semianalytic Technique" on page 11-25 is particularly relevant as background material.

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



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

Example: Using the Semianalytic Tab in BERTool

This example illustrates how BERTool applies the semianalytic technique, using 16-QAM modulation. This example is a variation on the example in "Example: Using the Semianalytic Technique" on page 11-27, but it is tailored to use BERTool instead of using the `semianalytic` function directly.

Running the Semianalytic Example

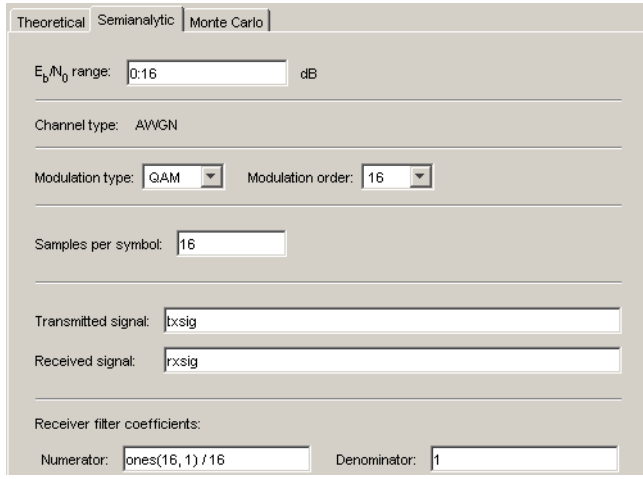
1. To set up the transmitted and received signals, run steps 1 through 4 from the code example in "Example: Using the Semianalytic Technique" on page 11-27. The code is repeated below.

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

% Step 2. Modulate the message signal using baseband modulation.
hMod = comm.RectangularQAMModulator(M);  % Use 16-QAM.
```

```
modsig = step(hMod,msg'); % Modulate data
Nsamp = 16;
modsig = rectpulse(modsig,Nsamp); % Use rectangular pulse shaping.

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

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

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

3 Set parameters as shown in the following figure.



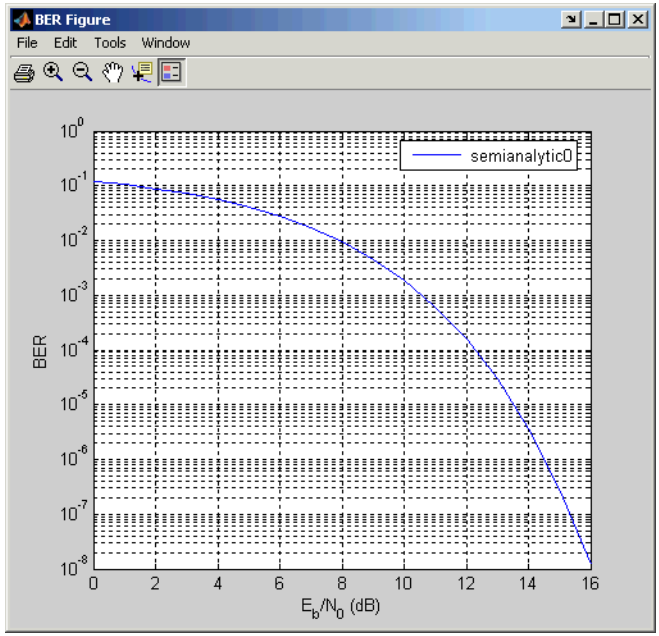
4 Click **Plot**.

Visible Results of the Semianalytic Example

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

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	semianalytic0	[0 1 2 3 4 5 6...	[0.1199 0.1...	[272]

BERTool plots the data in the BER Figure window.



Procedure for Using the Semianalytic Tab in BERTool

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

1. Generate a message signal containing *at least* M^L symbols, where M is the alphabet size of the modulation and L is the length of the impulse response of the channel in symbols. A common approach is to start with an augmented binary pseudonoise (PN) sequence of total length $(\log_2 M)M^L$. An *augmented* PN sequence is a PN sequence with an extra zero appended, which makes the distribution of ones and zeros equal.
2. Modulate a carrier with the message signal using baseband modulation. Supported modulation types are listed on the reference page for `semianalytic`. Shape the resultant signal with rectangular pulse shaping, using the oversampling factor that you will later use to filter the modulated signal. Store the result of this step as `txsig` for later use.
3. Filter the modulated signal with a transmit filter. This filter is often a square-root raised cosine filter, but you can also use a Butterworth, Bessel, Chebyshev type 1 or 2, elliptic, or more general FIR or IIR filter. If you use a square-root raised cosine

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

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

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

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

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

6 Click **Plot**.

Semianalytic Computations and Results

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

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

Run MATLAB Simulations

Section Overview

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

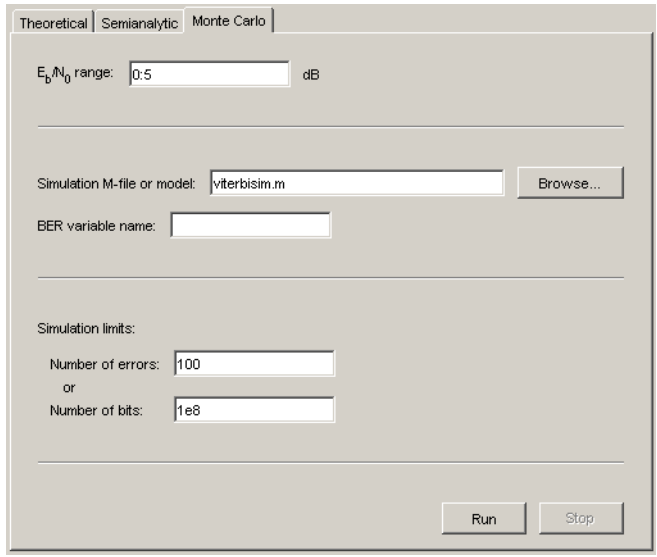
To learn how to make your own simulation functions compatible with BERTool, see "Use Simulation Functions with BERTool" on page 11-59.

Example: Using a MATLAB Simulation with BERTool

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

To run this example, follow these steps:

1. Open BERTool and go to the **Monte Carlo** tab. (The default parameters depend on whether you have Communications System Toolbox software installed. Also note that the **BER variable name** field applies only to Simulink models.)
2. Set parameters as shown in the following figure.



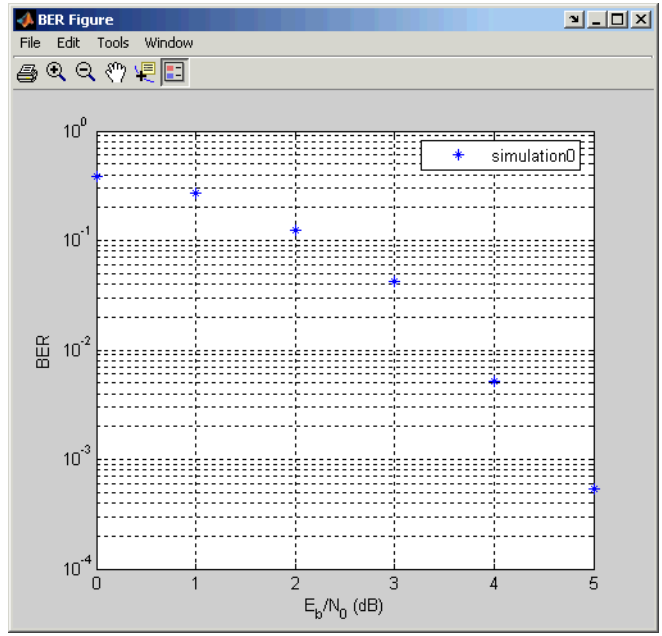
3. Click **Run**.

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

   Then BERTool creates a listing in the data viewer.

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☐	☑	simulation0	[0 1 2 3 4 5]	[0.3743 0.2...	[10000 1000...

   BERTool plots the data in the BER Figure window.

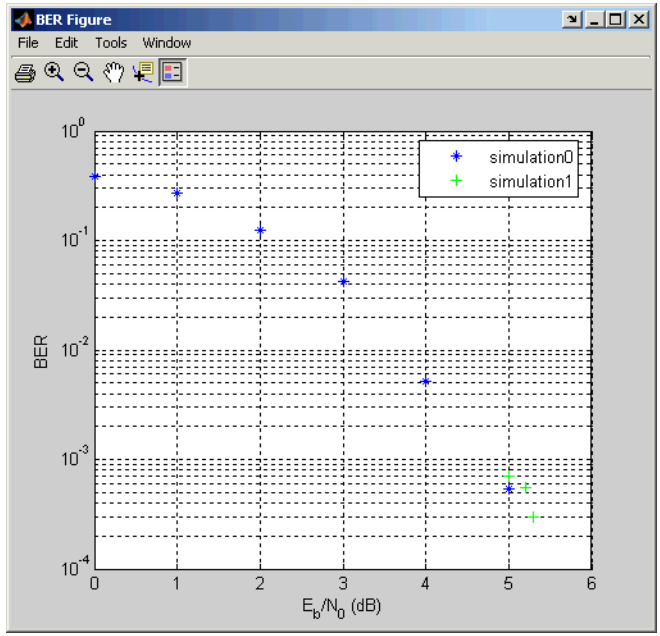


4 To change the range of E_b/N_0 while reducing the number of bits processed in each case, type `[5 5.2 5.3]` in the **Eb/No range** field, type `1e5` in the **Number of bits** field, and click **Run**.

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

Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☐	☑	simulation0	[0 1 2 3 4 5]	[0.3739 0.2...	[10000 1000...
off		☑	simulation1	[5 5.2 5.3]	[3.37E-4 5.4...	[109680 109...

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



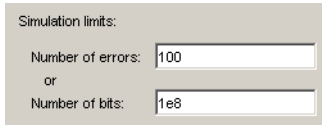
The two points corresponding to 5 dB from the two data sets are different because the smaller value of **Number of bits** in the second simulation caused the simulation to end before observing many errors. To learn more about the criteria that BERTool uses for ending simulations, see "Varying the Stopping Criteria" on page 11-56.

For another example that uses BERTool to run a MATLAB simulation function, see "Example: Prepare a Simulation Function for Use with BERTool" on page 11-63.

Varying the Stopping Criteria

When you create a MATLAB simulation function for use with BERTool, you must control the flow so that the simulation ends when it either detects a target number of errors or processes a maximum number of bits, whichever occurs first. To learn more about this requirement, see "Requirements for Functions" on page 11-59; for an example, see "Example: Prepare a Simulation Function for Use with BERTool" on page 11-63.

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



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

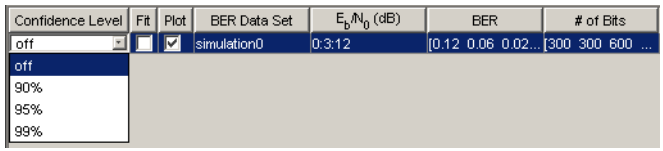
As an example, follow the procedure described in "Example: Using a MATLAB Simulation with BERTool" on page 11-53 and set **Confidence Level** to 95 for each of the two data sets. The confidence intervals for the second data set are larger than those for the first data set. This is because the second data set uses a small value for **Number of bits** relative to the communication system properties and the values in **Eb/No range**, resulting in BER values based on only a small number of observed errors.

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

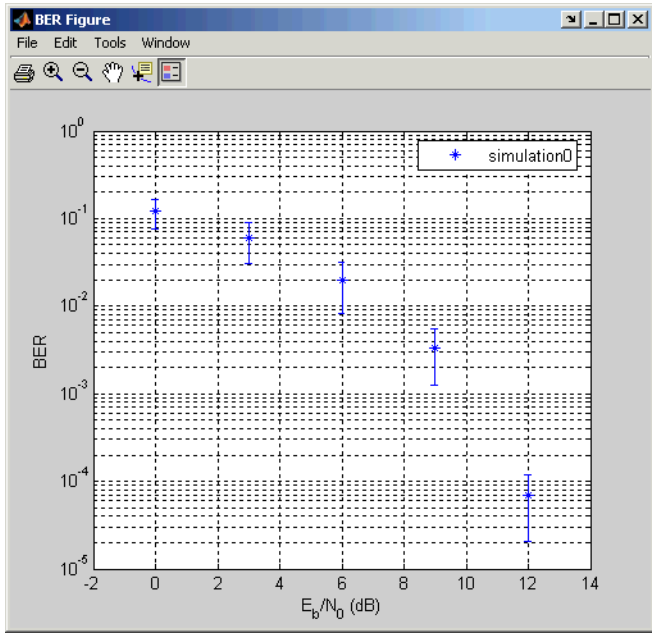
Plotting Confidence Intervals

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

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



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



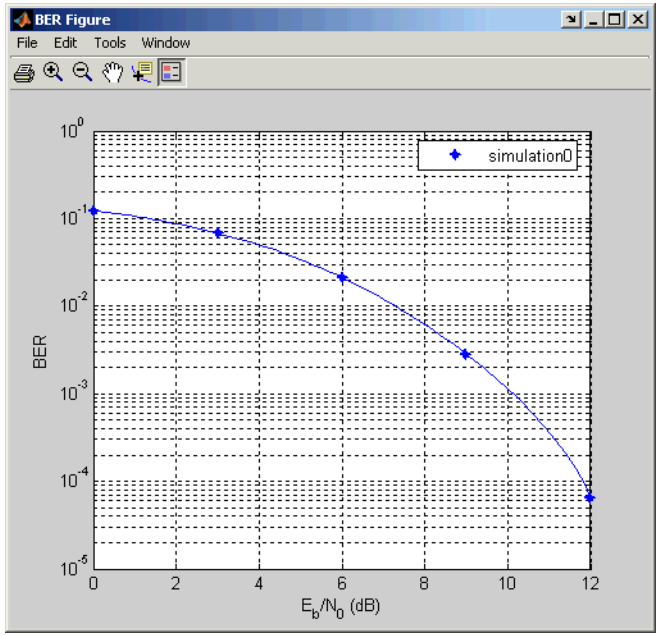
For an example that plots confidence intervals for a Simulink simulation, see "Example: Using a Simulink Model with BERTool" on page 11-67.

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

Fitting BER Points to a Curve

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

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



For an example that performs curve fitting for data from a Simulink simulation and generates the plot shown above, see “Example: Using a Simulink Model with BERTool” on page 11-67. For an example that performs curve fitting for data from a MATLAB simulation function, see “Example: Prepare a Simulation Function for Use with BERTool” on page 11-63.

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

Use Simulation Functions with BERTool

Requirements for Functions

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

Input Arguments

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

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

Output Arguments

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

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

BERTool uses these output arguments when reporting and plotting results.

Simulation Operation

Your simulation function must perform these tasks:

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

Template for a Simulation Function

Use the following template when adapting your code to work with BERTool. You can open it in an editor by entering `edit bertooltemplate` in the MATLAB Command Window. The description in "Understanding the Template" on page 11-61 explains the template's key sections, while "Using the Template" on page 11-62 indicates how to use the template with your own simulation code. Alternatively, you can develop your simulation function without using the template, but be sure it satisfies the requirements described in "Requirements for Functions" on page 11-59.

Note: The template is not yet ready for use with BERTool. You must insert your own simulation code in the places marked `INSERT YOUR CODE HERE`. For a complete

example based on this template, see “Example: Prepare a Simulation Function for Use with BERTool” on page 11-63.

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

% Initialize variables related to exit criteria.
berVec = zeros(3,1); % Updated BER values

% --- Set up parameters. ---
% --- INSERT YOUR CODE HERE.
% Simulate until number of errors exceeds maxNumErrs
% or number of bits processed exceeds maxNumBits.
while((berVec(2) < maxNumErrs) && (berVec(3) < maxNumBits))

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

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

% Assign values to the output variables.
ber = berVec(1);
numBits = berVec(3);
```

Understanding the Template

From studying the code in the function template, observe how the function either satisfies the requirements listed in “Requirements for Functions” on page 11-59 or indicates where your own insertions of code should do so. In particular,

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

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

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

Using the Template

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

1. Determine the setup tasks you must perform. For example, you might want to initialize variables containing the modulation alphabet size, filter coefficients, a convolutional coding trellis, or the states of a convolutional interleaver. Place the code for these setup tasks in the template section marked `Set up parameters`.
2. Determine the core simulation tasks, assuming that all setup work has already been performed. For example, these tasks might include error-control coding, modulation/demodulation, and channel modeling. Place the code for these core simulation tasks in the template section marked `Proceed with simulation`.
3. Also in the template section marked `Proceed with simulation`, include code that updates the values of `totErr` and `numBits`. The quantity `totErr` represents the number of errors observed so far. The quantity `numBits` represents the number of bits processed so far. The computations to update these variables depend on how your core simulation tasks work.

   Note: Updating the numbers of errors and bits is important for ensuring that the loop terminates. However, if you accidentally create an infinite loop early in your development work using the function template, click **Stop** in BERTool to abort the simulation.
4. Omit any setup code that initializes `EbNo`, `maxNumErrs`, or `maxNumBits`, because BERTool passes these quantities to the function as input arguments after evaluating the data entered in the GUI.
5. Adjust your code or the template's code as necessary to use consistent variable names and meanings. For example, if your original code uses a variable called `ebn0` and the template's function declaration (first line) uses the variable name `EbNo`, you must change one of the names so they match. As another example, if your original code uses SNR instead of E_b/N_0, you must convert quantities appropriately.

Example: Prepare a Simulation Function for Use with BERTool

This section adapts the function template given in "Template for a Simulation Function" on page 11-60.

Preparing the Function

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

1. Copy the template from "Template for a Simulation Function" on page 11-60 into a new MATLAB file in the MATLAB Editor. Save it in a folder on your MATLAB path using the file name `bertool_simfcn`.
2. From the original example, the following lines are setup tasks. They are modified from the original example to rely on the input arguments that BERTool provides to the function, instead of defining variables such as `EbNovec` and `numerrmin` directly.

   ```
   % Set up initial parameters.
   siglen = 1000; % Number of bits in each trial
   M = 2; % DBPSK is binary.
   % DBPSK modulation and demodulation System objects
   hMod  = comm.DBPSKModulator;
   hDemod = comm.DBPSKDemodulator;
   % AWGNChannel System object
   hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');
   % ErrorRate calculator System object to compare decoded symbols to the
   % original transmitted symbols.
   hErrorCalc = comm.ErrorRate;
   snr = EbNo; % Because of binary modulation
   hChan.SNR = snr; %Assign Channel SNR
   ```

   Place these lines of code in the template section marked `Set up parameters.`
3. From the original example, the following lines are the core simulation tasks, after all setup work has been performed.

   ```
   msg = randi([0,M-1], siglen, 1); % Generate message sequence.
   txsig = step(hMod, msg); % Modulate.
   hChan.SignalPower = (txsig'*txsig)/length(txsig);  % Calculate and
                                                      % assign signal power
   rxsig = step(hChan,txsig); % Add noise.
   decodmsg = step(hDemod, rxsig); % Demodulate.
   berVec = step(hErrorCalc, msg, decodmsg); % Calculate BER
   ```

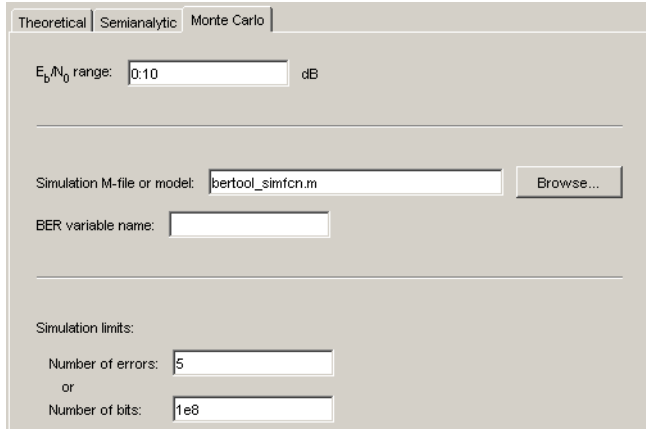
   Place the code for these core simulation tasks in the template section marked `Proceed with simulation`.

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

Using the Prepared Function

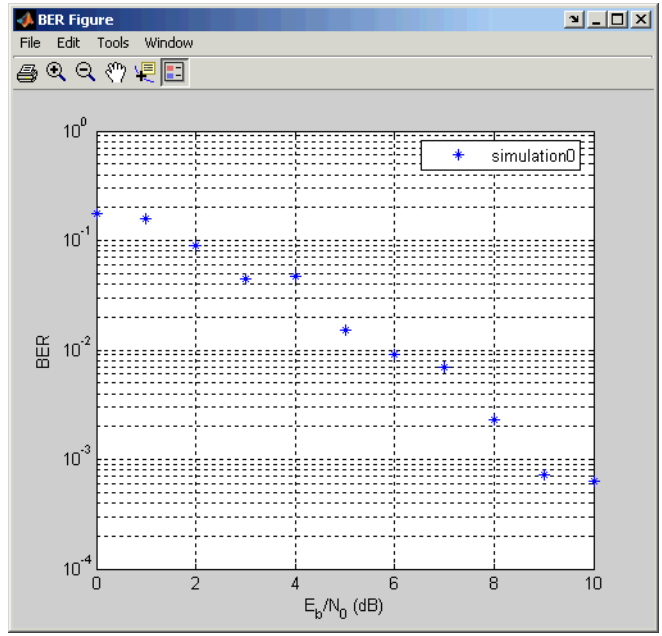
To use `bertool_simfcn` in conjunction with BERTool, continue the example by following these steps:

1. Open BERTool and go to the **Monte Carlo** tab.
2. Set parameters on the **Monte Carlo** tab as shown in the following figure.



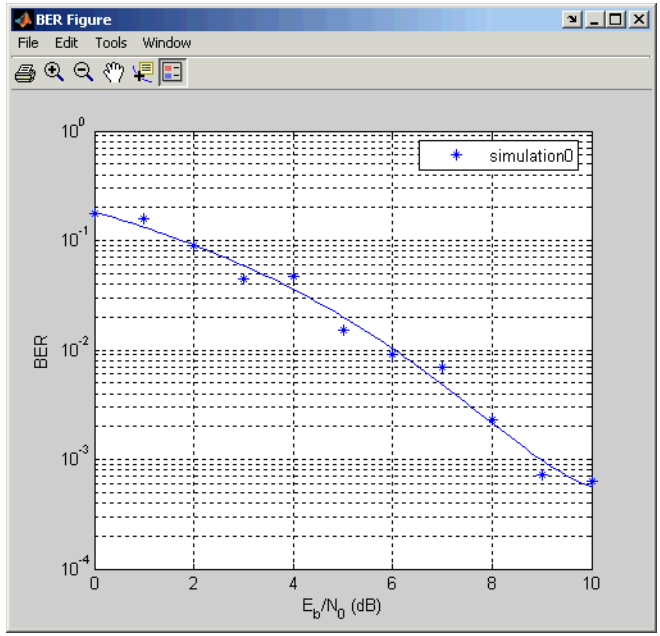
3. Click **Run**.

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



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

BERTool plots the curve, as shown in the following figure.



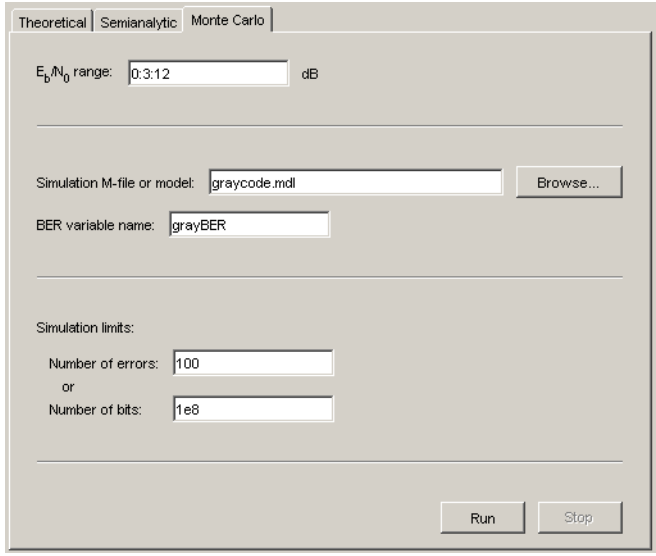
Run Simulink Simulations

Section Overview

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

Note: To use Simulink models within BERTool, you must have a Simulink license. Communications System Toolbox software is highly recommended. The rest of this section assumes you have a license for both Simulink and Communications System Toolbox applications.

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



For further details about confidence intervals and curve fitting for simulation data, see "Plotting Confidence Intervals" on page 11-57 and "Fitting BER Points to a Curve" on page 11-58, respectively.

Example: Using a Simulink Model with BERTool

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

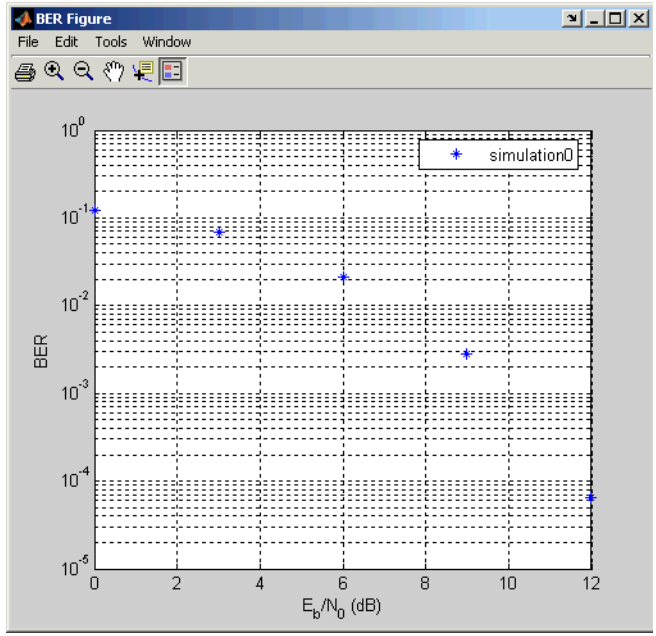
To run this example, follow these steps:

1. Open BERTool and go to the **Monte Carlo** tab. The model's file name, `commgraycode.mdl`, appears as the **Simulation M-file or model** parameter. (If `viterbisim.m` appears there, select to indicate that Communications System Toolbox software is installed.)
2. Click **Run**.

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

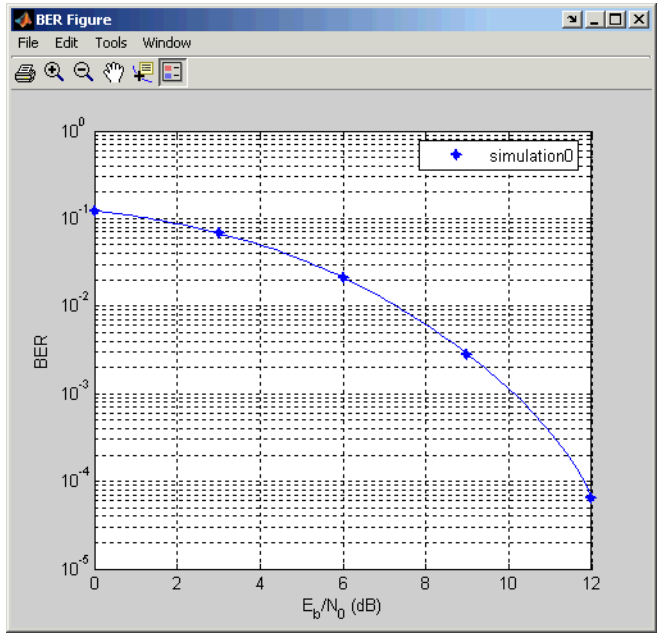
Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☐	☑	simulation0	0:3:12	[0.1233 0.0...	[900 1500 4...

BERTool plots the data in the BER Figure window.



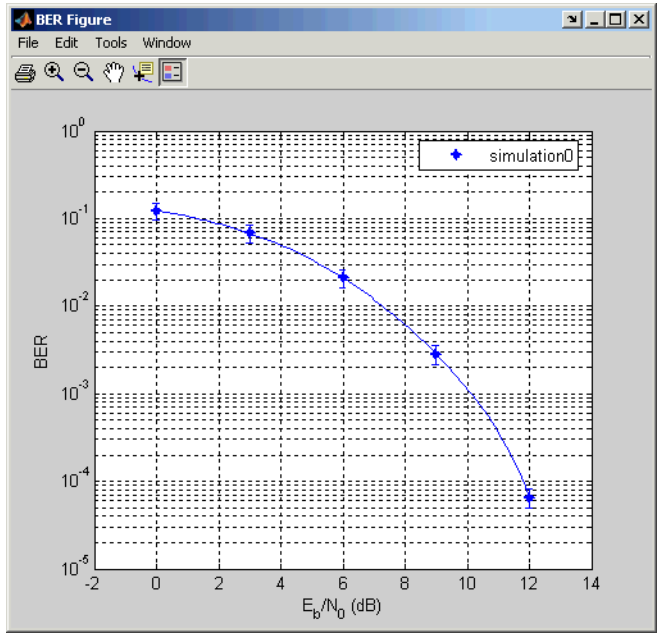
3 To fit a curve to the series of points in the BER Figure window, select the box next to **Fit** in the data viewer.

BERTool plots the curve, as below.



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

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

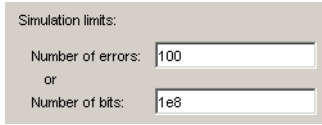


Another example that uses BERTool to manage a series of Simulink simulations is in "Example: Prepare a Model for Use with BERTool" on page 11-74.

Varying the Stopping Criteria

When you create a Simulink model for use with BERTool, you must set it up so that the simulation ends when it either detects a target number of errors or processes a maximum number of bits, whichever occurs first. To learn more about this requirement, see "Requirements for Models" on page 11-71; for an example, see "Example: Prepare a Model for Use with BERTool" on page 11-74.

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



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

You can also click **Stop** in BERTool to stop a series of simulations prematurely.

Use Simulink Models with BERTool

Requirements for Models

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

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

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

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

Tips for Preparing Models

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

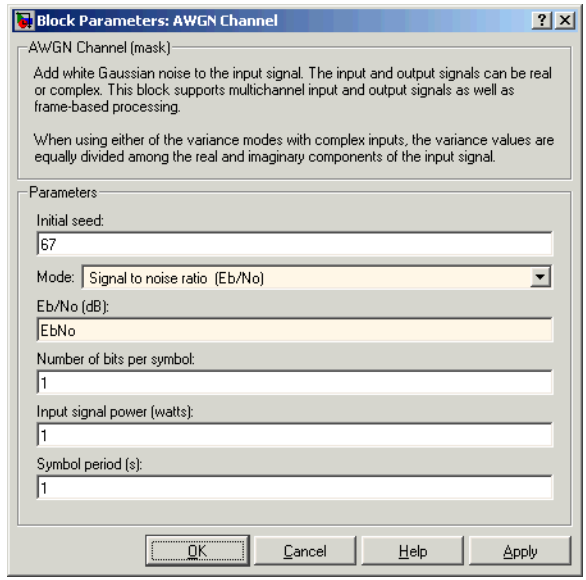
- To avoid using an undefined variable name in the dialog box for a Simulink block in the steps that follow, set up variables in the MATLAB workspace using a command such as the one below.

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

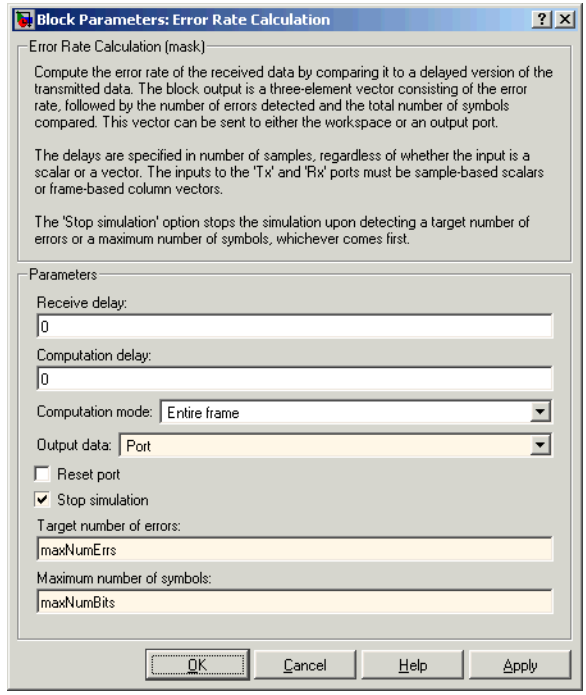
You might also want to put the same command in the model's preload function callback, to initialize the variables if you reopen the model in a future MATLAB session.

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

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



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



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

related to each other, press the AWGN Channel block's **Help** button to view the online reference page.

- If your model uses a preload function or other callback to initialize variables in the MATLAB workspace upon loading, make sure before you use the **Run** button in BERTool that one of these conditions is met:

  - The model is not currently in memory. In this case, BERTool loads the model into memory and runs the callback functions.
  - The model is in memory (whether in a window or not), and the variables are intact.

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

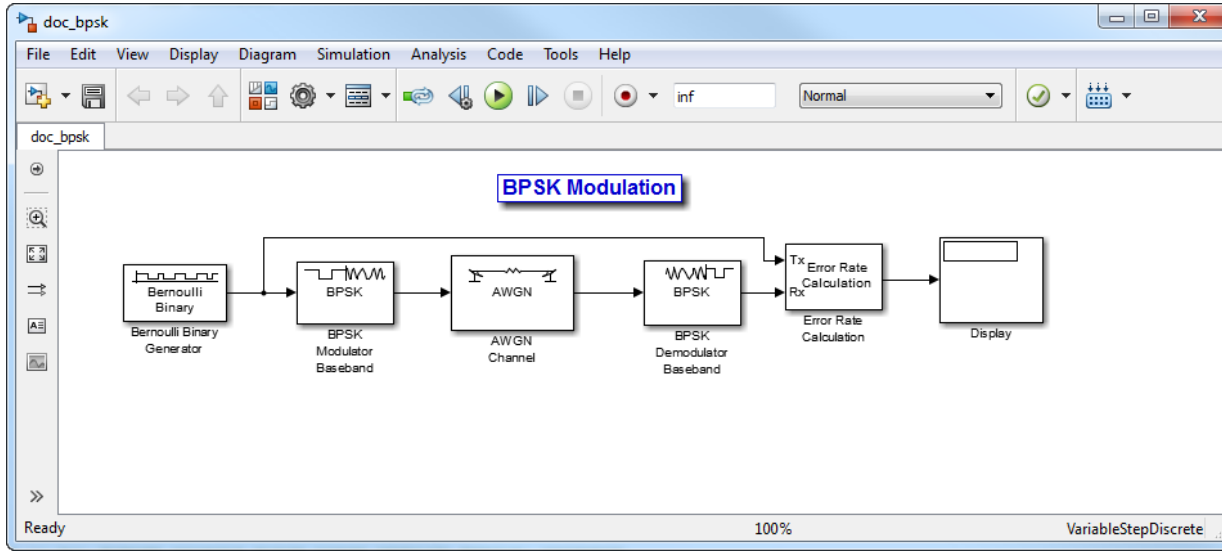
Example: Prepare a Model for Use with BERTool

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

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

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

   ```
   doc_bpsk
   ```



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

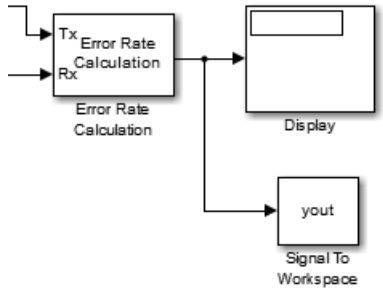
```
EbNo = 0; maxNumErrs = 100; maxNumBits = 1e8;
```

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

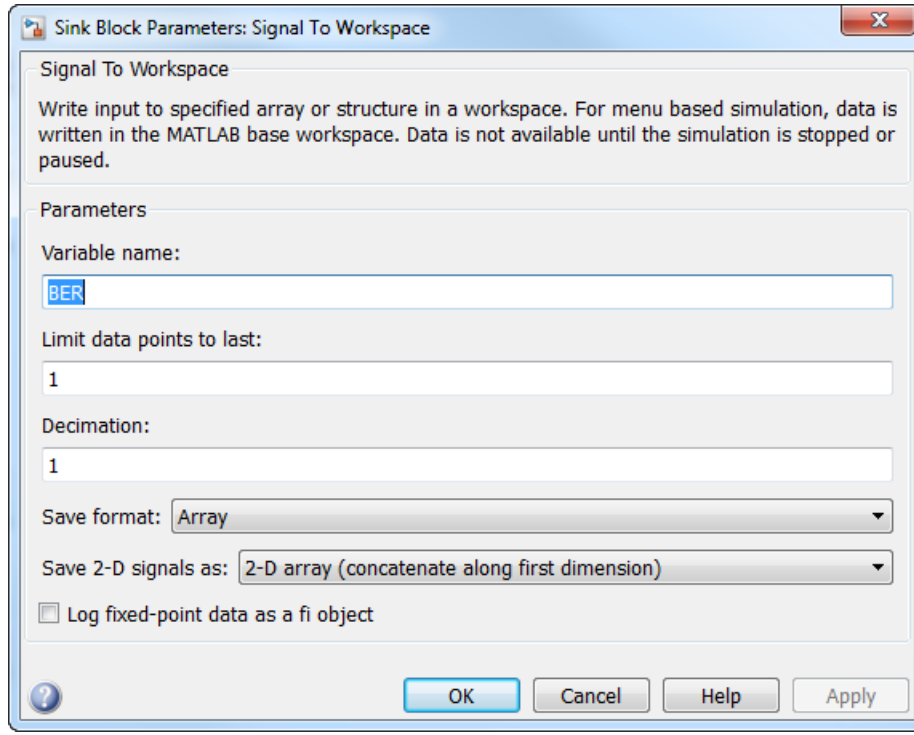
4 To ensure that BERTool uses the correct stopping criteria for each iteration, open the dialog box for the Error Rate Calculation block. Set **Target number of errors** to `maxNumErrs`, set **Maximum number of symbols** to `maxNumBits`, and click **OK**.

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

Note: The Signal to Workspace block is in DSP System Toolbox software and is different from the To Workspace block in Simulink.



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

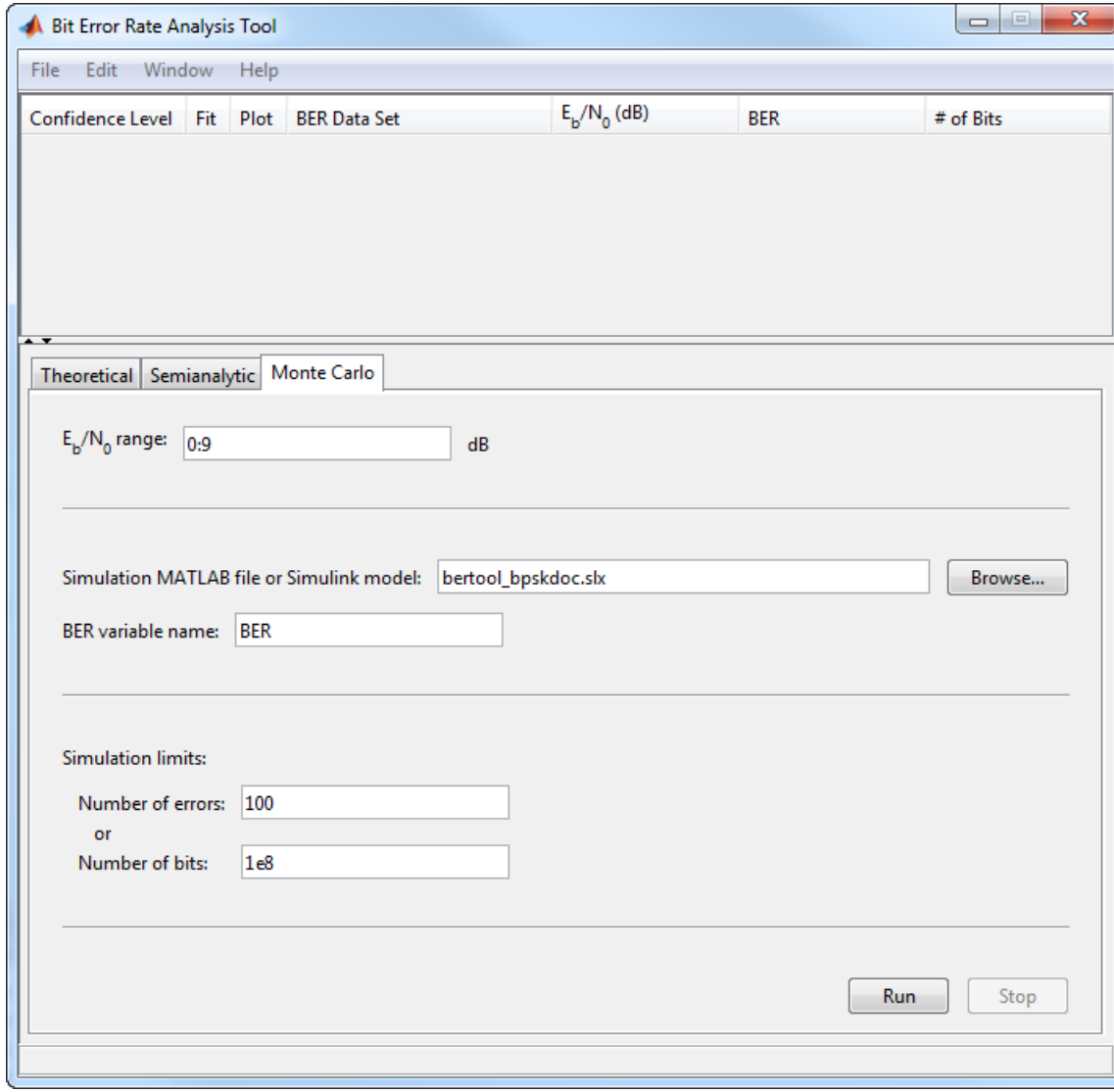


7. (*Optional*) To make the simulation run faster, especially at high values of E_b/N_0, open the dialog box for the Bernoulli Binary Generator block. Select **Frame-based outputs** and set **Samples per frame** to `1000`.
8. Save the model in a folder on your MATLAB path using the file name `bertool_bpskdoc.slx`.
9. (*Optional*) To cause Simulink to initialize parameters if you reopen this model in a future MATLAB session, enter the following command in the MATLAB Command Window and resave the model.

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

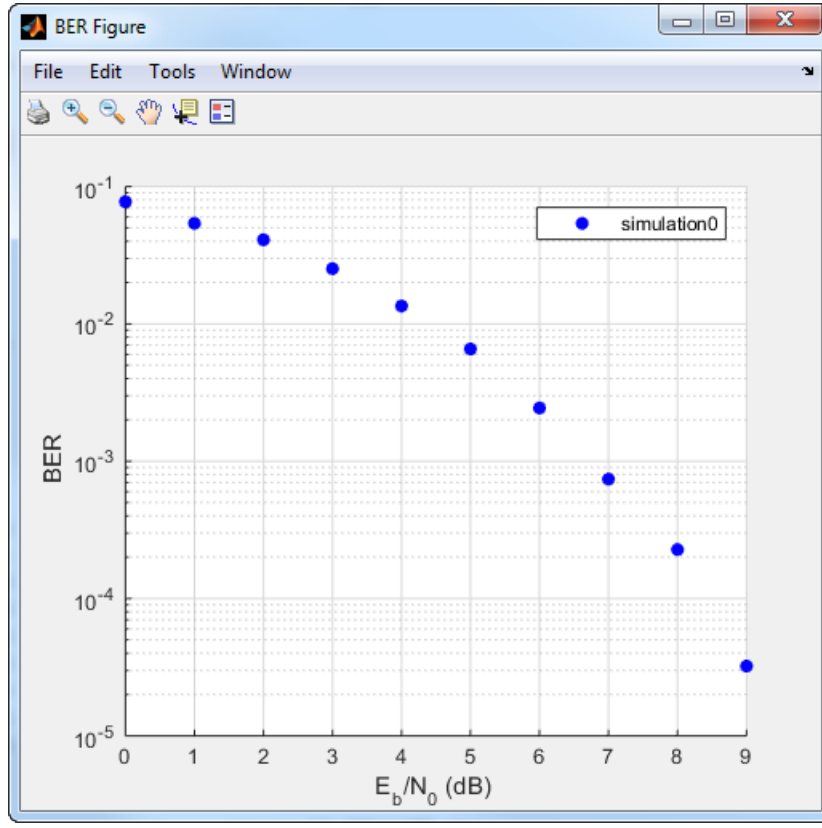
The `bertool_bpskdoc` model is now compatible with BERTool. To use it in conjunction with BERTool, continue the example by following these steps:

1. Open BERTool and go to the **Monte Carlo** tab.
2. Set parameters on the **Monte Carlo** tab as shown in the following figure.

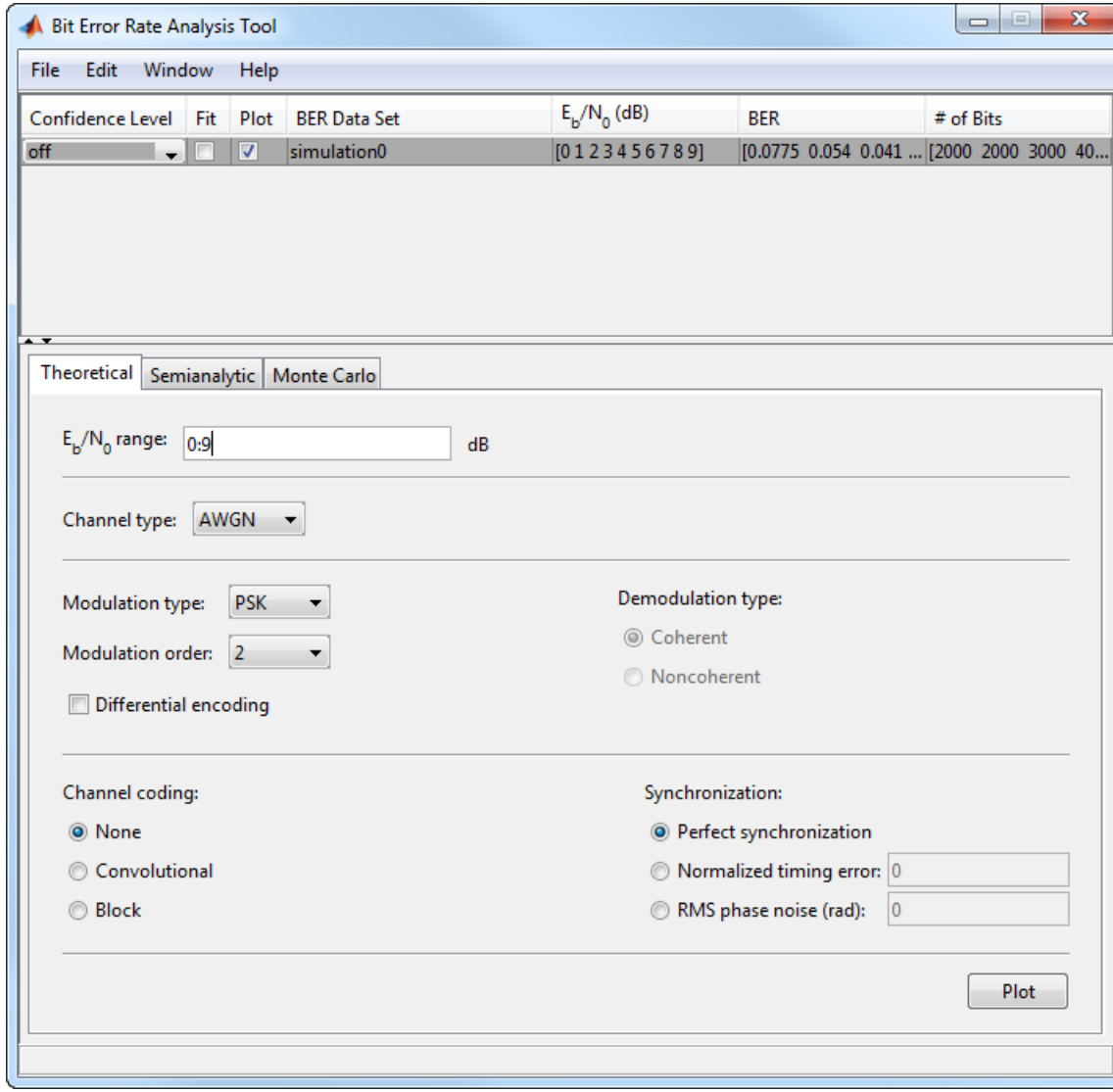


3 Click **Run**.

BERTool spends some time computing results and then plots them.

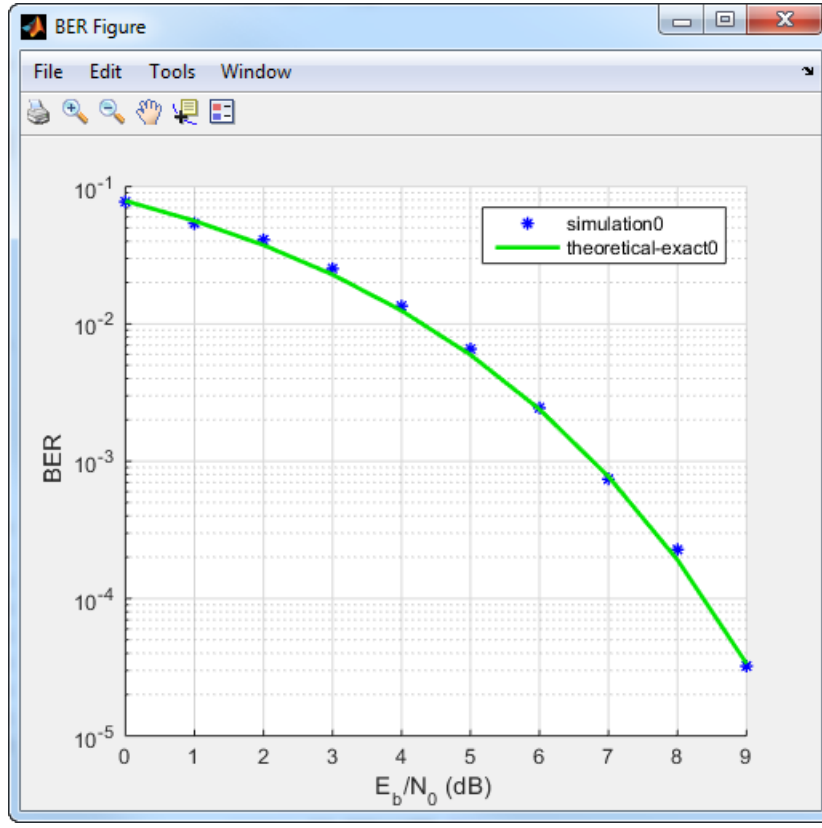


4 To compare these simulation results with theoretical results, go to the **Theoretical** tab in BERTool and set parameters as shown below.



5 Click **Plot**.

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



Manage BER Data

Exporting Data Sets or BERTool Sessions

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

commands in MATLAB. Another option for exporting enables you to reimport the data into BERTool later.

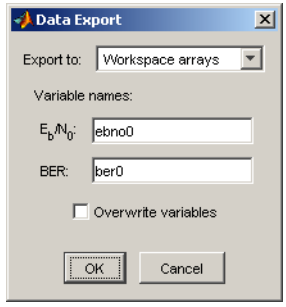
BERTool also enables you to save an entire session, which is useful if your session contains multiple data sets that you want to return to in a later session.

This section describes these capabilities:

Exporting Data Sets

To export an individual data set, follow these steps:

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



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

   a. If you want to reimport the data into BERTool later, you *must* choose either `Workspace structure` or `MAT-file structure` to create a structure in the MATLAB workspace or a MAT-file, respectively.

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

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

   b. If you do *not* need to reimport the data into BERTool later, a convenient way to access the data outside BERTool is to have BERTool create a pair of arrays in the MATLAB workspace. One array contains E_b/N_0 values, while the other array contains BER values. To choose this option, set **Export to** to `Workspace arrays`.

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

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

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

To reimport a structure later, see "Importing Data Sets" on page 11-85.

Examining an Exported Structure

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

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

Parameter Fields

The `params` and `paramsEvaled` fields are similar to each other, except that `params` describes the exact state of the GUI whereas `paramsEvaled` indicates the values that are actually used for computations. As an example of the difference, for a theoretical system with an AWGN channel, `params` records but `paramsEvaled` omits a diversity order parameter. The diversity order is not used in the computations because it is relevant only for systems with Rayleigh channels. As another example, if you type

`[0:3]+1` in the GUI as the range of E_b/N_0 values, `params` indicates `[0:3]+1` while `paramsEvaled` indicates `1 2 3 4`.

The length and exact contents of `paramsEvaled` depend on the data set because only relevant information appears. If the meaning of the contents of `paramsEvaled` is not clear upon inspection, one way to learn more is to reimport the data set into BERTool and inspect the parameter values that appear in the GUI. To reimport the structure, follow the instructions in "Importing Data Sets or BERTool Sessions" on page 11-84.

Data Field

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

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

Saving a BERTool Session

To save an entire BERTool session, follow these steps:

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

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

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

Importing Data Sets or BERTool Sessions

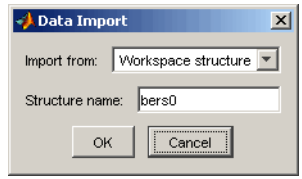
BERTool enables you to reimport individual data sets that you previously exported to a structure, or to reload entire sessions that you previously saved. This section describes these capabilities:

To learn more about exporting data sets or saving sessions from BERTool, see "Exporting Data Sets or BERTool Sessions" on page 11-81.

Importing Data Sets

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

1. Choose **File > Import Data**.



2. Set **Import from** to either `Workspace structure` or `MAT-file structure`. If you select `Workspace structure`, type the name of the workspace variable in the **Structure name** field.
3. Click **OK**. If you select `MAT-file`, BERTool prompts you to select the file that contains the structure you want to import.

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

Opening a Previous BERTool Session

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

1. Choose **File > Open Session**.

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

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

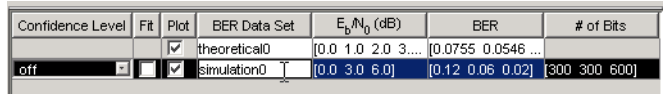
After BERTool reads the session file, the data viewer shows the data sets from the file.

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

Managing Data in the Data Viewer

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

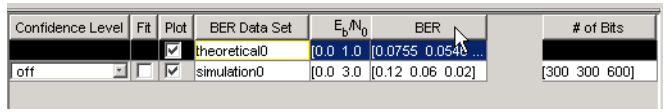
- To rename a data set in the data viewer, double-click its name in the **BER Data Set** column and type a new name.



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

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

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



Error Rate Test Console

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

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

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

You obtain error rate results at different locations in the system under test, by defining unique test points. Each test point contains a pair of probes that the system uses to log data to the test console. The information you register with the test console specifies how

each pair of test probes compares data. For example, in a frame based system, the Error Rate Test Console can compare transmitted and received header bits or transmitted and received data bits. Similarly, it can compare CRC error counts to obtain frame error rates at different points in the system. You can also configure the Error Rate Test Console to compare data in multiple pairs of probes, obtaining multiple error rate results.

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

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

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

To see a full-scale example on creating a system and running simulations, see "Bit Error Rate Simulations For Various Eb/No and Modulation Order Values".

The following sections describe the Error Rate Test Console and its functionality:

Creating a System

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

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

- Writing a system class, extending the testconsole.SystemBasicAPI class.

- Writing a registration method
  - Registration is test related
  - Defines items such as test parameters, test probes, and test inputs
- Writing a setup method
- Writing a reset method
- Writing a run method

  Methods allows the system to communicate with the test console.

To see the system file, navigate to the following location:

matlab\toolbox\comm\comm\+commtest

Then, enter the following syntax at the MATLAB command line:

```
edit MPSKSYSTEM.m
```

Writing A Register Method

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

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

Registering Test Inputs

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

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

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

- `sys` represents the handle to a user-defined system object.
- `inputName` represents the name of the input that the system registers. This name must coincide with the name of an available test input in the Error Rate Test Console or an error occurs.
  - 'NumTransmissions' - calling the `getInput` method returns the frame length. The system itself is responsible for generating a data frame using a data source.
  - 'RandomIntegerSource' - calling the `getInput` method returns a vector of symbols with a length the Error Rate Test Console `FrameLength` property specifies. If the system registers this source type, then it must also register a test parameter named 'M' that corresponds to the modulation order.

Registering Test Parameters

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

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

- `sys` represents the handle to the user-defined system object
- `name` represents the parameter name that the system registers to the Error Rate Test Console
- `default` specifies the default value of the test parameter – it can be a numeric value or a string
- `validRange` specifies a range of input values for the test parameter — it can be a 1x2 vector of numeric values with upper and lower ranges or a cell array of chars (an Enum).

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

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

Registering Test Probes

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

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

- `sys` represents the handle to the user-defined system object
- `name` represents the name of the test probe
- `description` contains information about the test probes; useful for indicating what the probe is used for. The description input is optional.

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

Writing a Setup Method

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

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

Writing a Reset Method

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

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

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

Writing a Run Method

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

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

- requests test inputs from the test console using the `getInput` method
- logs test data to its test probes using the `setTestProbeData` method
- logs user-data to the test console using the `setUserData` method
- Although it is recommended you do this at setup time, the system can also request the current simulation sweep values using the `getTestParameter` method.

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

- "Getting Test Inputs From the Error Rate Test Console" on page 11-91
- "Getting the Current Simulation Sweep Value of a Registered Test Parameter" on page 11-92
- "Logging Test Data to a Registered Test Probe" on page 11-92
- "Logging User-Defined Data To The Test Console" on page 11-92

Getting Test Inputs From the Error Rate Test Console

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

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

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

The system under test provides the following inputs:

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

Getting the Current Simulation Sweep Value of a Registered Test Parameter

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

Logging Test Data to a Registered Test Probe

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

- `sys` represents the handle to the system
- `name` represents the name of a registered test probe
- `data` represents the data the probe logs to the Error Rate Test Console.

Logging User-Defined Data To The Test Console

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

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

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

Debug Mode

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

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

Implementing A Default Input Generator Function For Debug Mode

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

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

Run Simulations Using the Error Rate Test Console

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

- Creating a test console
- Attaching a system
- Defining simulation conditions
  - Specifying stop criterion
  - Specifying iteration mode
  - Specifying reset mode
  - Specifying sweep values
- Registering test points
- Running simulations

- Getting results and plotting

Creating a Test Console

You create a test console in one of the following ways:

- `h = commtest.ErrorRate` returns an error rate test console, h. The error rate test console runs simulations of a system under test to obtain error rates.
- `h = commtest.ErrorRate(sys)` returns an error rate test console, h, with an attached system under test, sys.
- `h = commtest.ErrorRate(sys,'PropertyName',PropertyValue,...)` returns an error rate test console, h, with an attached system under test, sys. Each specified property, 'PropertyName', is set to the specified value, PropertyValue.
- `h = commtest.ErrorRate('PropertyName',PropertyValue,...)` returns an error rate test console, h, with each specified property 'PropertyName', set to the specified value, PropertyValue.

Attaching a System to the Error Rate Test Console

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

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

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

- To attach a system at construction time of an Error Rate Test Console, see Creating a Test Console.
- `mySystem` is the name of the system under test

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

Defining Simulation Conditions

Stop Criterion

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

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

Iteration Mode

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

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

Specifying and Obtaining Sweep Values

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

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

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

where

- `obj` represents handle to the Error Rate Test Console.
- `name` represents the name of the registered test parameter (this name must correspond to a test parameter registered by the system under test or an error occurs)
- `value` represents the sweep values you specify for the test parameter named 'name'. Depending on the application, sweep values may be a vector with numeric values or a cell array of characters. The test console issues an error if you attempt to set sweep values that are out of the specified valid range for a test parameter (valid ranges are defined by the system when attaching to a test console).

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

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

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

Reset Mode

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

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

Registering a Test Point

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

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

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

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

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

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

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

Writing a user-defined error calculator function

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

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

Getting Test Information

Returns a report of the current test console settings.

info(h) displays:

- Test console name
- System under test name

- Available test inputs
- Registered test inputs
- Registered test parameters
- Registered test probes
- Registered test points
- Metric calculator functions
- Test metrics

Running a Simulation

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

`run(testConsole)` runs a specified number of iterations of an attached system under test for a specified set of parameter values. If a Parallel Computing Toolbox™ license is available and a parpool is open, then you can distribute the iterations among the available number of workers.

Getting Results and Plotting Data

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

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

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

Parsing and Plotting Results for Multiple Parameter Simulations

The DPSKModulationTester.mat file contains an Error Rate Test Console with a DPSK modulation system. This system defines three test parameters:

- The bit energy to noise power spectral density ratio, `EbNo` (in decibels)
- The modulation order, `M`
- The maximum Doppler shift, `MaxDopplerShift` (in hertz)

These parameters have the following sweep values:

- `EbNo` = [-2:4] dB
- `M` = [2 4 8 16]

- `MaxDopplerShift` = [0 0.001 0.09] Hz

Because simulations generally take a long time to run, a simulation was run offline. DPSKModulationTester.mat file contains a saved Error Rate Test Console with the saved results. The simulations were run to obtain at least 2500 errors and 5e6 frame transmissions per simulation point.

Load the simulation results by entering the following at the MATLAB command line:

```
load DPSKModulationTester.mat
```

To parse and plot results for multiple parameter simulations, perform the following steps:

1. Using the `getSweepParameterValues` method, display the sweep parameter values used in the simulation for each test parameter. For example, you display the sweep values for `MaxDopplerShift` by entering:

   ```
   getTestParameterSweepValues(testConsole,'MaxDopplerShift')
   ```

   MATLAB returns the following result:

   ```
   ans =

            0    0.0010    0.0900
   ```

2. Get the results object that parses and plots simulation results by entering the following at the command line:

   ```
   DPSKResults = getResults(testConsole)
   ```

   MATLAB returns the following result:

   ```
   DPSKResults =

          TestConsoleName: 'commtest.ErrorRate'
      SystemUnderTestName: 'commexample.DPSKModulation'
            IterationMode: 'Combinatorial'
                TestPoint: 'BitErrors'
                   Metric: 'ErrorRate'
           TestParameter1: 'EbNo'
           TestParameter2: 'None'
   ```

3. Use the `setParsingValues` method to enable the plotting of error rate results versus Eb/No for a modulation order of 4 and maximum Doppler shift of 0.001 Hz. To do so, enter the following:.

```
setParsingValues(DPSKResults,'M',4,'MaxDopplerShift',0.001)
```

4 Use the `getParsingValues` method to verify the current parsing values settings:

```
getParsingValues(DPSKResults)
```

MATLAB returns the following:

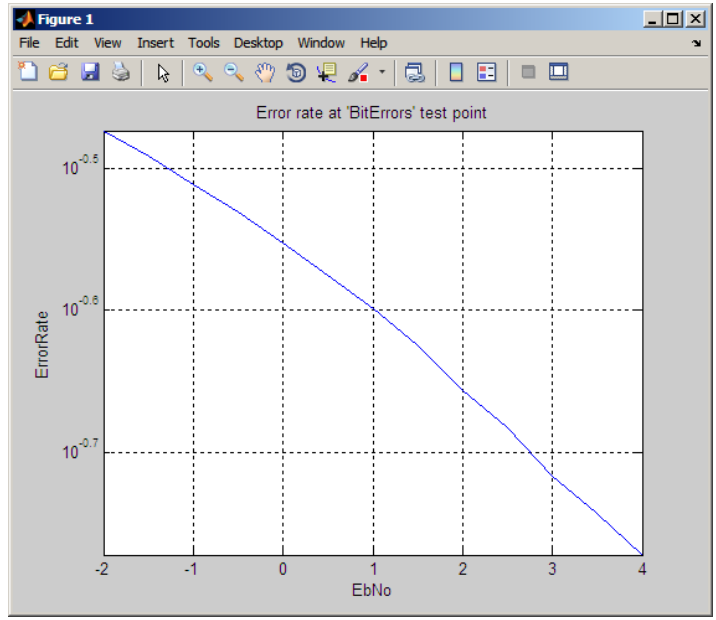
```
ans =

               EbNo: -2
                  M: 4
    MaxDopplerShift: 1.0000e-003
```

If not specified, the parsing value for a test parameter defaults to its first sweep value. In this example, the first sweep value for EbNo equals -2 dB. However, in this example, `TestParameter1` is set to EbNo; therefore, the Error Rate Test Console plots results for all EbNo sweep values, not just for the value listed by the `getParsingValues` method.

5 Obtain a log-scale plot of bit error rate versus Eb/No for a modulation order of 4 and a maximum Doppler shift of 0.001 Hz:

```
semilogy(DPSKResults)
```

MATLAB generates the following figure.



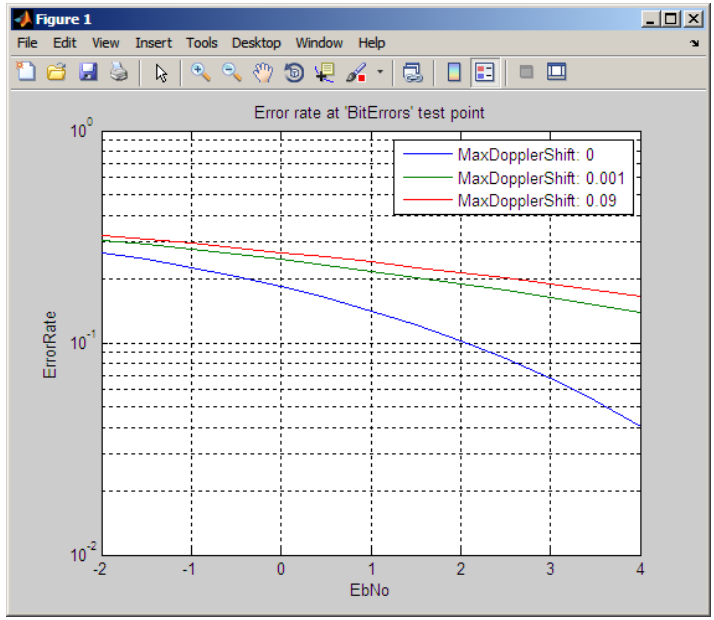
6 Set the `TestParameter2` property of the results object to 'MaxDopplerShift'. This setting enables the plotting of multiple error rate curves versus Eb/No for each sweep value of the maximum Doppler shift.

```
DPSKResults.TestParameter2 = 'MaxDopplerShift';
```

7 Obtain log-scale plots of bit error rate versus Eb/No for a modulation order of 2 at each of the maximum Doppler shift sweep values.

```
setParsingValues(DPSKResults,'M',2)
semilogy(DPSKResults)
```

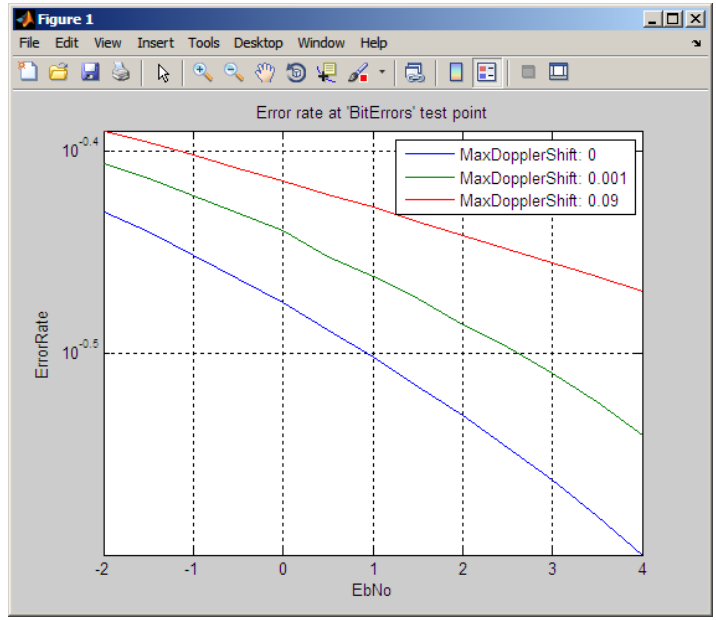
MATLAB generates the following figure.



8 Obtain the same type of curves as in the previous step, but now for a modulation order of 16.

```
setParsingValues(DPSKResults,'M',16)
semilogy(DPSKResults)
```

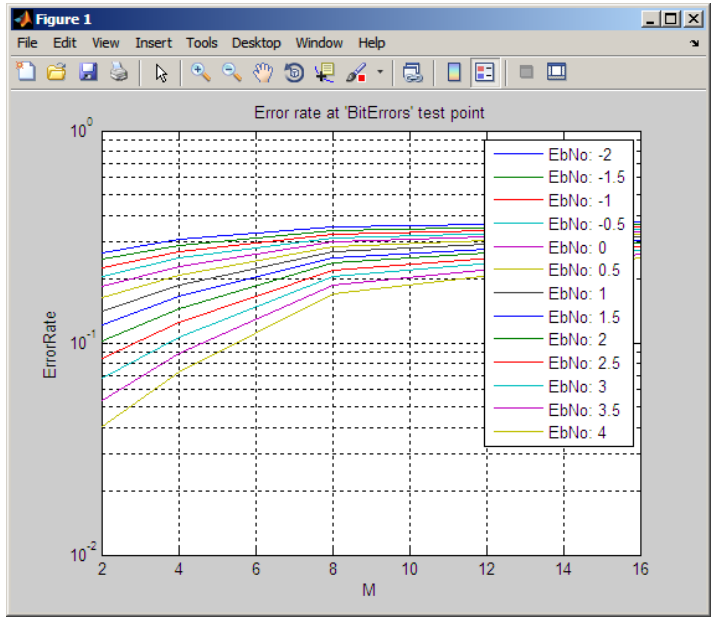
MATLAB generates the following figure.



9 Obtain error rate plots versus the modulation order for each Eb/No sweep value by setting `TestParameter1` equal to M and `TestParameter2` equal to EbNo. You can plot the results for the case when the maximum Doppler shift is 0 Hz by using the `setParsingValues` method:

```
DPSKResults.TestParameter1 = 'M';
DPSKResults.TestParameter2 = 'EbNo';
setParsingValues(DPSKResults, 'MaxDopplerShift',0)
semilogy(DPSKResults)
```

MATLAB generates the following figure.



10 Obtain a data matrix with the bit error rate values previously plotted by entering the following:

```
BERMatrix = getData(DPSKResults)
```

MATLAB returns the following result:

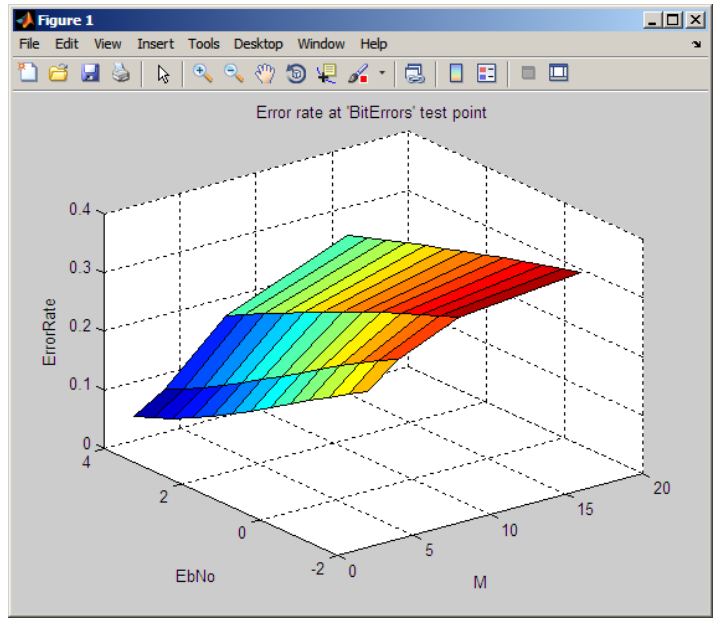
```
BERMatrix =

  Columns 1 through 7

    0.2660    0.2467    0.2258    0.2049    0.1837    0.1628    0.1418
    0.3076    0.2889    0.2702    0.2504    0.2296    0.2082    0.1871
    0.3510    0.3384    0.3258    0.3120    0.2983    0.2837    0.2685
    0.3715    0.3631    0.3535    0.3442    0.3350    0.3246    0.3147

  Columns 8 through 13

    0.1217    0.1022    0.0844    0.0677    0.0534    0.0406
    0.1658    0.1451    0.1254    0.1065    0.0890    0.0728
    0.2531    0.2369    0.2204    0.2042    0.1874    0.1704
    0.3044    0.2945    0.2839    0.2735    0.2626    0.2512
```

The rows of the matrix correspond to the values of the test parameter defined by the `TestParameter1` property, M. The columns correspond to the values of the test parameter defined by the `TestParameter2` property, EbNo.

11 Plot the results as a 3-D data plot by entering the following:

```
surf(DPSKResults)
```

MATLAB generates the following plot:



In this case, the parameter defined by the `TestParameter1` property, M, controls the x-axis and the parameter defined by the `TestParameter2` property, EbNo, controls the y-axis.

Bit Error Rate Simulations For Various Eb/No and Modulation Order Values

Tasks for running bit error rate simulations for various En/No and modulation order values.

- "Load the Error Rate Test Console" on page 11-106
- "Run the Simulation and Obtain Results" on page 11-107

Load the Error Rate Test Console

The Error Rate Test Console is a simulation tool for obtaining error rate results. The MATLAB software includes a data file for use with the Error Rate Test Console. You will use the data file while performing the steps of this tutorial. The data file contains an Error Rate Test Console object with an attached Gray coded modulation system. This example Error Rate Test Console is configured to run bit error rate simulations for various EbNo and modulation order, or M, values.

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

   ```
   load GrayCodedModTester_EbNo_M
   ```

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

   ```
   testConsole
   ```

   MATLAB returns the following output:

   ```
   testConsole =

                       Description: 'Error Rate Test Console'
             SystemUnderTestName: 'commexample.GrayCodedMod_EbNo_M'
                   IterationMode: 'Combinatorial'
                 SystemResetMode: 'Reset at new simulation point'
           SimulationLimitOption: 'Number of errors or transmissions'
       TransmissionCountTestPoint: 'DemodBitErrors'
              MaxNumTransmissions: 100000000
              ErrorCountTestPoint: 'DemodBitErrors'
                     MinNumErrors: 100
   ```

   Notice that SystemUnderTest is a Gray coded modulation system. Because the SimulationLimitOption is 'Number of error or transmission', the simulation runs until reaching 100 errors or 1e8 bits.

Run the Simulation and Obtain Results

In this example, you use `tic` and `toc` to compare simulation run time.

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

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

   MATLAB returns output similar to the following:

   ```
   Running simulations...
   Elapsed time is 174.671632 seconds.
   ```

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

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

   MATLAB returns the following output:

   ```
   grayResults =

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

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

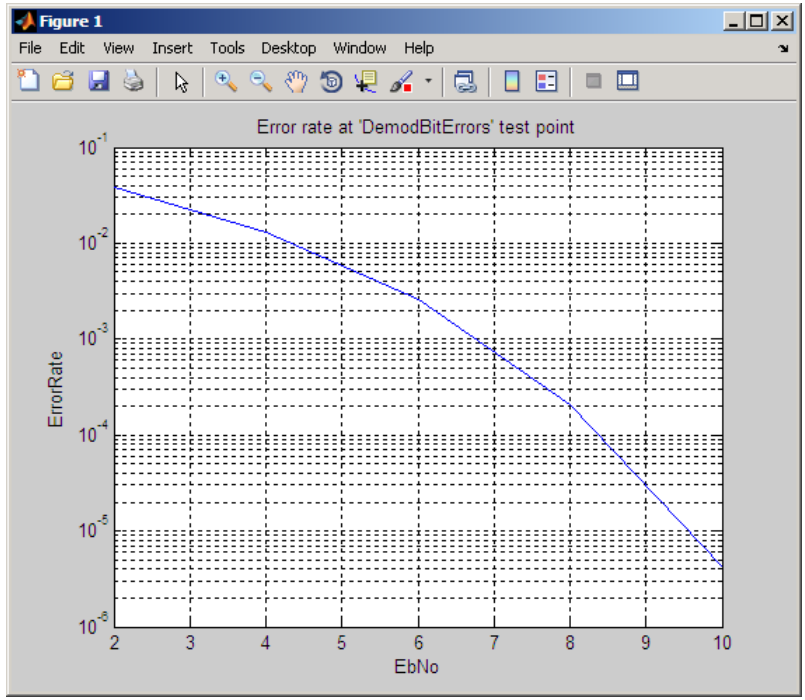
Generate an Error Rate Results Figure Window

The `semilogy` method generates a figure containing error rate curves for the demodulator bit error test point (DemodBitErrors) of the Gray coded modulation system. The next figure shows an Error Rate and E_b over N_o curve for the demodulator bit errors test point. This test point collects bit errors by comparing the bits the system transmits with the bits it receives. The x-axis displays the `TestParameter1` property of `grayResults`, which contains EbNo values.

1. Generate the figure by entering the following at the MATLAB command line:

   ```
   semilogy(grayResults)
   ```

This script generates the following figure.



2 Set the `TestParameter2` property to `M`. At the MATLAB command line, enter:

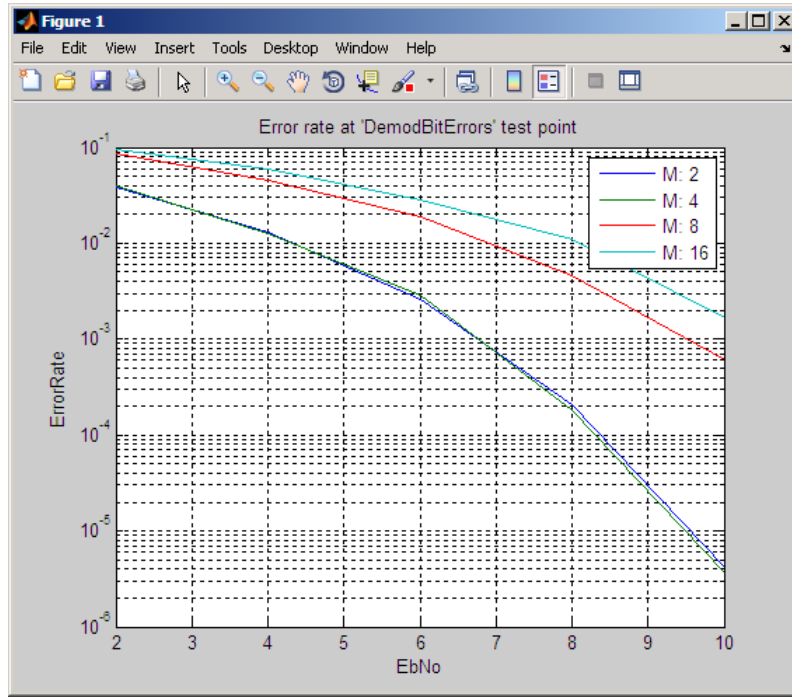
```
grayResults.TestParameter2 = 'M'
```

Previously, the simulation ran for multiple modulation order (M) values. The x-axis displays the `TestParameter1` property of `grayResults`, which contains EbNo values. Although the simulation ran for multiple M values, this run contains data for M=2.

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

```
semilogy(grayResults)
```

This script generates the following figure.



Run Parallel Simulations Using Parallel Computing Toolbox Software

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

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

---

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

   ```
   mypool = parpool()
   ```

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

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

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

MATLAB returns output similar to the following:

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

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

Create a System File and Attach It to the Test Console

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

This example outlines the tasks necessary for converting legacy code to a system file you can attach to the Error Rate Test Console. Use commdoc_gray as the starting point for your system file. The files you use in this section of the tutorial reside in the following folder:

matlab\help\toolbox\comm\examples

1. Copy the system basic API template, SystemBasicTemplate.m, as MyGrayCodedModulation.m.
2. Rename the references to the system name in the file. First, rename the system definition by changing the class name to MyGrayCodedModulation. Replace the following lines, lines 1 and 2, of the file:

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

   with these lines:

   ```
   classdef MyGrayCodedModulation < testconsole.SystemBasicAPI
   %MyGrayCodedModulation Gray coded modulation system
   ```
3. Rename the constructor by replacing:

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

   ```
   function obj = MyGrayCodedModulation
   ```

```
%MyGrayCodedModulation Construct a Gray coded modulation system
```

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

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

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

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

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

7 Change EbNo to a test parameter. This change allows the system to obtain EbNo values from the Error Rate Test Console. As a test parameter, EbNo becomes a variable, which allows simulations to run for different values. Locate the following line of syntax in the file:

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

Replace it with:

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

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

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

Replace it with:

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

9 Register the test parameters to the test console.

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

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

  The parameter defaults to 0 dB and can take values between -50 dB and 50 dB.
- Declare M as a test parameter by placing the following line of code in the body of the `register` method:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12 Register the test probes to the Error Rate Test Console, making it possible to obtain data from the system. Add these probes to the function `register(obj)` by adding two lines to the `register` method:

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

```
            registerTestParameter(obj,'EbNo',0,[-50 50]);
        registerTestParameter(obj,'M',16,[2 1024]);
            registerTestProbe(obj,'TxBits')
            registerTestProbe(obj,'RxBits')
        end
```

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

14 Create a Gray coded modulation system. At the MATLAB command line, enter:

```
mySystem = MyGrayCodedModulation
```

MATLAB returns the following output:

```
mySystem =

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

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

```
testConsole = commtest.ErrorRate
```

The MATLAB software returns the following output:

```
testConsole =

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

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

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

Configure the Error Rate Test Console and Run a Simulation

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

1. At the MATLAB command line, enter:

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

   This line defines the test point, DemodBitErrors, and compares bits from the TxBits probe to the bits from the RxBits probe. The Error Rate Test Console calculated metrics for this test point.

2. Configure the Error Rate Test Console to run simulations for EbNo values. Start at 2 dB and end at 10 dB, with a step size of 2 dB and M values of 2, 4, 8, and 16. At the MATLAB command line, enter:

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

3. Set the simulation limit to the number of transmissions.

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

4. Set the maximum number of transmissions to 1000.

   ```
   testConsole.MaxNumTransmissions = 1000
   ```

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

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

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

   ```
   run(testConsole)
   ```

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

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

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

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

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

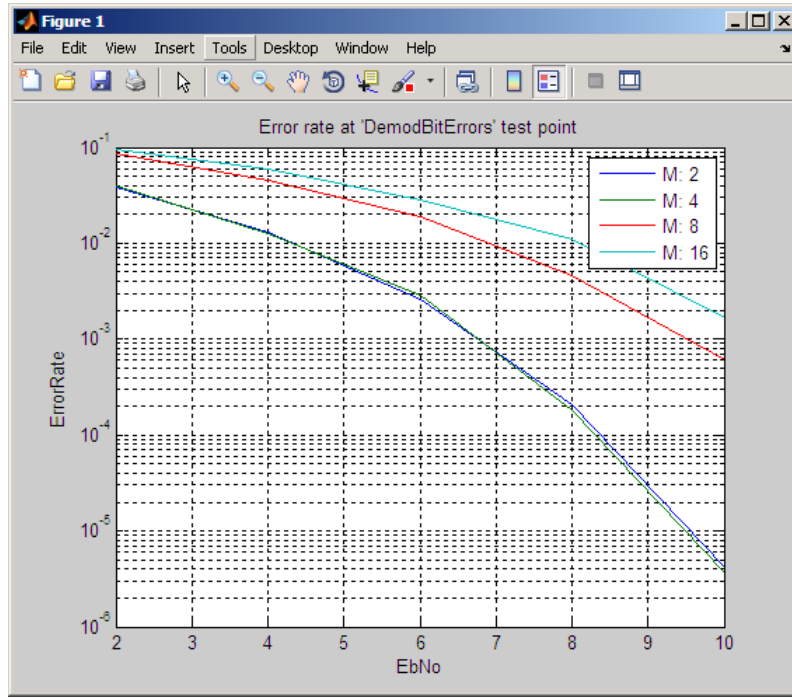
9 Run the simulation by entering the following at the MATLAB command line.

```
run(testConsole);
```

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

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

This script generates the following figure.



Optimize System Performance Using Parameterized Simulations

In the previous example, the system only utilizes the `run` method. Every time the object calls the `run` method, which is every 3e4 bits for this simulation, the object sets the M

and SNR values. This time interval includes: obtaining numbers from the test console, calculating intermediate values, and setting other variables.

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

The `run` method of a system also creates a new modulator (hMod) and a new demodulator (hDemod). Creating a modulator or a demodulator is much more time consuming than just modifying a property of these objects. Create a modulator and a demodulator object once when the system is constructed. Then, modify its properties in the setup method of the system to speed up the simulations.

1. Save the file MyGrayCodedModulation as MyGrayCodedModulationOptimized.
2. In the MyGrayCodedModulationOptimized file, replace the constructor name and the class definition name.

   - Locate the following lines of code:

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

   - Replace them with:

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

3. In the MyGrayCodedModulationOptimized file, replace the constructor name.

   - Locate the following lines of code:

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

   - Replace them with:

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

4. Move the oversampling rate definition from the `run` method to the `setup` method.

   ```
   nSamp = 1;                    % Oversampling rate
   ```

5. Move code related to setting M to the `setup` method. Cut the following lines from the `run` method and paste to the `setup` method.

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

6 In the `setup` method, replace M with the object property M.

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

This change provides access to the M value from the `run` method.

7 Move code related to setting EbNo to the `setup` method. Cut the following lines from the `run` method and paste to the `setup` method.

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

SNR = EbNo + 10*log10(k) - 10*log10(nSamp);
```

8 In the `setup` method, replace EbNo with the object property EbNo. This change provides access to the EbNo value from the run method.

```
obj.EbNo = getTestParameter(obj,'EbNo');
SNR = obj.EbNo + 10*log10(k) - 10*log10(nSamp);
```

9 Create a new internal variable called SNR to store the calculated SNR value. Define the SNR property as a private property; it is not a test parameter. With this change, the system calculates SNR in the `setup` method and accesses it from the `run` method. Add the following lines of code the system file, after the Test Parameters block.

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

10 In the `setup` method, replace SNR with object property SNR.

```
 obj.SNR = obj.EbNo + 10*log10(k) - 10*log10(nSamp);
```

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

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

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

12 Move the QAM modulator and demodulator creation out of the `run` method. Move following lines from the `run` method to the constructor (i.e the method named MyGrayCodedModulationOptimized)

```
%% Create Modulator and Demodulator
hMod = comm.RectangularQAMModulator(obj.M);     % Create a 16-QAM modulator
hMod.BitInput = true;                           % Accept bits as inputs
hMod.SymbolMapping = 'Gray';                    % Gray coded symbol mapping
hDemod = comm.RectangularQAMDemodulator(obj.M); % Create a 16-QAM demodulator
hDemod.BitOutput = true;                        % Output bits
hDemod.SymbolMapping = 'Gray';                  % Gray coded symbol mapping
```

Create private properties called Modulator and Demodulator to store the modulator and demodulator objects.

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

13 In the constructor method, replace `hMod` and `hDemod` with the object property `obj.Modulator` and `obj.Demodulator` respectively.

```
% Create a 16-QAM modulator
obj.Modulator = comm.RectangularQAMModulator(obj.M, ...
'BitInput',true,'SymbolMapping','Gray');
% Create a 16-QAM demodulator
obj.Demodulator = comm.RectangularQAMDemodulator(obj.M, ...
'BitOutput',true,'SymbolMapping','Gray');
```

In the `run` method, replace `hMod` and `hDemod` with object properties `obj.Modulator` and `obj.Demodulator`.

```
y = modulate(obj.Modulator,x);
z = demodulate(obj.Demodulator,yRx);
```

14 Locate the setup region of the file.

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

15 Set the M value of the modulator and demodulator by adding the following lines of code to the setup.

```
obj.Modulator.M = obj.M;
```

```
obj.Demodulator.M = obj.M;
```

16 Save the file.

17 Create an optimized system. At the MATLAB command line, enter:

```
myOptimSystem = MyGrayCodedModulationOptimized
```

18 Create an Error Rate Test Console and attach the system to the test console. At the MATLAB command line, type:

```
testConsole = commtest.ErrorRate(myOptimSystem)
```

19 At the MATLAB command line, type:

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

This line defines the test point, DemodBitErrors, and compares bits from the TxBits probe to the bits from the RxBits probe. The Error Rate Test Console calculated metrics for this test point.

20 Configure the Error Rate Test Console to run simulations for EbNo values. Start at 2 dB and end at 10 dB, with a step size of 2 dB and M values of 2, 4, 8, and 16. At the MATLAB command line, type:

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

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

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

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

```
testConsole.SimulationLimitOption = 'Number of errors or transmissions';
testConsole.MinNumErrors = 100;
testConsole.ErrorCountTestPoint = 'DemodBitErrors';
testConsole.MaxNumTransmissions = 1e8;
testConsole
```

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

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

MATLAB returns the following information:

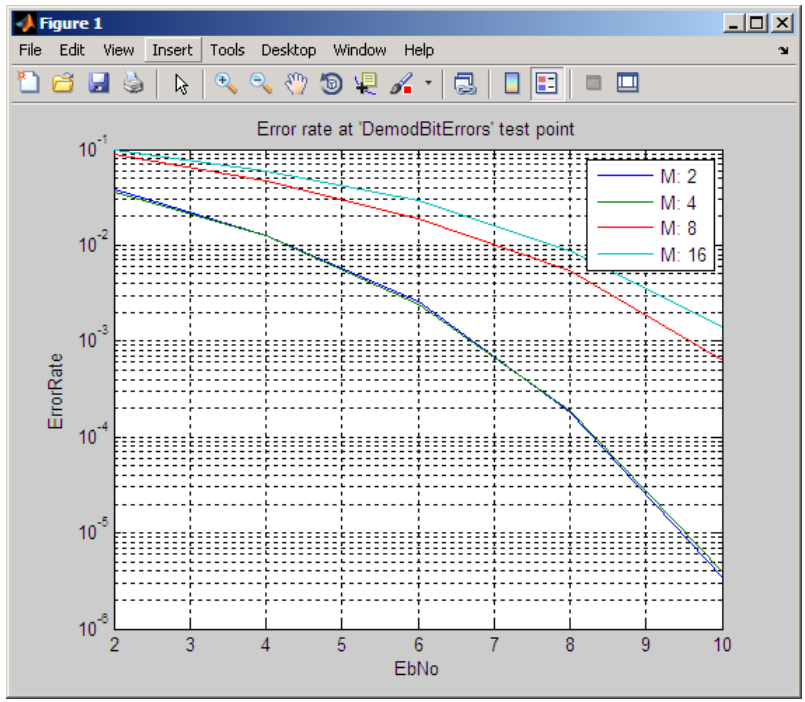
```
Running simulations...
Elapsed time is 191.748359 seconds.
```

Notice that these optimization changes reduce the simulation run time about 10%.

24 Generate the new results in a Figure window. At the MATLAB command line, type:

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

This script generates the following figure.



Error Vector Magnitude (EVM)

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

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

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

Measuring Modulator Accuracy

Overview

The Communications System Toolbox provides two blocks you can use for measuring modulator accuracy: EVM Measurement and MER Measurement.

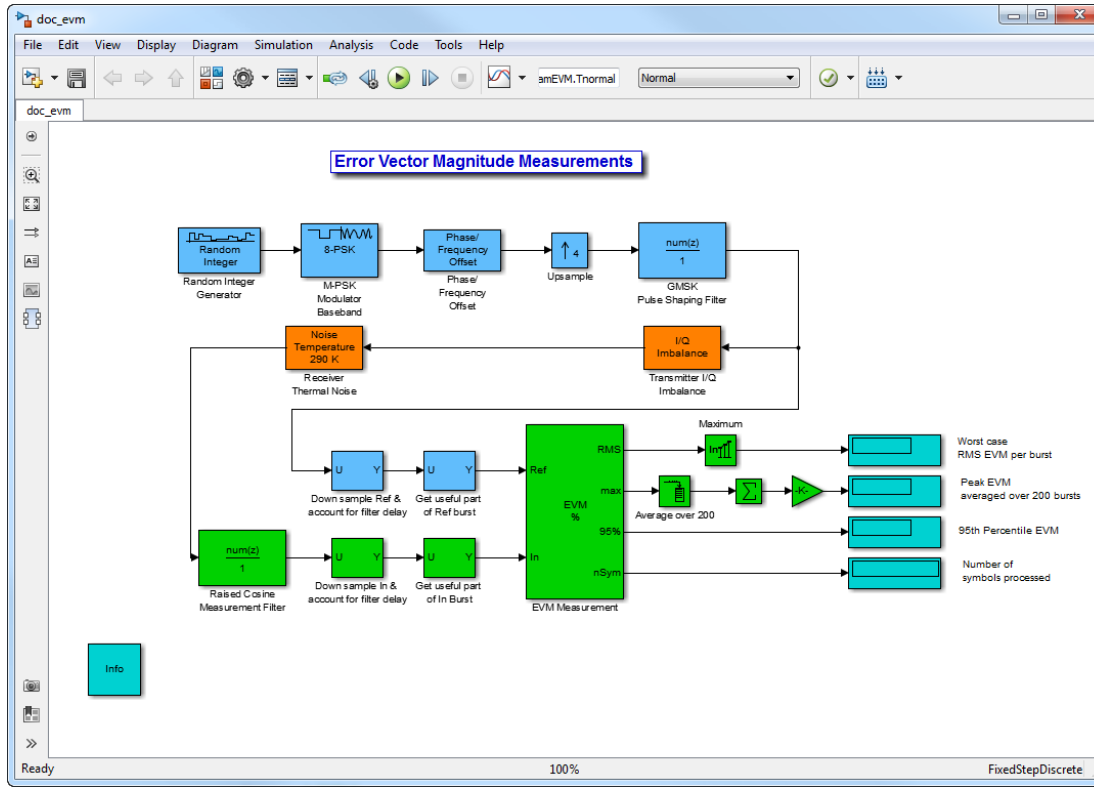
This example tests an EDGE transmitter for system design impairments using EVM measurements. In this example, the EVM Measurements block compares an ideal reference signal to a measured signal, and then computes RMS EVM, maximum EVM, and percentile EVM values. According to the EDGE standard [1], the error vector magnitude of the received signal, calculated relative to the transmitted waveform, should not exceed the following values:

EDGE Standard Measurement Specifications [2]

Measurement	Mobile Station		Base Transceiver Station	
	Normal	Extreme	Normal	Extreme

Measurement	Mobile Station		Base Transceiver Station	
RMS	9%	10%	7%	8%
Peak EVM	30%	30%	22%	22%
95th Percentile EVM	15%	15%	11%	11%

This example uses the following model.



You can open this model by typing `doc_evm` at the MATLAB command line.

Structure

The model essentially contains three parts:

- Transmitter

- Receiver impairments
- EVM calculation

The following sections of the tutorial contain descriptions for each part of the model.

Transmitter

The following blocks comprise the transmitter:

- Random Integer Generator
- M–PSK Modulator Baseband
- Phase/Frequency Offset
- Upsample
- Discrete FIR Filter
- I/Q Imbalance

The Random Integer Generator block simulates random data generation. The EDGE standard specifies that the transmitter performs measurements during the useful part of the burst – excluding tail bits – over at least 200 bursts. In this mode, the transmitter produces 435 symbols per burst (9 additional symbols account for filter delays). The Phase Offset block provides continuous 3π/8 phase rotation to the signal. For synchronization purposes, the Upsample block oversamples the signal by a factor of 4.

The Discrete FIR Filter block provides a GMSK pulse linearization, the main component in a Laurent decomposition of the GMSK modulation [3]. A helper function computes the filter coefficients and uses a direct-form FIR digital filter to create the pulse shaping effect. The filter normalization provides unity gain at the main tap.

The I/Q Imbalance block simulates transmitter impairments. This block adds rotation to the signal, simulating a defect in the transmitter under test. The **I/Q amplitude imbalance** is `0.5` dB, and **I/Q phase imbalance** is 1°.

Receiver Impairments

In this model, the Receiver Thermal Noise block represents receiver impairments. This model assumes 290 K of thermal noise, representing imperfections of the hardware under test.

EVM Calculation

The EVM calculation relies upon the following blocks:

- Discrete FIR Filter
- Selector
- EVM Measurement
- Display

The EVM measurement block computes the vector difference between an ideal reference signal and an impaired signal. The output of the FIR filter provides the `Reference` input for the EVM block. The output of the Noise Temperature block provides the impaired signal at the `Input` port of the EVM block.

While the block has different normalization options available, the EDGE standard requires normalizing by the `Average reference signal power`. For illustration purposes in this example, the EVM block outputs RMS, maximum, and percentile measurement values.

Experimenting with the Model

1. Run the model by clicking the play button in the Simulink model window.
2. Examine the output of the EVM block and compare the measurements to the limits in the EDGE Standard Measurement Specifications table.

   In this example, the EVM Measurement block computes the following:

   - Worst case RMS EVM per burst: 9.77%
   - Peak EVM: 18.95%
   - 95th Percentile EVM:14.76%

   As a result, this simulated EDGE transmitter passes the EVM test for a Mobile Station under extreme conditions.
3. Double-click the I/Q Imbalance block.
4. Enter `2` into **I/Q Imbalance (dB)** and click **OK**.
5. Click the Play button in the Simulink model window.
6. Examine the output of the EVM block. Then, compare the measurements to the limits in the EDGE Standard Measurement Specifications table.

   In this example, the EVM Measurement block computes the following results:

   - Worst case RMS EVM per burst: 15.15%
   - Peak EVM: 29.73%

- 95th Percentile EVM: 22.55%.

These EVM values are clearly unacceptable according to the EDGE standard. You can experiment with the other I/Q imbalance values, examine the impact on calculations, and compare them to the values provided in the table.

References

[1] 3GPP TS 45.004, "Radio Access Networks; Modulation," Release 7, v7.2.0, 2008-02.

[2] 3GPP TS 45.005, "Radio Access Network; Radio transmission and reception," Release 8, v8.1.0, 2008-05.

[3] Laurent, Pierre. "Exact and approximate construction of digital phase modulation by superposition of amplitude modulated pulses (AMP)." *IEEE Transactions on Communications*. Vol. COM-34, #2, Feb. 1986, pp. 150-160.

Modulation Error Ratio (MER)

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

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

Adjacent Channel Power Ratio (ACPR)

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

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

For more information, see the `comm.ACPR` help page.

Obtain ACPR Measurements

Communications System Toolbox contains the `comm.ACPR` System object. In this tutorial, you obtain ACPR measurements using a WCDMA communications signal, according to the 3GPP™ TS 125.104 standard.

This example uses baseband WCDMA sample signals at the input and output of a nonlinear amplifier. The WCDMASignal.mat file contains sample data for use with the tutorial. This file divides the data into 25 signal snapshots of 7e3 samples each and stores them in the columns of data matrices, `dataBeforeAmplifier` and `dataAfterAmplifier`.

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

Create comm.ACPR System Object and Set Up Measurements

1 Define the sample rate, load the WCDMA file, and get the data by entering the following at the MATLAB command line:

```
% System sampling frequency, 3.84 MHz chip rate, 8 samples per chip
SampleRate = 3.84e6*8;
load WCDMASignal.mat
```

```
% Use the first signal snapshot
txSignalBeforeAmplifier = dataBeforeAmplifier(:,1);
txSignalAfterAmplifier = dataAfterAmplifier(:,1);
```

2 Create the comm.ACPR System object and specify the sampling frequency.

```
hACPR = comm.ACPR('SampleRate', SampleRate)
```

The System object presents the following information:

```
               NormalizedFrequency: false
                        SampleRate: 30720000
                MainChannelFrequency: 0
           MainMeasurementBandwidth: 50000
              AdjacentChannelOffset: [-100000 100000]
      AdjacentMeasurementBandwidth: 50000
           MeasurementFilterSource: 'None'
                SpectralEstimation: 'Auto'
                         FFTLength: 'Next power of 2'
                           MaxHold: false
                        PowerUnits: 'dBm'
        MainChannelPowerOutputPort: false
    AdjacentChannelPowerOutputPort: false
```

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

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

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

```
hACPR.MainChannelFrequency = 0;
hACPR.MainMeasurementBandwidth = 3.84e6;
```

4 Specify *adjacent channel* offsets and measurement bandwidths.

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

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

```
hACPR.AdjacentMeasurementBandwidth = 3.84e6;
```

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

5 Set the `MainChannelPowerOutputPort` and `AdjacentChannelPowerOutputPort` properties to `true` by entering the following at the MATLAB command line:

```
hACPR.MainChannelPowerOutputPort = true
hACPR.AdjacentChannelPowerOutputPort = true
```

6 Create a `comm.ACPR` System object to measure the amplifier output.

```
hACPRoutput = clone(hACPR);
```

Obtain the ACPR Measurements

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

1 Obtain the ACPR measurements at the amplifier input:

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

The `comm.ACPR` System object produces the following output measurement data:

```
ACPR =

  -68.6668  -54.9002  -55.0653  -68.4604


mainChannelPower =

   29.5190


adjChannelPower =

  -39.1477  -25.3812  -25.5463  -38.9414
```

2 Obtain the ACPR measurements at the amplifier output:

```
[ACPR mainChannelPower adjChannelPower] = ...,
step(hACPRoutput,txSignalAfterAmplifier)
```

The `comm.ACPR` System object produces the following input measurement data:

```
ACPR =

  -42.1625  -27.0912  -26.8785  -42.4915

mainChannelPower =

   40.6725

adjChannelPower =

   -1.4899   13.5813   13.7941   -1.8190
```

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

Specifying a Measurement Filter

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

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

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

1 Create the filter (using `rcosdesign`, from the Signal Processing Toolbox software) and obtain measurements by entering the following at the MATLAB command line:

```
% Scale for 0 dB passband gain
measFilt = rcosdesign(0.22,16,8)/sqrt(8);
```

2 Set the filter you created in the previous step as the measurement filter for the ACPR object.

```
release(hACPR);
hACPR.MeasurementFilterSource = 'Property';
hACPR.MeasurementFilter = measFilt;
```

3 Implement the same filter at the amplifier output by cloning the `comm.ACPR` System object.

```
hACPRoutput = clone(hACPR)
```

4 Obtain the ACPR power measurements at the amplifier input.

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

The `comm.ACPR` System object produces the following measurement data:

```
ACPR =
  -71.4648  -55.5514  -55.9476  -71.3909
```

5 Obtain the ACPR power measurements at the amplifier output.

```
ACPRoutput = step(hACPRoutput,txSignalAfterAmplifier)
```

The `comm.ACPR` System object produces the following measurement data:

```
ACPR =
   -42.2364  -27.2242  -27.0748  -42.5810
```

Control the Power Spectral Estimator

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

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

```
release(hACPRoutput)
```

```
hACPRoutput.SpectralEstimation = 'Specify window parameters'
```

This change allows you to customize the spectral estimates for obtaining power measurements. For example, you can set the spectral estimator segment length to 1024 and increase the overlap percentage to 50%, reducing the consequent variance increase. You can also choose a window with larger side lobe attenuation (compared to the default Hamming window).

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

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

3 Call the step method to obtain the ACPR power measurements at the amplifier output.

```
ACPRoutput = step(hACPRoutput,txSignalAfterAmplifier)
```

The ACPR object produces the following measurement data:

```
ACPR =
  -44.9399  -30.7136  -30.7670  -44.4450
```

Measure Power Using the Max-Hold Option.

Some communications standards specify using max-hold spectrum power measurements when computing ACPR values. Such calculations compare the current power spectral density vector estimation to the previous max-hold accumulated power spectral density vector estimation. When obtaining max-hold measurements, the object obtains the power spectral density vector estimation using the current input data. It obtains the previous max-hold accumulated power spectral density vector from the previous call to the `step` method. The object uses the maximum values at each frequency bin for calculating average power measurements. A call to the reset method clears the max-hold spectrum.

1 Accumulate max-hold spectra for 25 amplifier output data snapshots and get ACPR measurements by typing the following at the MATLAB command line:

```
for idx = 1:24
    step(hACPRoutput,dataAfterAmplifier(:,idx));
end
```

```
ACPRoutput = step(hACPRoutput,dataAfterAmplifier(:,25))
```

The ACPR object produces the following output data:

```
ACPR =

  -43.1123  -26.6964  -27.0009  -42.4803
```

Plotting the Signal Spectrum

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

```
win = hamming(1024);
[PSD1,F] = pwelch(txSignalBeforeAmplifier,win,50,1024,SampleRate,'centered');
[PSD2,F] = pwelch(txSignalAfterAmplifier,win,50,1024,SampleRate,'centered');
plot(F,10*log10(PSD1))
hold on
grid on
plot(F,10*log10(PSD2),'g')
legend('Amplifier input', 'Amplifier output')
```

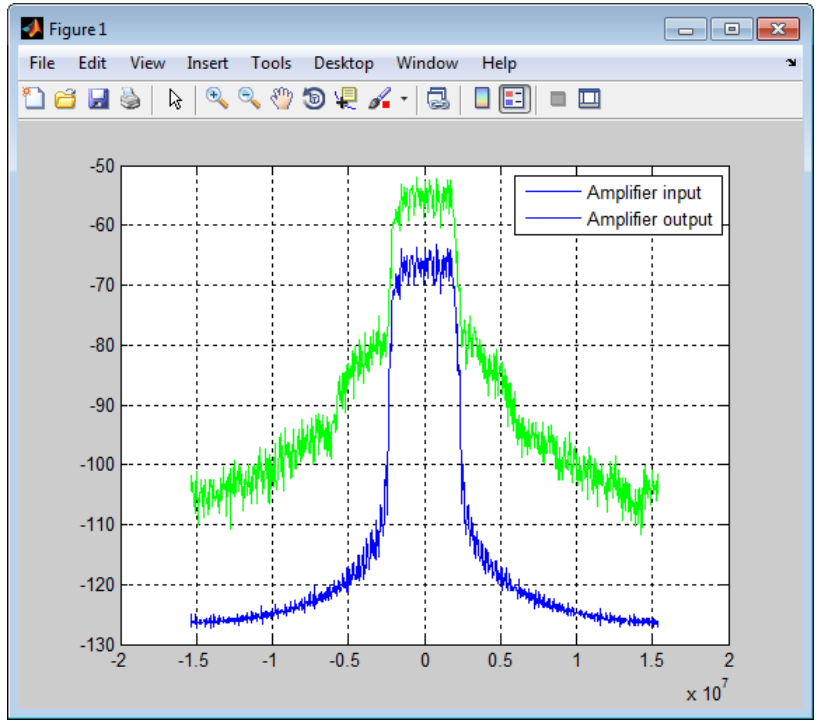


Figure 1
File Edit View Insert Tools Desktop Window Help
Amplifier input
Amplifier output
-50
-60
-70
-80
-90
-100
-110
-120
-130
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
x 10^7

Complementary Cumulative Distribution Function CCDF

The Communications System Toolbox software measures the probability of a signal's instantaneous power to be a specified level above its average power using the `comm.CCDF` System object.

Selected Bibliography for Measurements

[1] Proakis, J. G.,
Digital Communications, 4th Ed., McGraw-Hill, 2001.

[2] Simon, M. K., and Alouini, M. S.,
Digital Communication over Fading Channels – A Unified Approach to Performance Analysis, 1st Ed., Wiley, 2000.

[3] Simon, M. K , "On the bit-error probability of differentially encoded QPSK and offset QPSK in the presence of carrier synchronization",
IEEE Trans. Commun., vol. 54, May 2006, pp. 806-812.

[4] Lee, P. J., "Computation of the bit error rate of coherent M-ary PSK with Gray code bit mapping",
IEEE Trans. Commun., Vol. COM-34, Number 5, pp. 488-491, 1986.

[5] Cho, K., and Yoon, D., "On the general BER expression of one- and two-dimensional amplitude modulations",
IEEE Trans. Commun., vol. 50, no. 7, July 2002, pp. 1074-1080.

[6] Simon, M. K , Hinedi, S. M., and Lindsey, W. C.,
Digital Communication Techniques – Signal Design and Detection, Prentice-Hall, 1995.

[7] Sklar, B.,
Digital Communications, 2nd Ed., Prentice-Hall, 2001.

[8] Lindsey, W. C., "Error probabilities for Rician fading multichannel reception of binary and N-ary signals",
IEEE Trans. Inform. Theory, Vol. IT-10, pp. 339-350, 1964.

[9] Odenwalder, J. P.,
Error Control Coding Handbook (Final report), Linkabit Corp., 15 July 1976.

[10] Gulliver, T. A., "Matching Q-ary Reed-Solomon codes with M-ary modulation,"
IEEE Trans. Commun., vol. 45, no. 11, Nov. 1997, pp. 1349-1353.

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

[12] Frenger, Pal, Pal Orten, and Tony Ottosson, "Convolutional Codes with Optimum Distance Spectrum,"
IEEE Communications Letters, Vol. 3, #11, Nov. 1999, pp. 317-319

Filtering Section

Filtering

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

In this section...

For an example involving raised cosine filters, type `showdemo rcosdemo`.

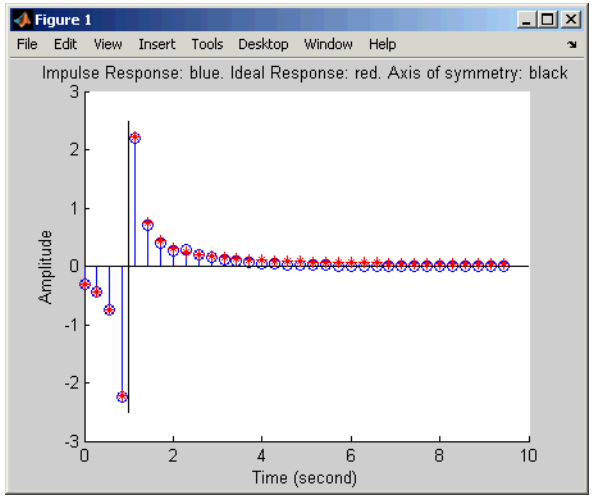
Filter Features

Without propagation delays, both Hilbert filters and raised cosine filters are noncausal. This means that the current output depends on the system's future input. In order to design only *realizable* filters, the `hilbiir` function delays the input signal before producing an output. This delay, known as the filter's *group delay*, is the time between the filter's initial response and its peak response. The group delay is defined as

$$-\frac{d}{d\omega}\theta(\omega)$$

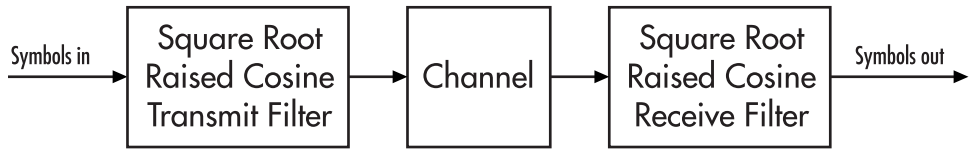
where θ represents the phase of the filter and ω represents the frequency in radians per second. This delay is set so that the impulse response before time zero is negligible and can safely be ignored by the function.

For example, the Hilbert filter whose impulse is shown below uses a group delay of one second. In the figure, the impulse response near time 0 is small and the large impulse response values occur near time 1.



Filtering tasks that blocks in the Communications System Toolbox support include:

- "Filter Using Simulink Raised Cosine Filter Blocks". Raised cosine filters are very commonly used for pulse shaping and matched filtering. The following block diagram illustrates a typical use of raised cosine filters.



- Shaping a signal using ideal rectangular pulses.
- Implementing an integrate-and-dump operation or a windowed integrator. An integrate-and-dump operation is often used in a receiver model when the system's transmitter uses an ideal rectangular-pulse model. Integrate-and-dump can also be used in fiber optics and in spread-spectrum communication systems such as CDMA (code division multiple access) applications.

Additional filtering capabilities exist in the Filter Designs and Multirate Filters libraries of the DSP System Toolbox product.

For more background information about filters and pulse shaping, see the works listed in the "Selected Bibliography Filtering" on page 12-4.

Selected Bibliography Filtering

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

[2] Oppenheim, Alan V., and Ronald W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ, Prentice Hall, 1989.

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

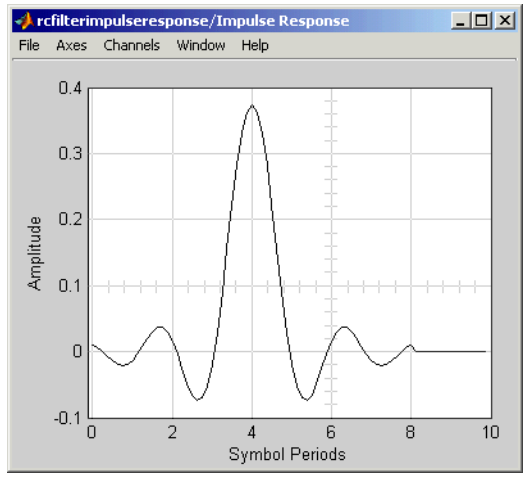
[4] Rappaport, Theodore S., *Wireless Communications: Principles and Practice*, Upper Saddle River, NJ, Prentice Hall, 1996.

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

Group Delay

The raised cosine filter blocks in the `commfilt2` library implement realizable filters by delaying the peak response. This delay, known as the filter's *group delay*, is the length of time between the filter's initial response and its peak response. The filter blocks in this library have a **Filter span in symbols** parameter, which is twice the group delay in symbols.

For example, the square-root raised cosine filter whose impulse response shown in the following figure uses a **Filter span in symbols** parameter of `8` in the filter block. In the figure, the initial impulse response is small and the peak impulse response occurs at the fourth symbol.

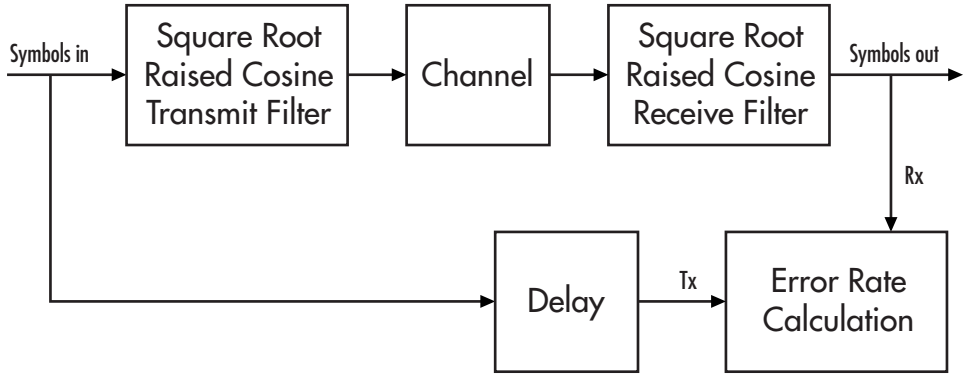


Implications of Delay for Simulations

A filter block's group delay has implications for other parts of your model. For example, suppose you compare the symbol streams marked Symbols In and Symbols Out in the schematics shown on the "Filtering" page by plotting or computing an error rate. Use one of these methods to make sure you are comparing symbols that truly correspond to each other:

- Use the Delay block in DSP System Toolbox to delay the Symbols In signal, thus aligning it with the Symbols Out signal. Set the **Delay** parameter equal to the filter's

group delay (or the sum of both values, if your model uses a pair of square root raised cosine filter blocks). The following figure illustrates this usage.



- Use the Align Signals block to align the two signals.
- When using the Error Rate Calculation block to compare the two signals, increase the **Receive delay** parameter by the group delay value (or the sum of both values, if your model uses a pair of square-root raised cosine filter blocks). The **Receive delay** parameter might include other delays as well, depending on the contents of your model.

For more information about how to manage delays in a model, see “Delays”.

Pulse Shaping Using a Raised Cosine Filter

Filter a 16-QAM signal using a pair of square root raised cosine matched filters. Plot the eye diagram and scatter plot of the signal. After passing the signal through an AWGN channel, calculate the number of bit errors.

Set the simulation parameters.

```
M = 16;                                 % Modulation order
k = log2(M);                            % Bits/symbol
n = 20000;                              % Transmitted bits
nSamp = 4;                              % Samples per symbol
EbNo = 10;                              % Eb/No (dB)
```

Create a rectangular QAM modulator and demodulator System object™ pair that operates on binary data.

```
hMod = comm.RectangularQAMModulator(M,'BitInput',true);
hDemod = comm.RectangularQAMDemodulator(M,'BitOutput',true);
```

Set the filter parameters.

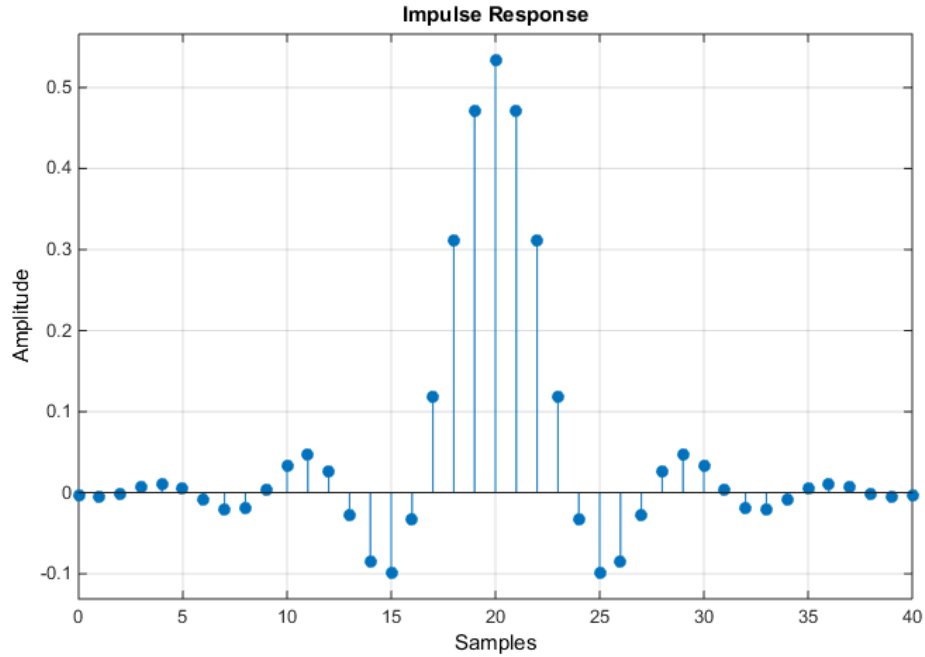
```
span = 10;                              % Filter span in symbols
rolloff = 0.25;                         % Rolloff factor
```

Create the raised cosine transmit and receive filters using the previously defined parameters.

```
hTxFilter = comm.RaisedCosineTransmitFilter('RolloffFactor',rolloff, ...
    'FilterSpanInSymbols',span,'OutputSamplesPerSymbol',nSamp);

hRxFilter = comm.RaisedCosineReceiveFilter('RolloffFactor',rolloff, ...
    'FilterSpanInSymbols',span,'InputSamplesPerSymbol',nSamp, ...
    'DecimationFactor',nSamp);
```

Plot the impulse response of `hTxFilter`.

```
fvtool(hTxFilter,'impulse')
```



Calculate the delay through the matched filters. The group delay is half of the filter span through one filter and is, therefore, equal to the filter span for both filters. Multiply by the number of bits per symbol to get the delay in bits.

```
filtDelay = k*span;
```

Create an error rate counter System object. Set the `ReceiveDelay` property to account for the delay through the matched filters.

```
hErr = comm.ErrorRate('ReceiveDelay',filtDelay);
```

Generate binary data.

```
x = randi([0 1],n,1);
```

Modulate the data using the `step` method of `hMod`.

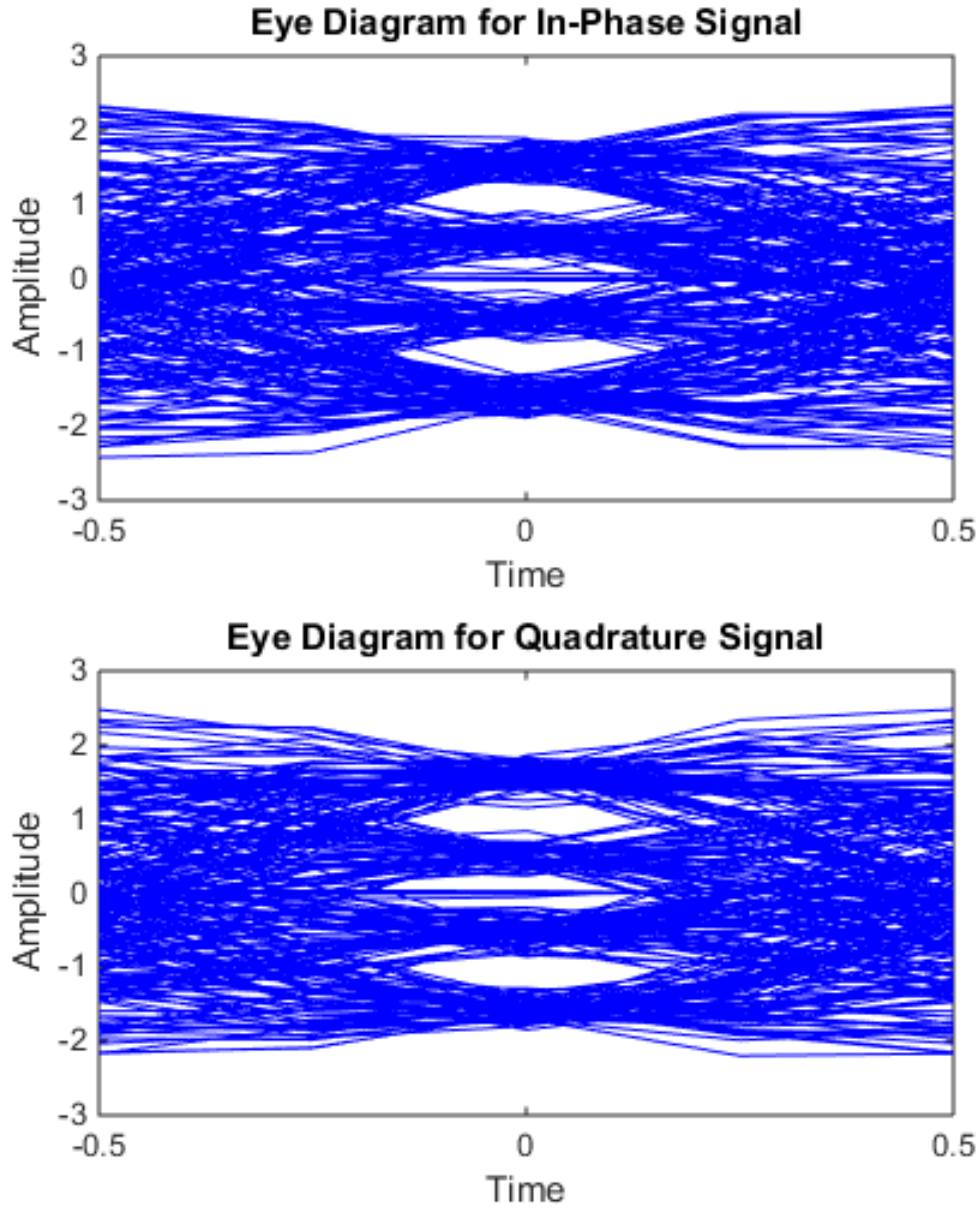
```
modSig = step(hMod,x);
```

Filter the modulated signal.

```
txSig = step(hTxFilter,modSig);
```

Plot the eye diagram of the first 1000 samples.

```
eyediagram(txSig(1:1000),nSamp)
```

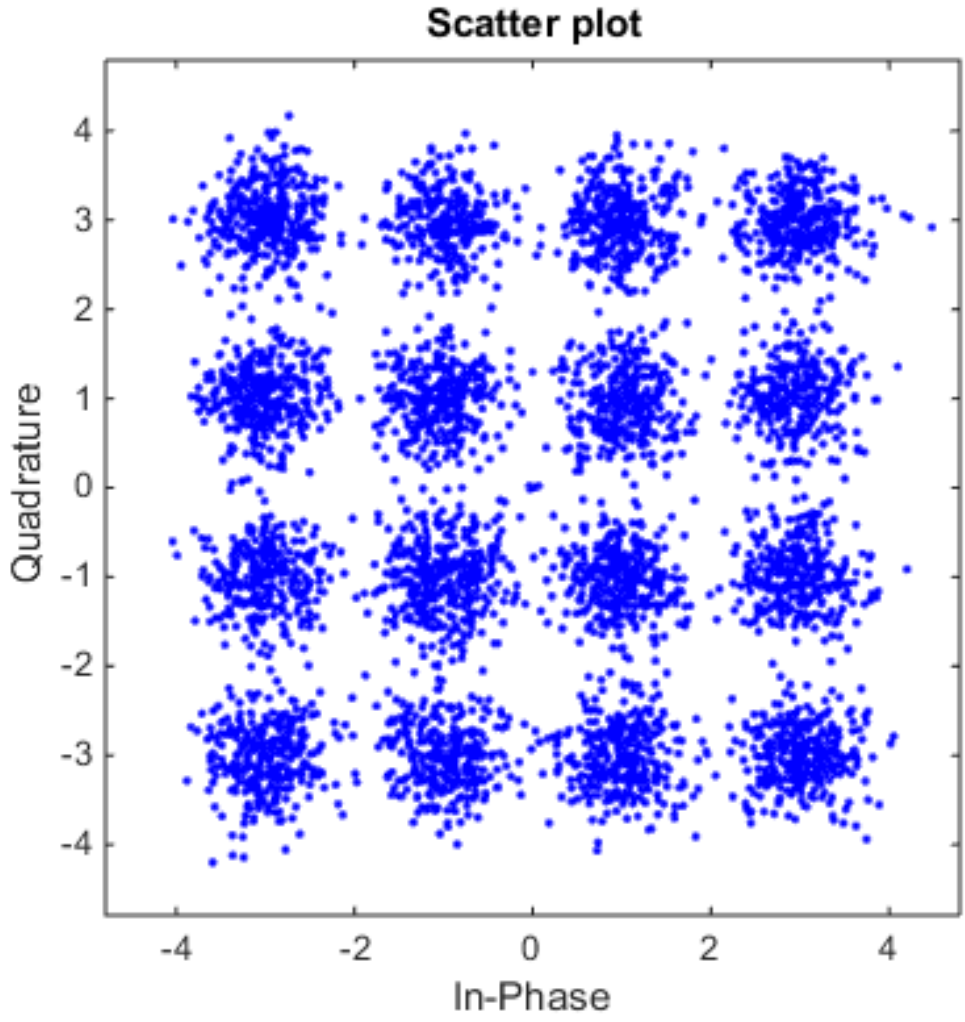

Eye Diagram for In-Phase Signal
Amplitude
Time
3
2
1
0
-1
-2
-3
-0.5
0
0.5
Eye Diagram for Quadrature Signal
Amplitude
Time
3
2
1
0
-1
-2
-3
-0.5
0
0.5

Calculate the signal-to-noise ratio (SNR) in dB given `EbNo`. Pass the transmitted signal through the AWGN channel using the `awgn` function.

```
SNR = EbNo + 10*log10(k) - 10*log10(nSamp);
noisySig = awgn(txSig,SNR,'measured');
```

Filter the noisy signal and display its scatter plot.

```
rxSig = step(hRxFilter,noisySig);
scatterplot(rxSig)
```



Demodulate the filtered signal and calculate the error statistics. The delay through the filters is accounted for by the `ReceiveDelay` property in `hErr` .

```
z = step(hDemod,rxSig);
```

```
errStat = step(hErr,x,z);
fprintf('\nBER = %5.2e\nBit Errors = %d\nBits Transmitted = %d\n',...
    errStat)
```

```
BER = 1.85e-03
Bit Errors = 37
Bits Transmitted = 19960
```

Design Raised Cosine Filters Using MATLAB Functions

In this section...

Section Overview

The `rcosdesign` function designs (but does not apply) filters of these types:

- Finite impulse response (FIR) raised cosine filter
- FIR square-root raised cosine filter

The function returns the FIR coefficients as output.

Example Designing a Square-Root Raised Cosine Filter

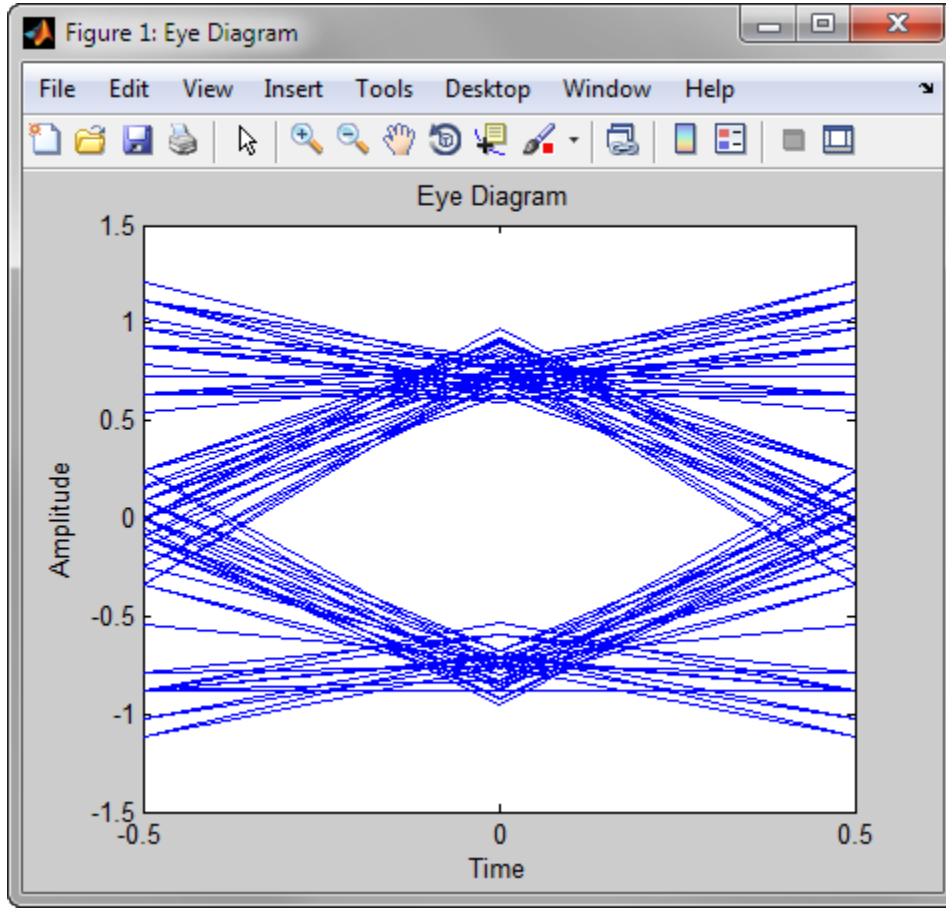
For example, the command below designs a square-root raised cosine FIR filter with a rolloff of 0.25, a filter span of 6 symbols, and an oversampling factor of 2.

```
sps = 2;
num = rcosdesign(0.25, 6, sps)

num =
  Columns 1 through 7
   -0.0265    0.0462    0.0375   -0.1205   -0.0454    0.4399    0.7558
  Columns 8 through 13
    0.4399   -0.0454   -0.1205    0.0375    0.0462   -0.0265
```

Here, the vector `num` contains the coefficients of the filter, in ascending order of powers of z^{-1}.

You can use the `upfirdn` function to filter data with a raised cosine filter generated by `rcosdesign`. The following code illustrates this usage:

```
d = 2*randi([0 1], 100, 1)-1;
f = upfirdn(d, num, sps);
eyediagram(f(7:200),sps)
```



The eye diagram shows an imperfect eye because `num` characterizes a square-root filter.

Filter Using Simulink Raised Cosine Filter Blocks

The Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks are designed for raised cosine filtering. Each block can apply a square-root raised cosine filter or a normal raised cosine filter to a signal. You can vary the rolloff factor and span of the filter.

The Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks are tailored for use at the transmitter and receiver, respectively. The transmit filter outputs an upsampled (interpolated) signal, while the receive filter expects its input signal to be upsampled. The receive filter lets you choose whether to have the block downsample (decimate) the filtered signal before sending it to the output port.

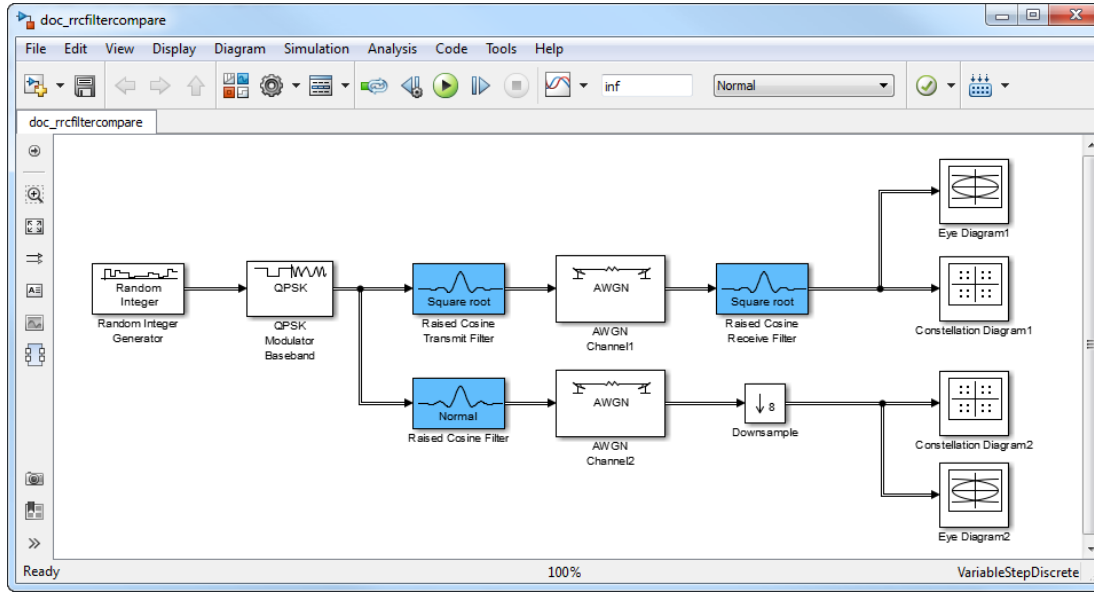
Both raised cosine filter blocks introduce a propagation delay, as described in "Group Delay".

Combining Two Square-Root Raised Cosine Filters

This model shows how to split the filtering equally between the transmitter's filter and the receiver's filter by using a pair of square root raised cosine filters.

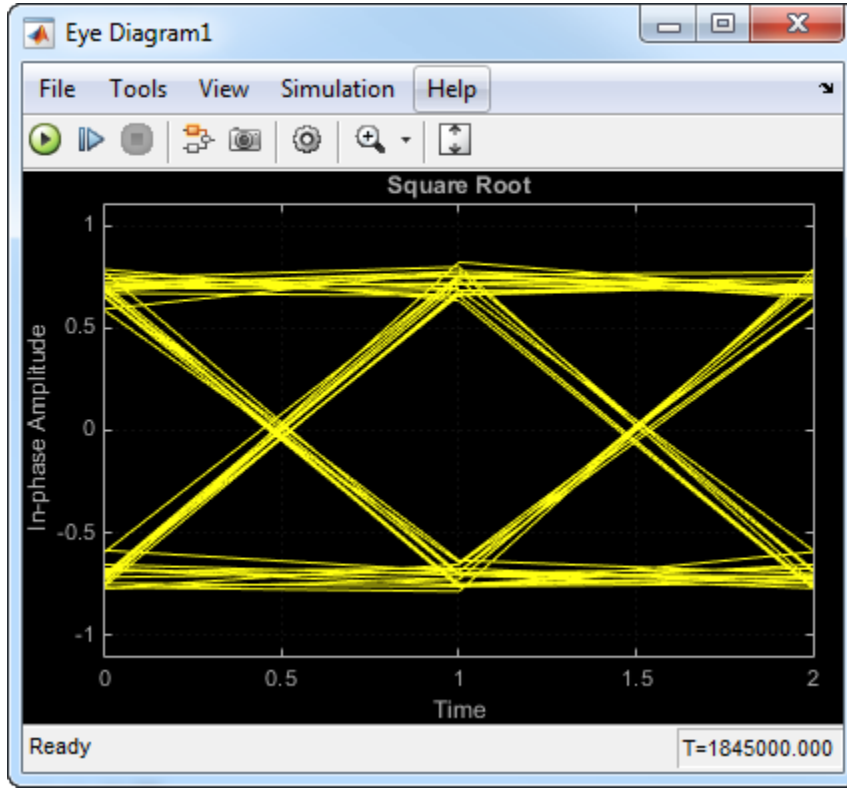
The use of two matched square root raised cosine filters is equivalent to a single normal raised cosine filter. To see this illustrated, load the model doc_rrcfiltercompare by typing the following at the MATLAB command line.

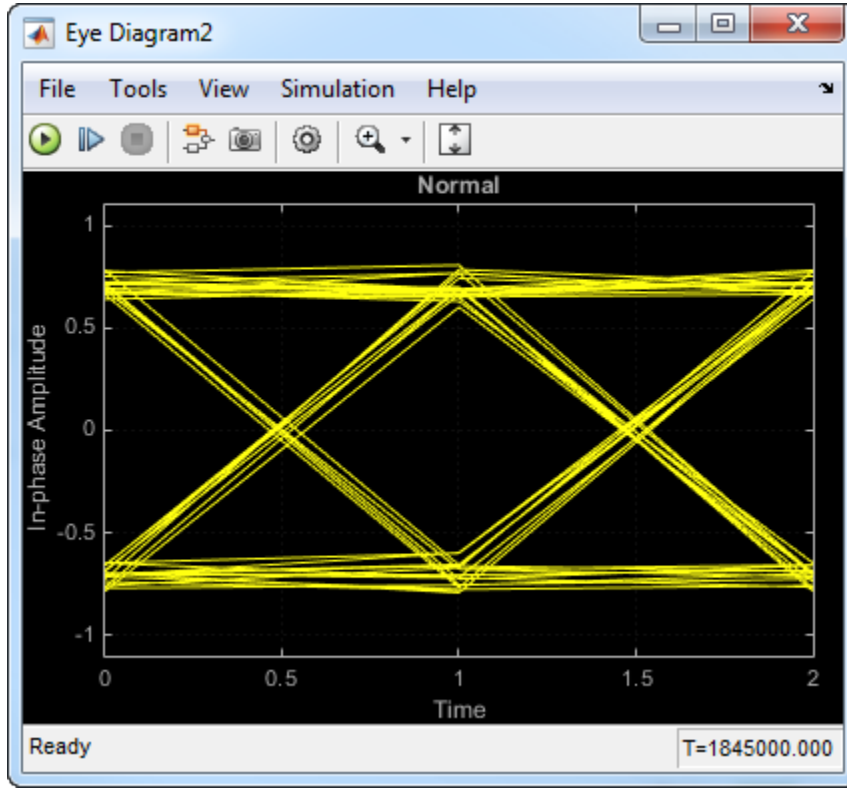
```
doc_rrcfiltercompare
```

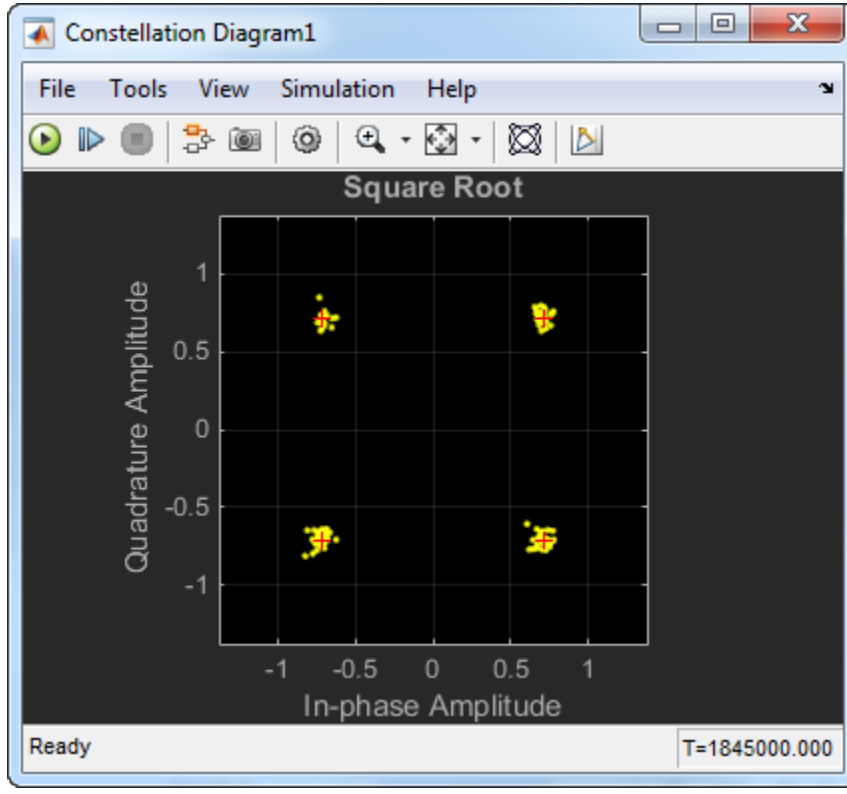


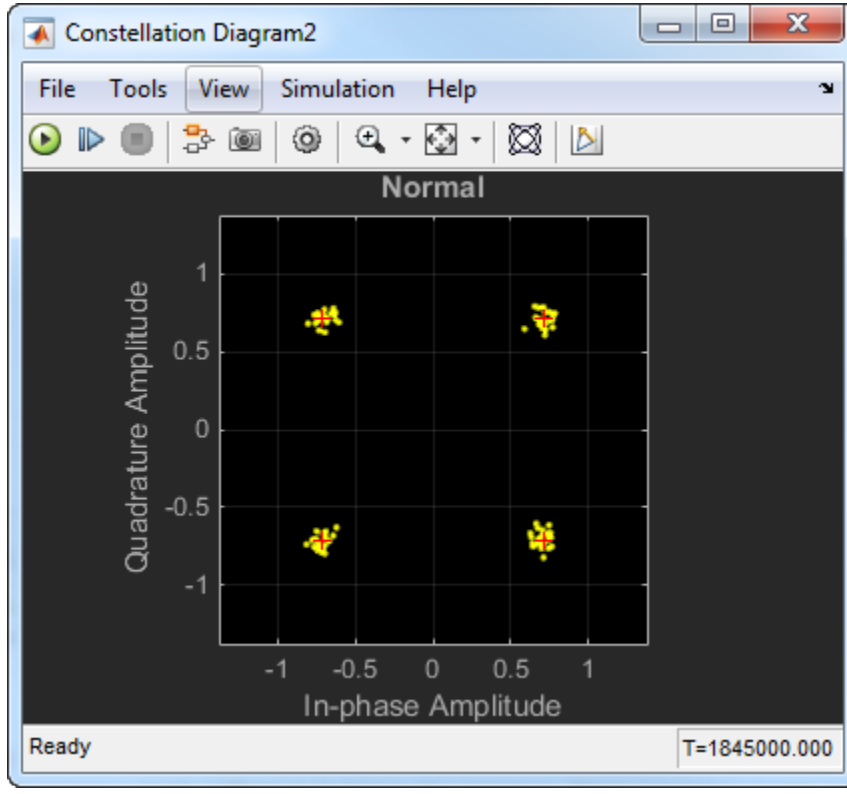
The filters share the same span and use the same number samples per symbol but the filters on the upper path have a square root shape while the filter on the lower path has the normal shape.

Run the model and observe the eye and constellation diagrams. The performance is nearly identical for the two methods. Note that the limited impulse response of practical square root raised cosine filters causes a slight difference between the response of two cascaded square root raised cosine filters and the response of one raised cosine filter.


Eye Diagram1
File
Tools
View
Simulation
Help
Square Root
In-phase Amplitude
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
Time
Ready
T=1845000.000

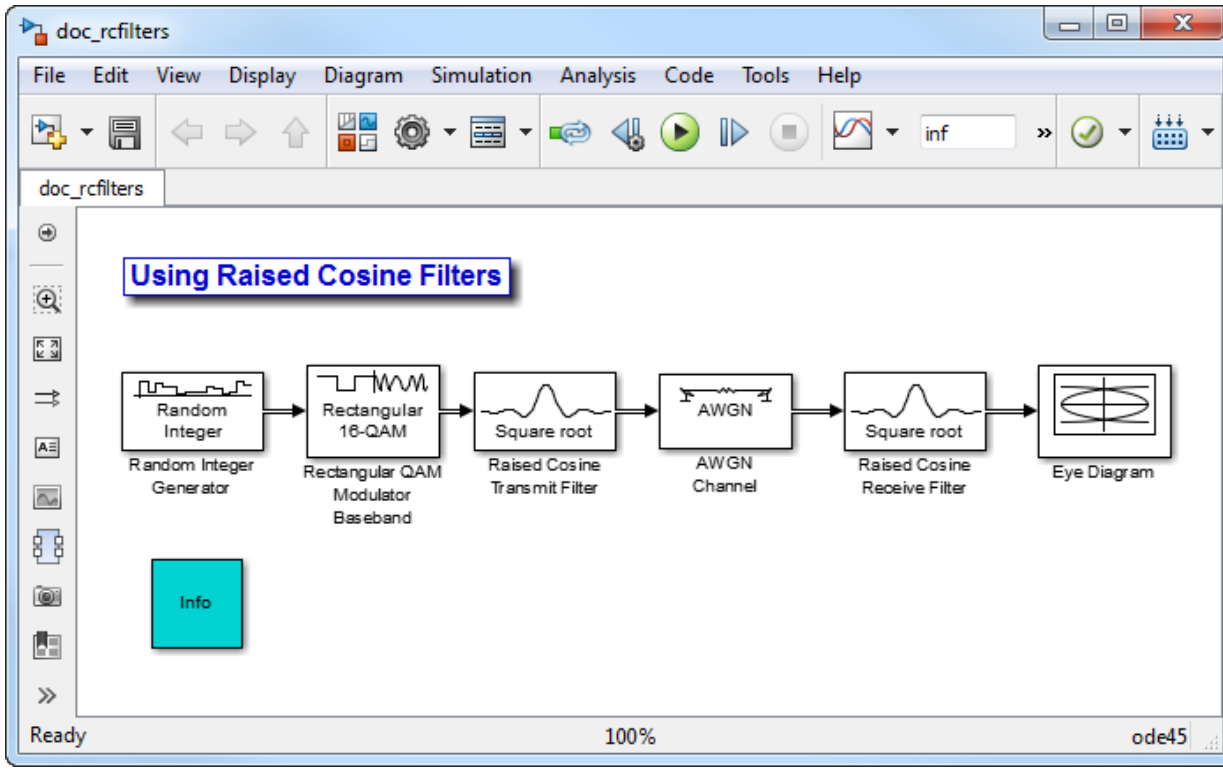

Eye Diagram2
File
Tools
View
Simulation
Help
Normal
In-phase Amplitude
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
Time
Ready
T=1845000.000


Constellation Diagram1
File
Tools
View
Simulation
Help
Square Root
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=1845000.000


Constellation Diagram2
File
Tools
View
Simulation
Help
Normal
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=1845000.000

Design Raised Cosine Filters in Simulink

This example illustrates a typical setup in which a transmitter uses a square root raised cosine filter to perform pulse shaping and the corresponding receiver uses a square root raised cosine filter as a matched filter. The example plots an eye diagram from the filtered received signal.

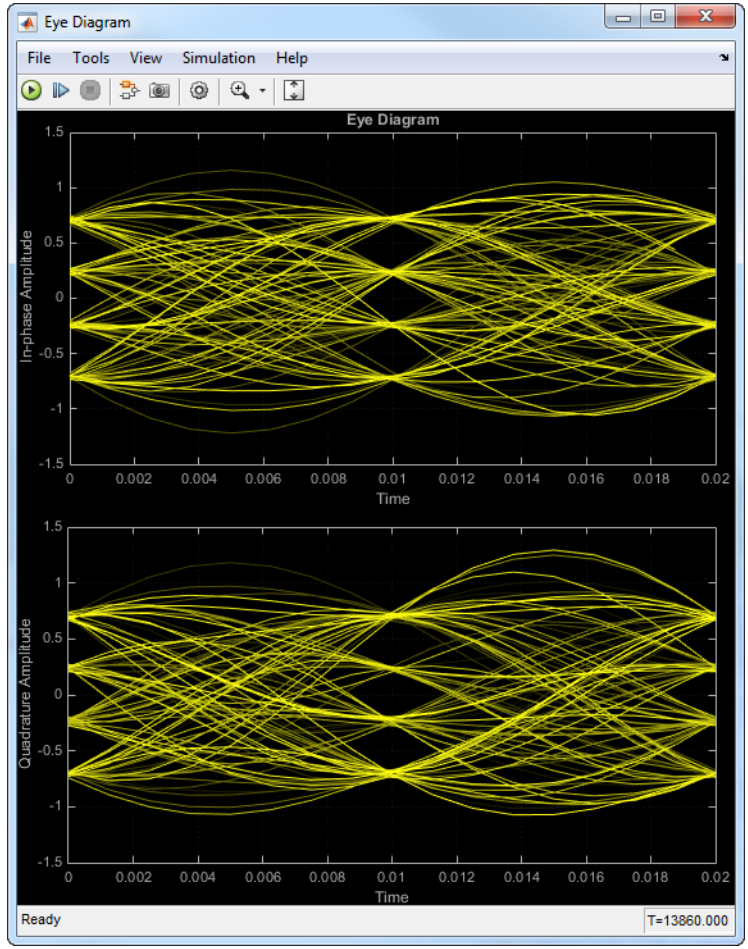


To open the model, enter `doc_rcfilters` at the MATLAB command line. The following is a summary of the block parameters used in the model:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library:
  - **M-ary number** is set to `16`.
  - **Sample time** is set to `1/100`.

  - **Frame-based outputs** is selected.
  - **Samples per frame** is set to `100`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation:
  - **Normalization method** is set to `Peak Power`.
  - **Peak power** is set to `1`.
- Raised Cosine Transmit Filter, in the Comm Filters library:
  - **Filter span in symbols** is set to `8`.
  - **Rolloff factor** is set to `0.2`
- AWGN Channel, in the Channels library:
  - **Mode** is set to `Signal to noise ratio (SNR)`.
  - **SNR** is set to `40`.
  - **Input signal power** is set to `0.0694`. The power gain of a square-root raised cosine transmit filter is $\frac{1}{N}$, where N represents the upsampling factor of the filter. The input signal power of filter is `0.5556`. Because the **Peak power** of the 16-QAM Rectangular modulator is set to 1 watt, it translates to an average power of 0.5556. Therefore, the output signal power of filter is $\frac{0.5556}{8} = 0.0694$.
- Raised Cosine Receive Filter, in the Comm Filters library:
  - **Filter span in symbols** is set to `8`.
  - **Rolloff factor** is set to `0.2`.
- Eye Diagram, in the Comm Sinks library:
  - **Symbols per trace** is set to `2`.
  - **Traces to display** is set to `100`.

Running the simulation produces the following eye diagram. The eye diagram has two widely opened “eyes” that indicate appropriate instants at which to sample the filtered signal before demodulating. This illustrates the absence of intersymbol interference at the sampling instants of the received waveform.



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.

Visual Analysis

Constellation Visualization

Some linear modulator blocks provide the capability to visualize a signal constellation right from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. The following 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 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

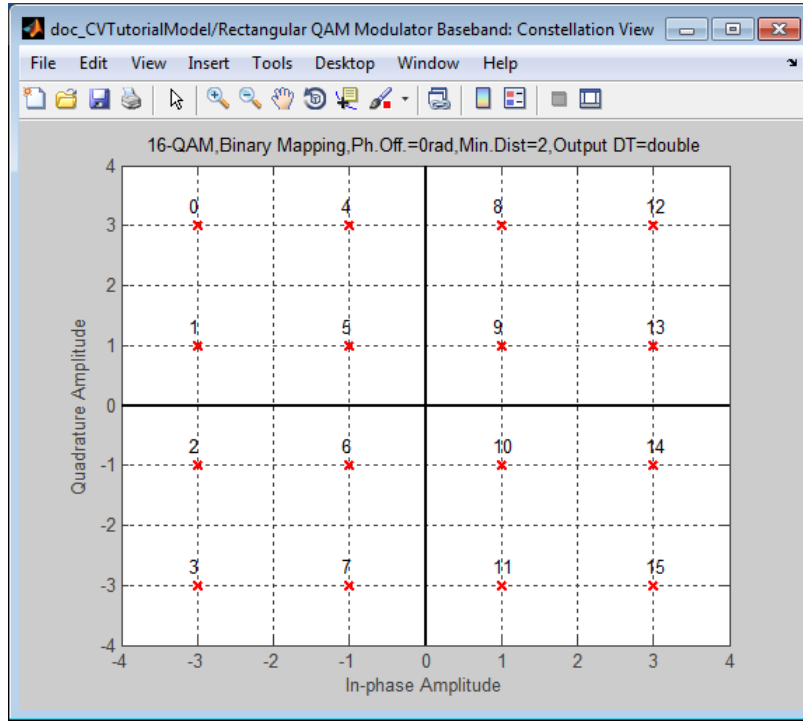
The title of the plot indicates the values of significant parameters. You can use the full array of MATLAB plot tools to manipulate plot figures. Selecting `Inherit via back propagation` for the **Output Data Type** generates a constellation plot with `double` as the **Output data type**.

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:

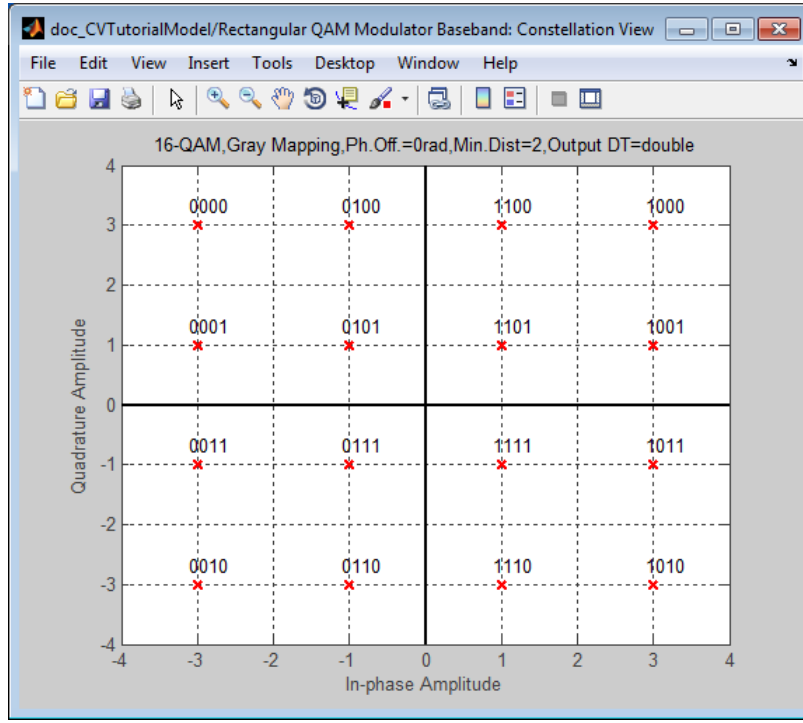
- Uses a 16-QAM modulation scheme
- Uses Binary constellation mapping
- Has 0 degree phase offset
- Has a minimum distance between two constellation points of 2

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.

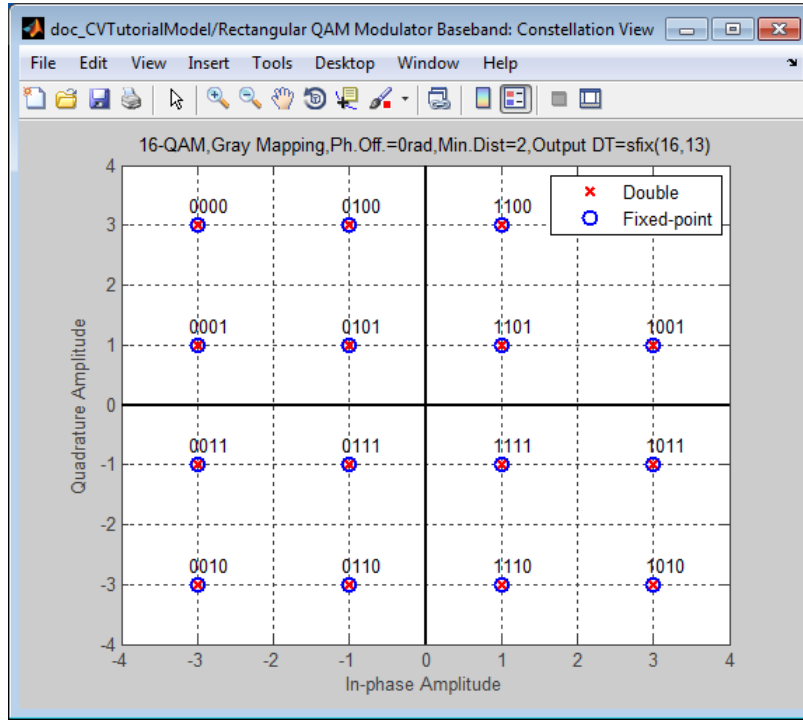


7 You can overlay and compare the effect that two different data type selections have on a signal constellation. For example, you can compare the effect of changing **Output data type** from `double` to `Fixed-point` on the signal constellation.

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.

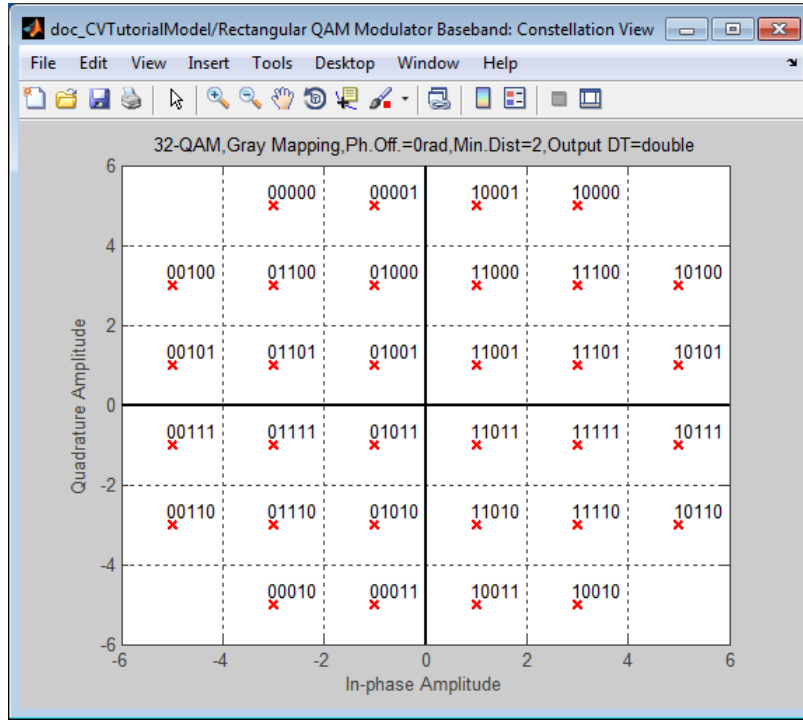
Note: The model workspace in Simulink has priority over the base workspace in MATLAB.

---

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.

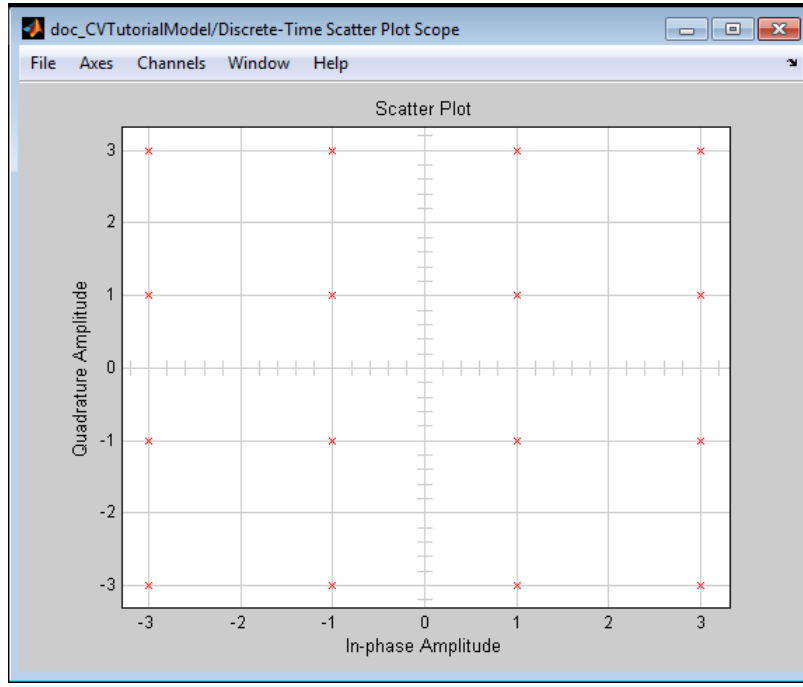
11 Click the **Data Types** tab and then select `double` for the **Output data type** parameter.

12 Click the **Main** tab. Then, click the **View Constellation** button and observe the results.

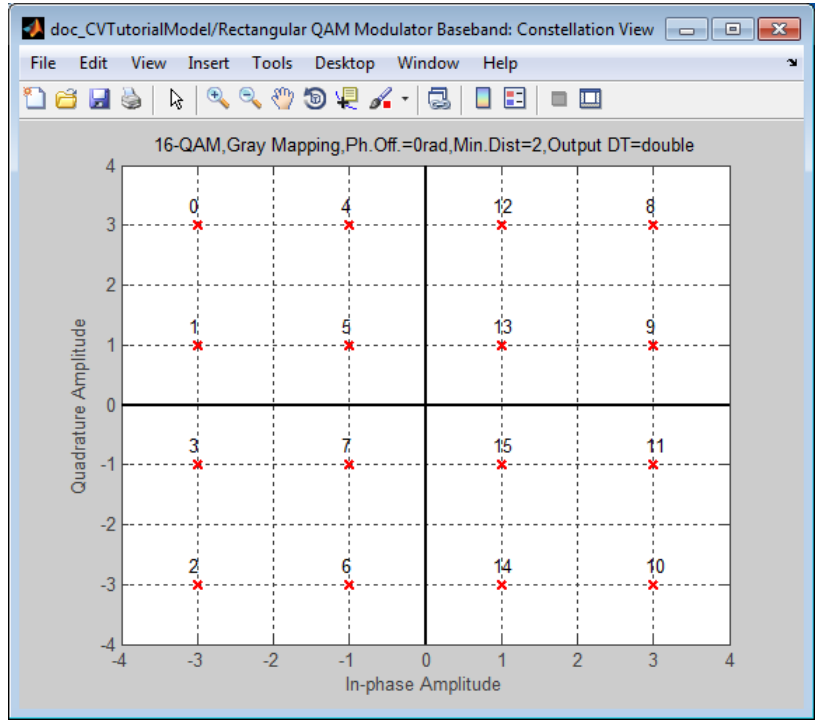


13 You can also use the Constellation Visualization feature while a simulation is running. Type `M=16` in the MATLAB workspace, select `Integer` for the **Input type** and click **Apply**.

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 Constellation Visualization 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

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.

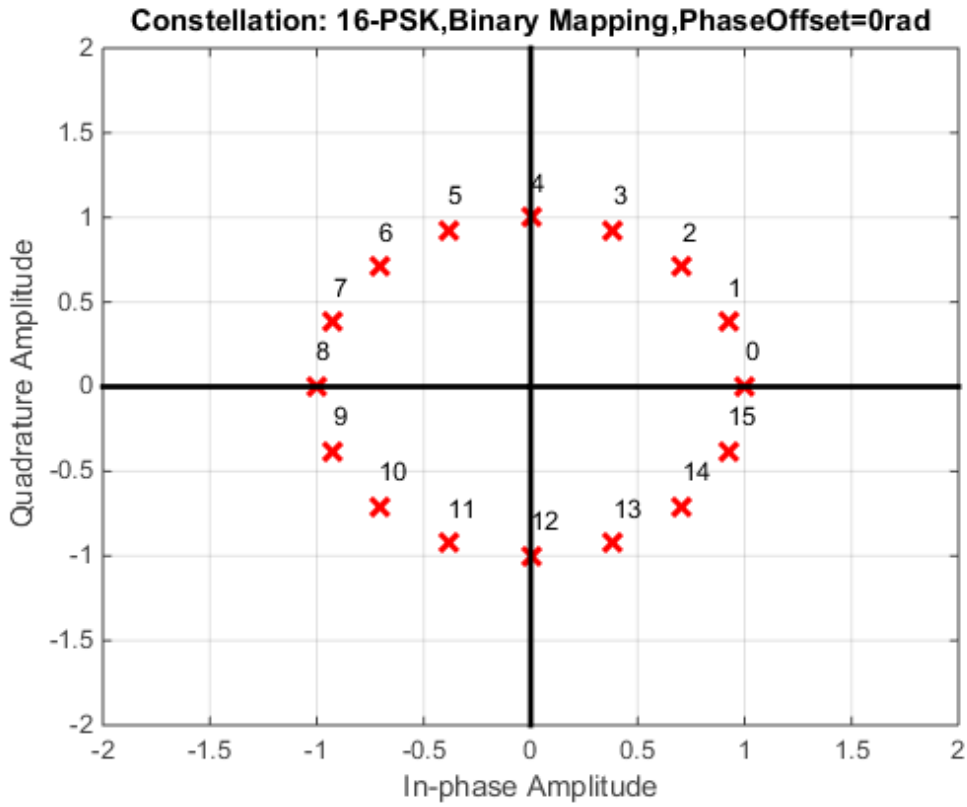
```
M = 16;             % Modulation alphabet size
phOffset = 0;       % Phase offset
symMap = 'binary';  % Symbol mapping (either 'binary' or 'gray')
```

Construct the modulator object.

```
hMod = comm.PSKModulator(M,phOffset,'SymbolMapping',symMap);
```

Plot the constellation.

```
constellation(hMod)
```



Create 32-QAM Constellation Diagram

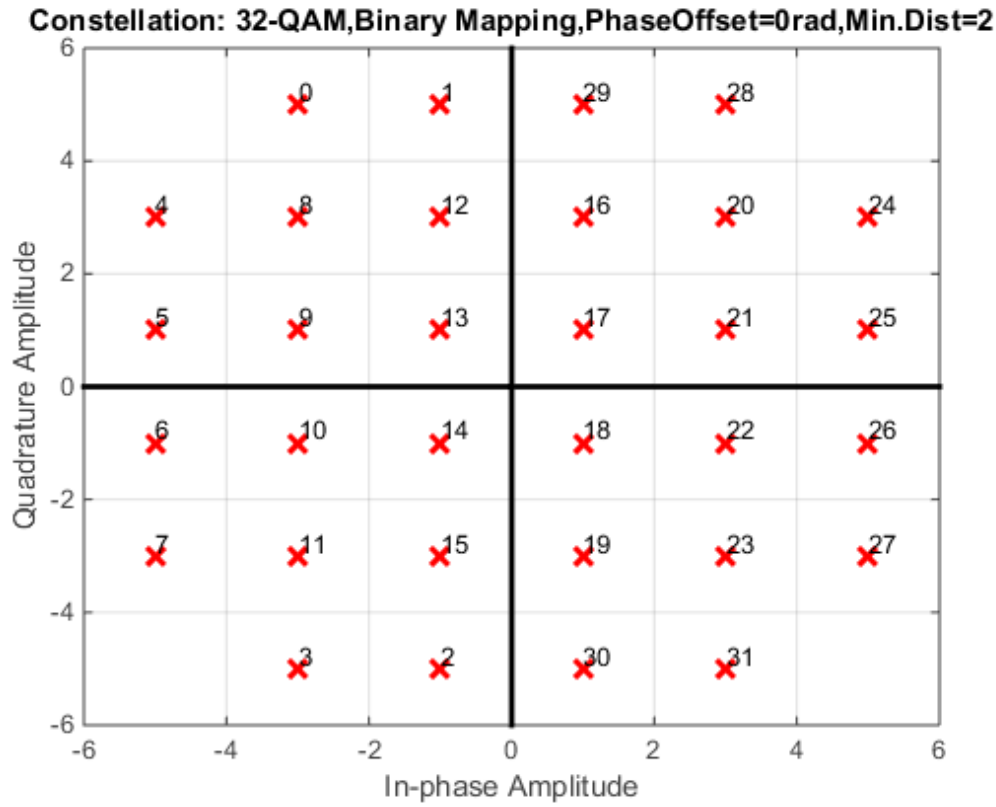
This example shows how to plot a QAM constellation having 32 points.

Construct the modulator object using name-value pairs to set the properties.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',32, ...
    'SymbolMapping','binary');
```

Plot the constellation.

```
constellation(hMod)
```



Create 8-QAM Gray Coded Constellation Diagram

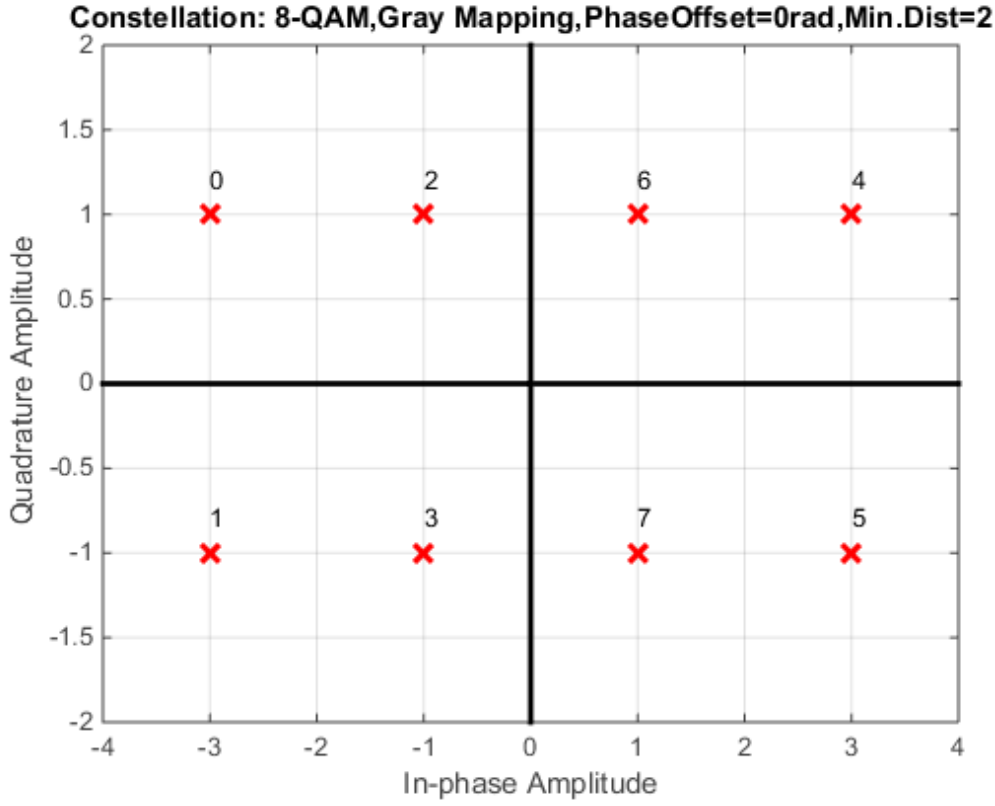
This example shows how to plot a Gray-coded 8-QAM constellation.

Construct the modulator object using a name-value pair to set the properties. Note that Gray coding is the default symbol mapping for the `comm.RectangularQAMModulator` System object.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',8);
```

Plot the constellation.

```
constellation(hMod)
```



Plot a Triangular Constellation for QAM

This example shows how to plot a customized QAM reference constellation.

Describe the constellation.

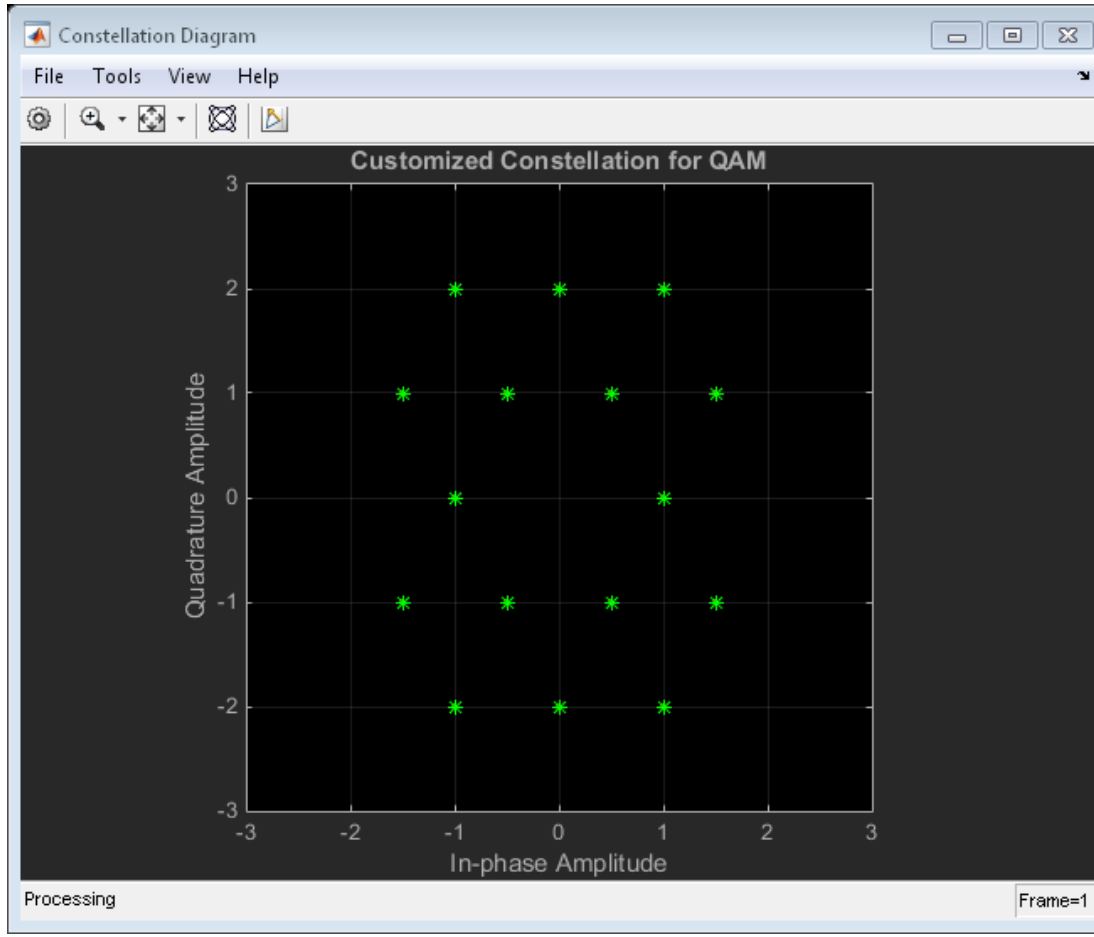
```
inphase = [1/2 -1/2 1 0 3/2 -3/2 1 -1];
quadr = [1 1 0 2 1 1 2 2];
inphase = [inphase; -inphase];
inphase = inphase(:);
quadr = [quadr; -quadr];
quadr = quadr(:);
refConst = inphase + 1i*quadr;
```

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.

```
h = comm.ConstellationDiagram('Title','Customized Constellation for QAM', ...
    'XLimits',[-3 3],'YLimits',[-3 3], ...
    'ReferenceConstellation',refConst, ...
    'ReferenceMarker','*','ReferenceColor',[0 1 0]);
```

Use the step function to plot the customized constellation.

```
step(h,refConst)
```



Eye Diagram Analysis

In digital communications, an eye diagram provides a visual indication of how noise might impact system performance.

Use the EyeScope tool to examine the data that an eye diagram object contains. EyeScope shows both the eye diagram plot and measurement results in a unified, graphical environment. You can import, and compare measurement results for, multiple eye diagram objects.

For 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 Diagram and Scatter Plot example covers eye diagram analysis applied to a communications system.

For a complete list of EyeScope measurements definitions, refer to "'Measurements'" in the *Communications System Toolbox User's Guide*.

For instructions on how to perform basic EyeScope tasks, see the `EyeScope` reference page.

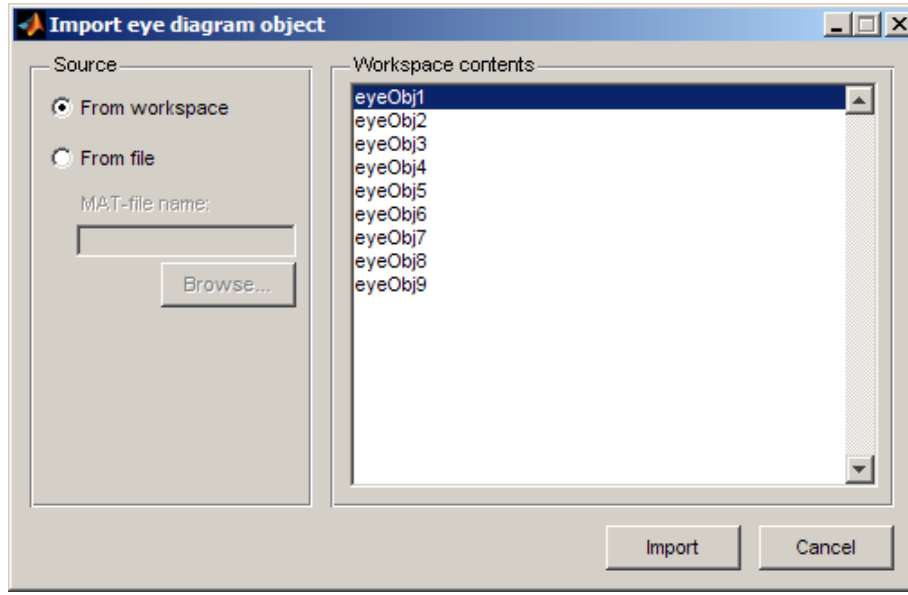
Import Eye Diagrams and Compare Measurement Results

This section provides a step-by-step introduction for using EyeScope to import eye diagram objects, select and change which eye diagram measurements EyeScope displays, compare measurement results, and print a plot object.

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

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

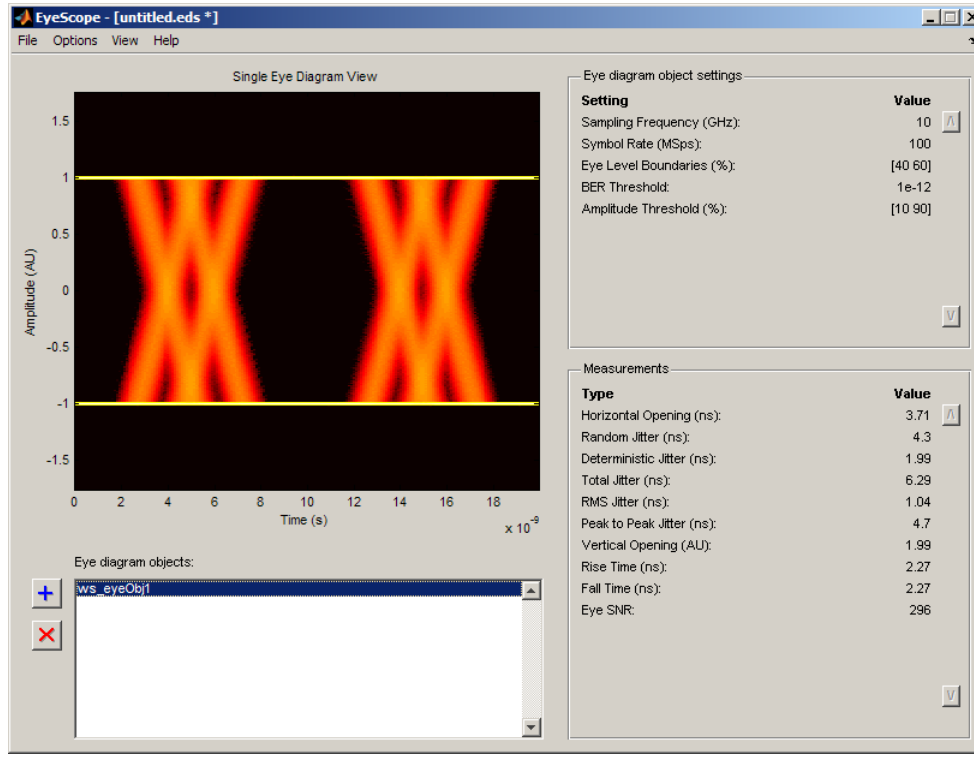
   The **Import eye diagram object** dialog box opens.



In this window, the **Workspace contents** panel displays all eye diagram objects available in the source location.

4 Select **eyeObj1** and click **Import**. EyeScope imports the object, displaying an image in the object plot and listing the file name in the **Eye diagram objects** list.

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



Review the image and note the default **Eye diagram object settings** and **Measurements** selections. For more information, refer to the "`EyeScope`" reference page.

5 In the EyeScope window, click the Import button.

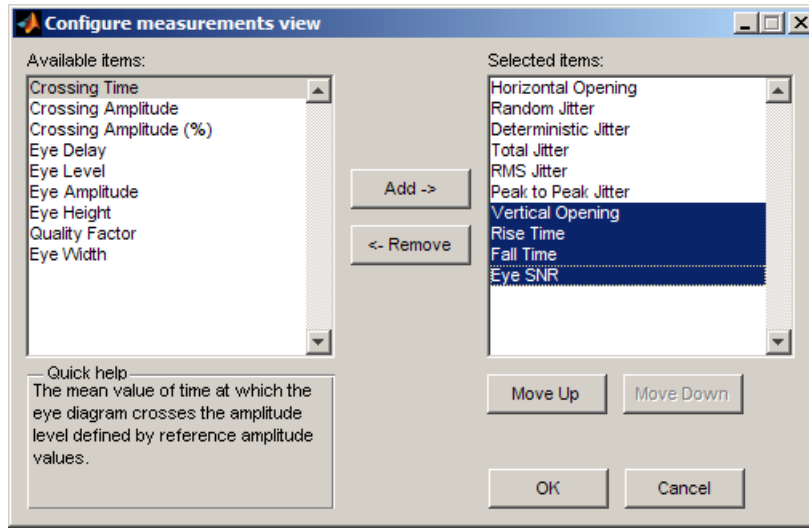
6 From the Import eye diagram object window, click to select eyeObj5 then click the **Import** button.

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

- Next, click **ws_eyeObj1** and note the EyeScope plot and measurement values changes.

7 To change or remove measurements from the EyeScope display:

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



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

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

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

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

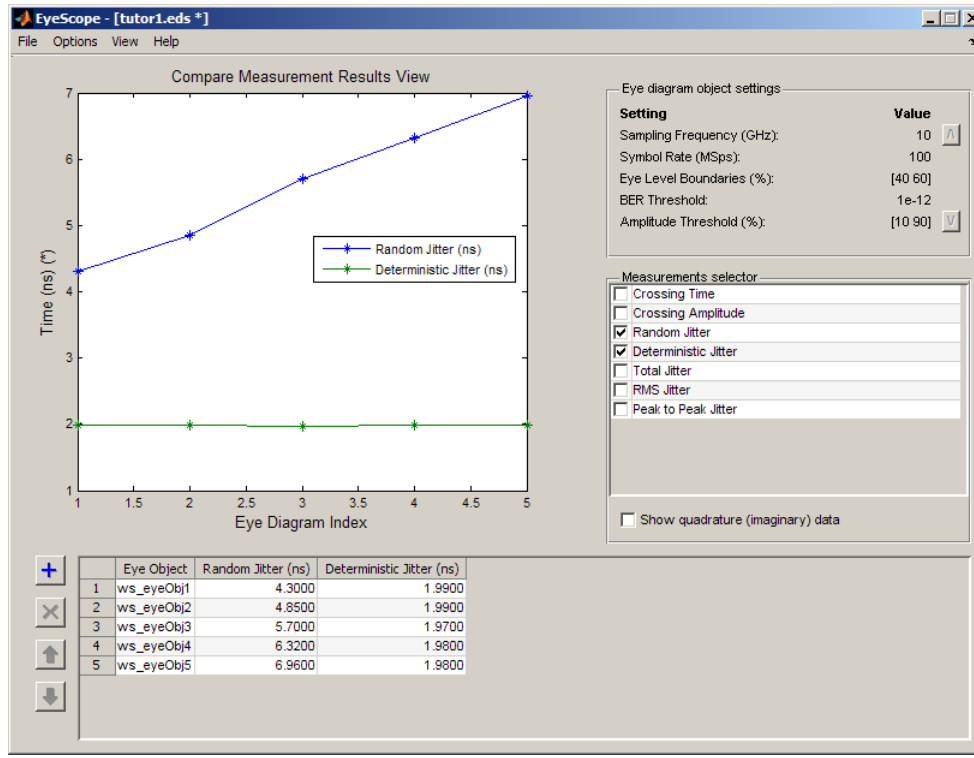
10 Select **File** > **Save session as** and then type a file name in the pop-up window.

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

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

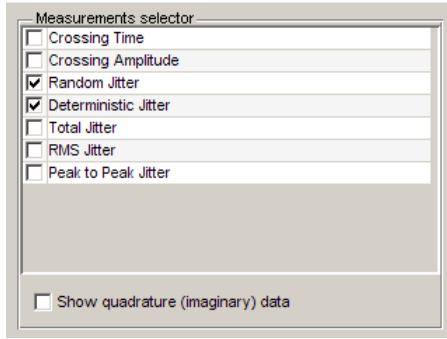
13 Click **File > Import Eye Diagram Object**, and select **ws_eyeObj5**.

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



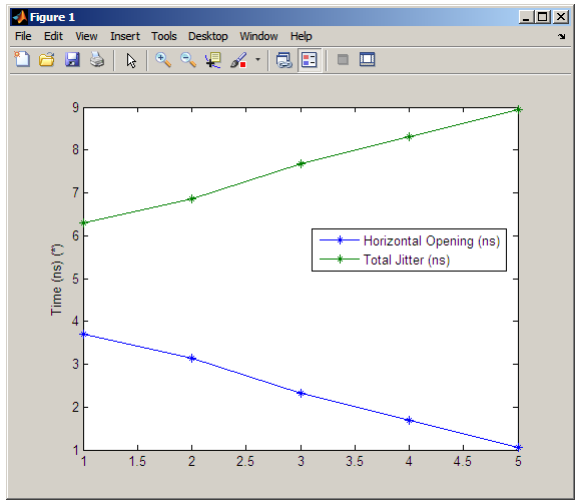
In the data set, random jitter increases from experiment 1 to experiment 5, as you can see in both the table and plot figure.

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



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

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



Scatter Plots and Constellation Diagrams

In this section...

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 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 in two ways: using a default object or by defining name-value pairs. For more information, see the reference page.

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.

```
hMod = comm.QPSKModulator;
```

Create a raised cosine transmit filter with an upsample rate of 16.

```
Rup = 16;  % upsampling rate
hRCTxFilter = comm.RaisedCosineTransmitFilter(...
    'Shape','Normal', ...
    'RolloffFactor',0.5, ...
    'FilterSpanInSymbols',Rup, ...
    'OutputSamplesPerSymbol',Rup);
```

Generate data symbols and apply QPSK modulation.

```
data = randi([0 3],100,1);
modData = step(hMod,data);
```

Create a constellation diagram and set the `SamplesPerSymbol` property to the upsampling rate of the signal.

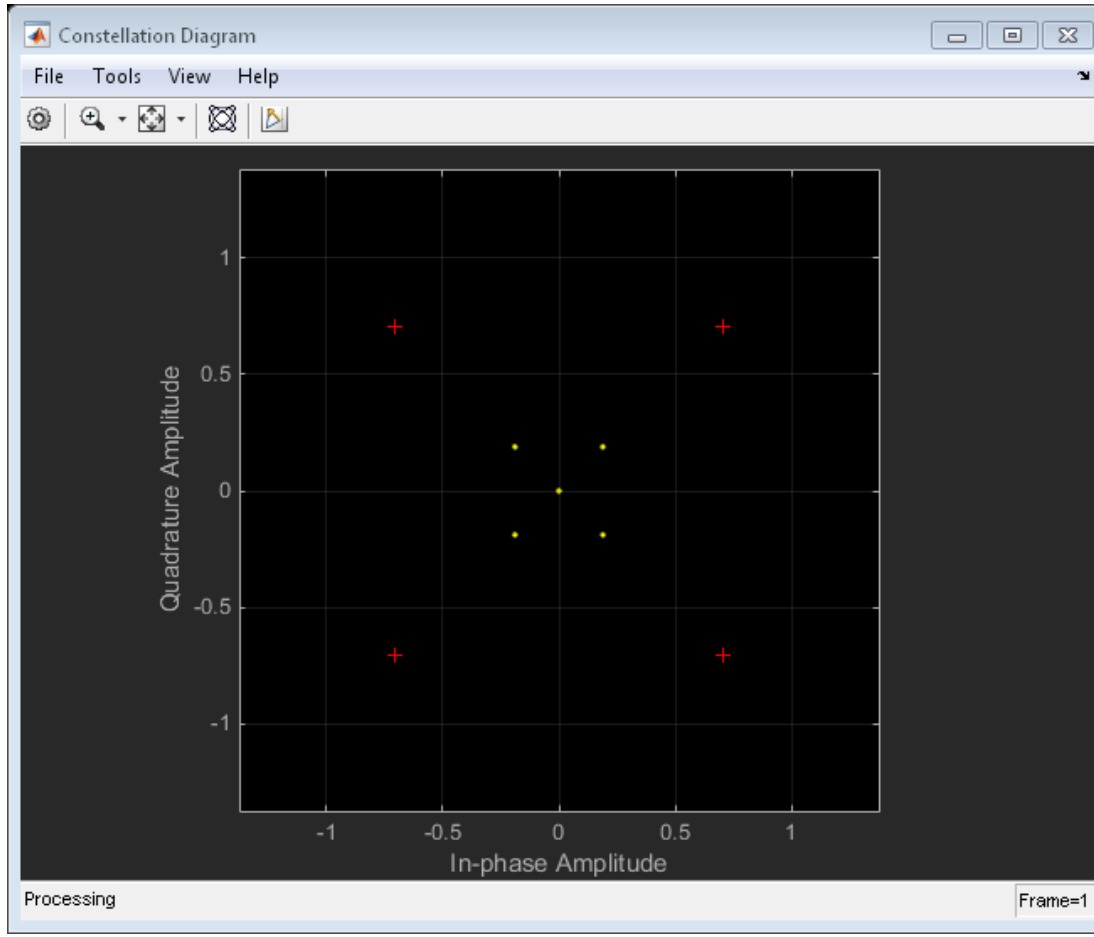
```
hScope = comm.ConstellationDiagram('SamplesPerSymbol',Rup);
```

Pass the modulated data through the raised cosine transmit filter.

```
txSig = step(hRCTxFilter,modData);
```

Display the constellation diagram of the transmitted signal.

```
step(hScope,txSig)
```



One way to create a better match between the two signals is to normalize the filter. Determine the required gain for the filter.

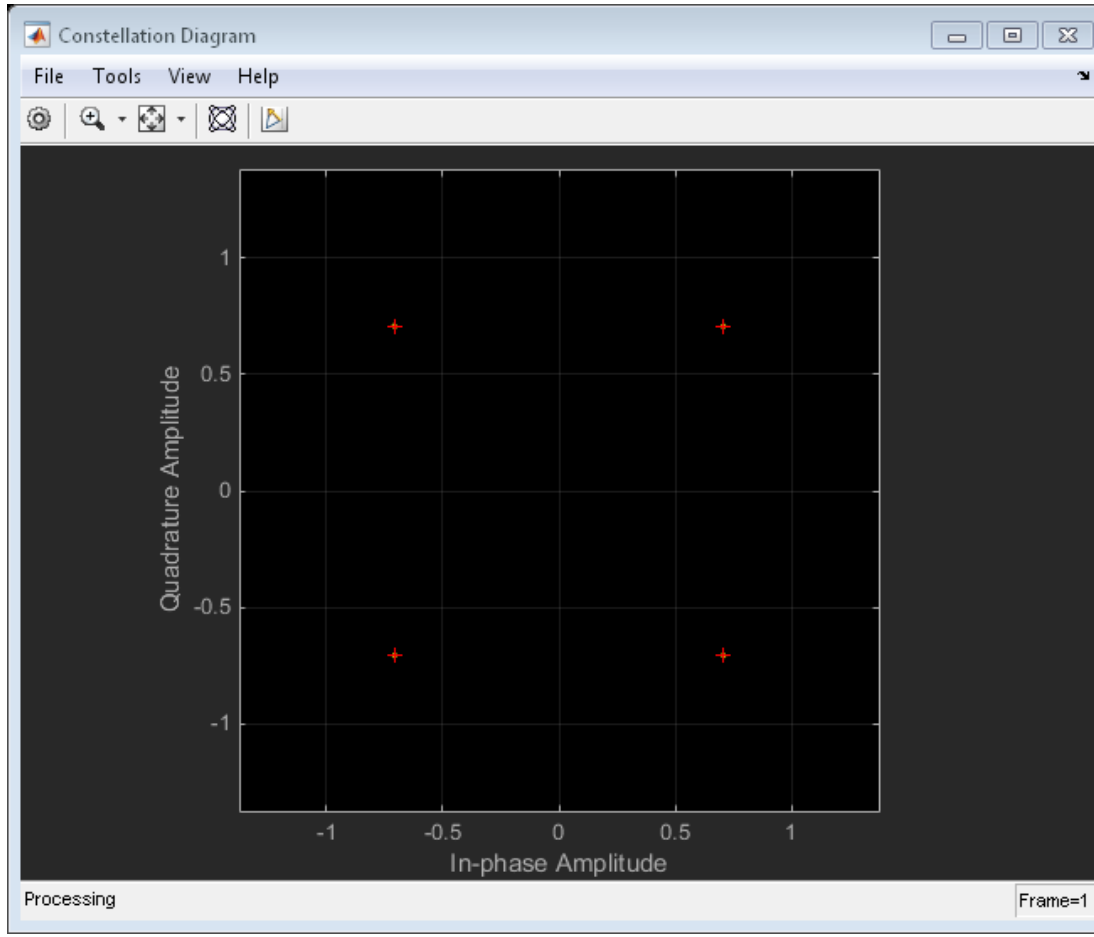
```
gain = 1/max(hRCTxFilter.coeffs.Numerator);
```

Apply a normalized filter the modulated signal.

```
txSig = gain*step(hRCTxFilter,modData);
```

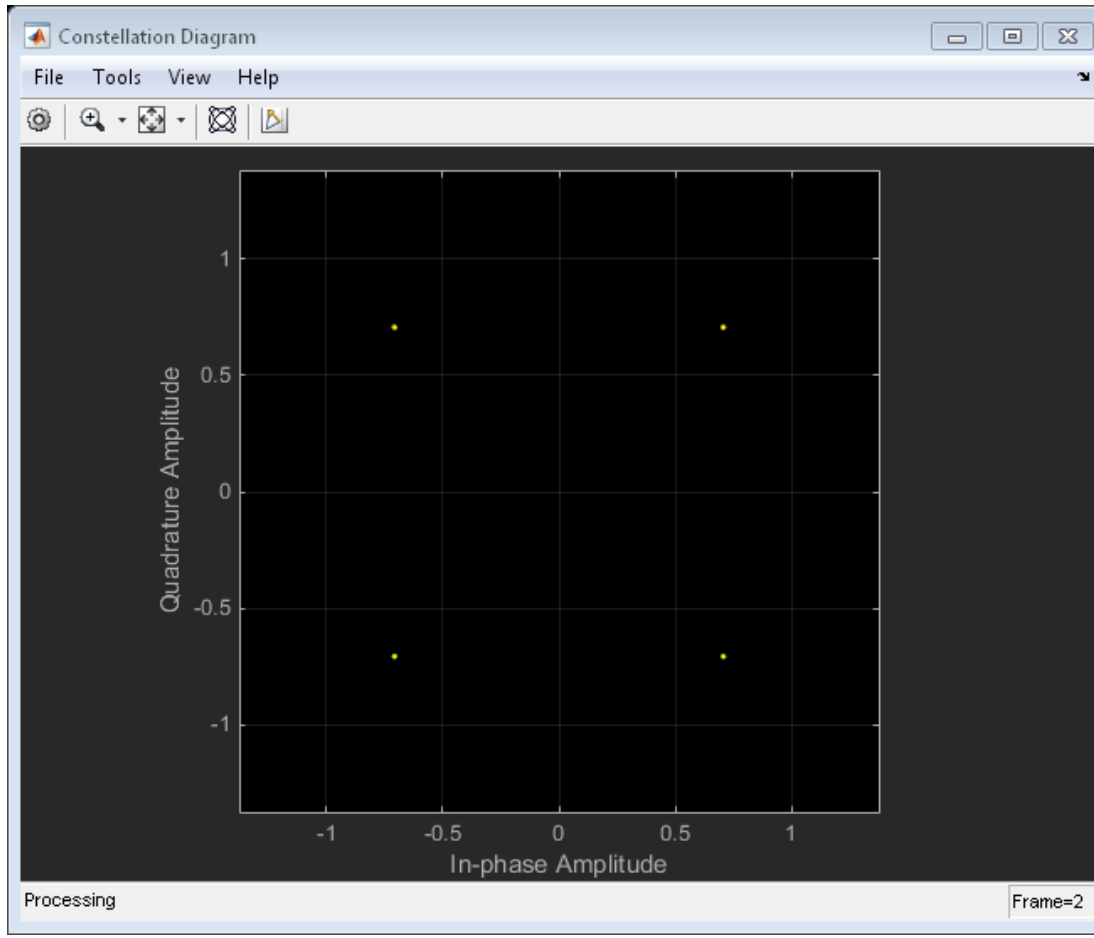
Release the constellation diagram, `hScope`. This is necessary because the input signal has been changed. Display the constellation diagram of the normalized signal. Observe that the data points and reference constellation overlay one another.

```
release(hScope)
step(hScope,txSig)
```



To view the transmitted signal more clearly, turn off the reference constellation by setting the `ShowReferenceConstellation` property to `false`.

```
hScope.ShowReferenceConstellation = false;
step(hScope,txSig)
```

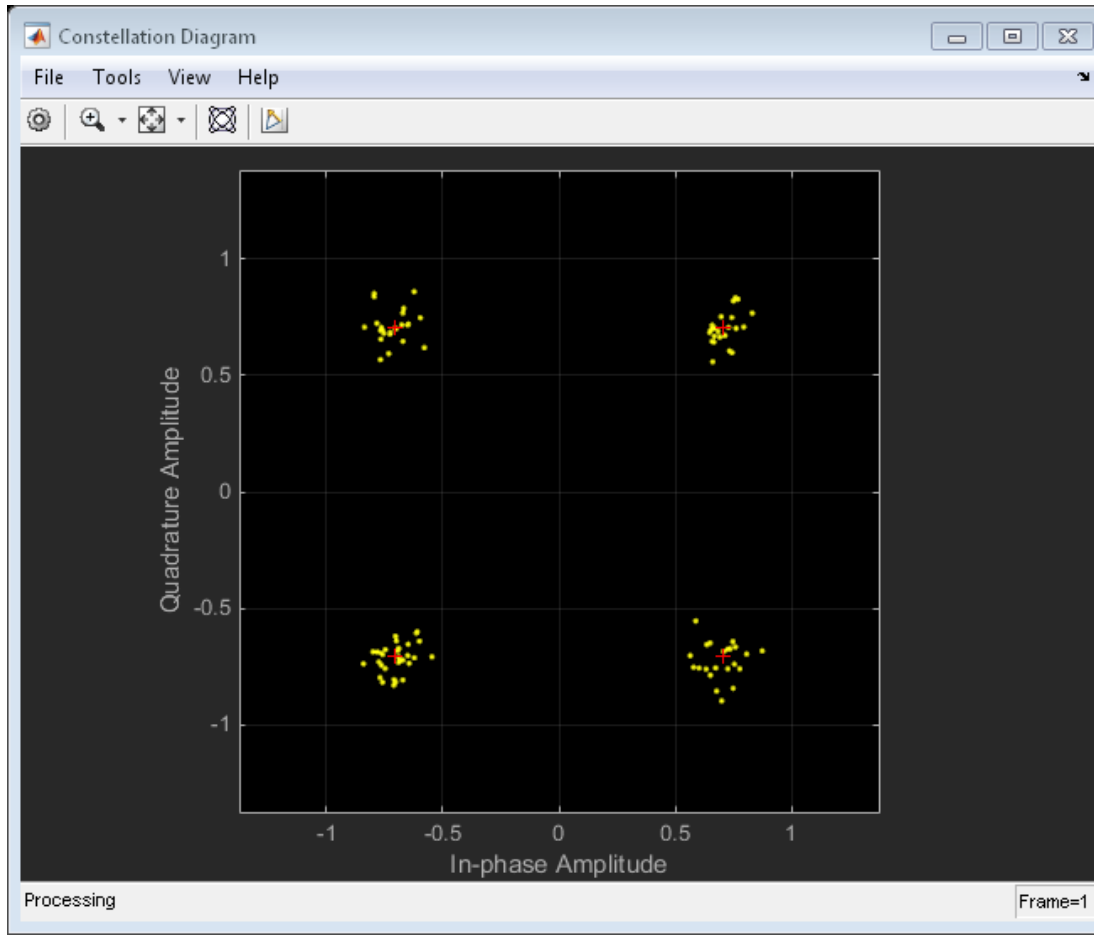


Create a noisy signal by Passing `txSig` through an AWGN channel.

```
rcv = awgn(txSig,20,'measured');
```

Turn on the reference constellation and display the received signal's constellation diagram.

```
release(hScope)
hScope.ShowReferenceConstellation = true;
step(hScope,rcv)
```



Illustrate How RF Impairments Distort Signal

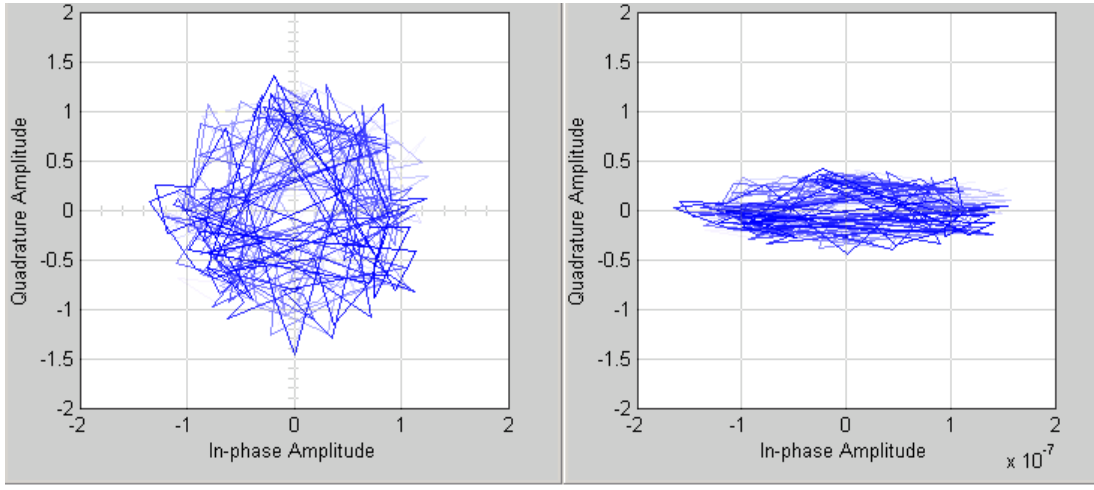
This example simulates RF impairments for a signal that was modulated using differential quaternary phase shift keying (DQPSK). Open the example model by typing `doc_receiverimpairments_dqpsk` at the MATLAB command line.

Overview of the Model

The model does the following:

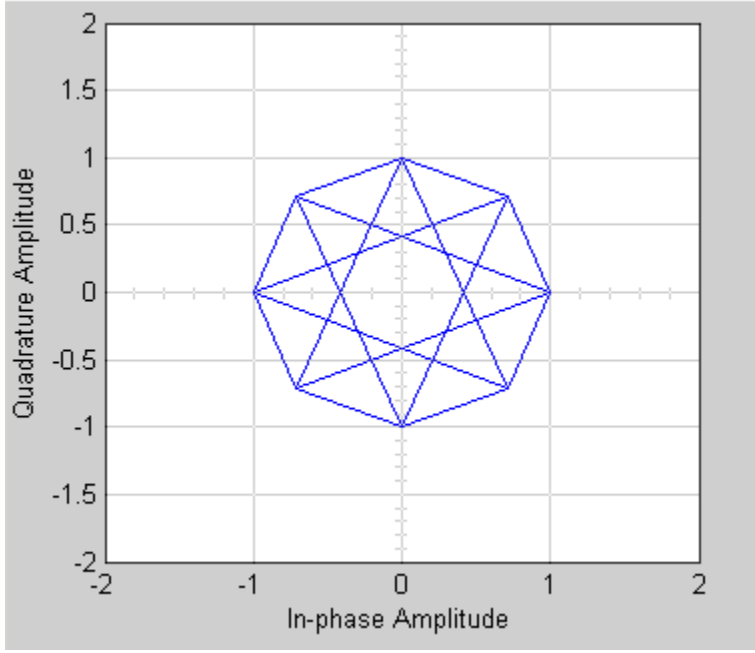
- Modulates a random signal using DQPSK modulation.
- Applies impairments to the signal using the blocks from the RF Impairments library.
- Forks the signal into two paths, and processes one path with an automatic gain control (AGC) to compensate for the free space path loss and the I/Q imbalance.
- Displays the trajectory of the signal with AGC and the trajectory of the signal without AGC.
- Demodulates both signals and calculates their error rates.

You can see the effect of the automatic gain by comparing the trajectories of the signals with and without AGC, as shown in the following figure.



Signal With (Left) and Without (Right) AGC

The trajectory of the signal with AGC more closely matches the undistorted trajectory for DQPSK, shown in the following figure, than does than the signal without AGC. Consequently, the error rate for the signal with AGC is much lower than the error rate for the signal without AGC.



In this example, the error rate for the demodulated signal without AGC is primarily caused by free space path loss and I/Q imbalance. The QPSK modulation minimizes the effects of the other impairments.

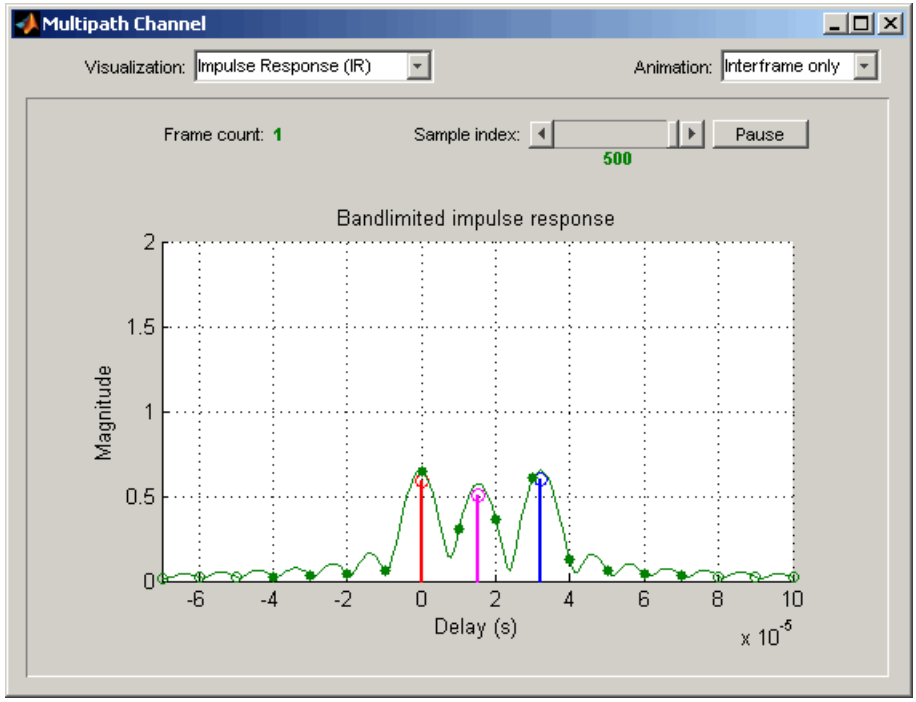
Channel Visualization

Communications System Toolbox software provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See “Fading Channels” for a description of fading channels and objects.

To open the channel visualization tool, type `plot(h)` at the command line, where `h` is a channel object that contains plot information. To populate a channel object with plot information, run a signal through it after setting its `StoreHistory` property to `true`.

For example, the following code opens the channel visualization tool showing a three-path Rayleigh channel through which a random signal is passed:

```
% Three-Path Rayleigh channel
h = rayleighchan(1/100000, 130, [0 1.5e-5 3.2e-5], [0, -3, -3]);
hMod = comm.DPSKModulator('ModulationOrder',2);
tx = randi([0 1],500,1);          % Random bit stream
dpskSig = step(hMod,tx);          % DPSK signal
% dpskSig = dpskmod(tx, 2);         % DPSK signal
h.StoreHistory = true;            % Allow states to be stored
y = filter(h, dpskSig);           % Run signal through channel
plot(h);                          % Call Channel Visualization Tool
```



The Channel Visualization GUI

The **Visualization** pull-down menu allows you to choose the visualization method. See "Visualization Options" on page 13-29 for details.

The **Frame count** counter shows the index of the current frame. It shows the number of frames processed by the filter method since the channel object was constructed or reset. A *frame* is a vector of M elements, interpreted to be M successive samples that are uniformly spaced in time, with a sample period equal to that specified for the channel.

The **Sample index** slider control indicates which channel snapshot is currently being displayed, while the **Pause** button pauses a running animation until you click it again. The slider control and **Pause** button apply to all visualizations except the `Doppler Spectrum`.

The **Animation** pull-down menu allows you to select how you want to display the channel snapshots within each frame. Setting this to `Slow` makes the tool show channel

snapshots in succession, starting at the sample set by the **Sample index** slider control. Selecting `Medium` or `Fast` makes the tool show fewer uniformly spaced snapshots, allowing you to go through the channel snapshots more rapidly. Selecting `Interframe only` (the default selection) prevents automatic animation of snapshots within the same frame. The **Animation** menu applies to all visualizations except the `Doppler Spectrum`.

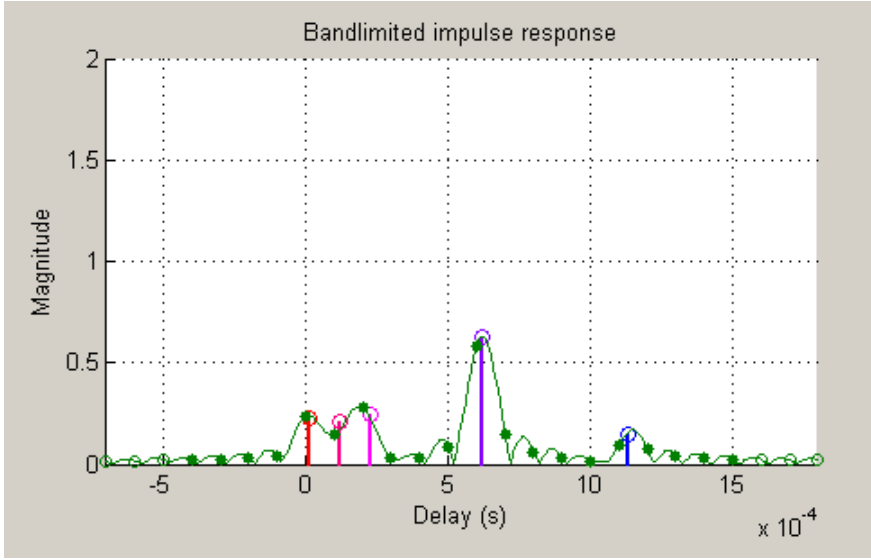
Visualization Options

The channel visualization tool plots the characteristics of a filter in various ways. Simply choose the visualization method from the **Visualization** menu, and the plot updates itself automatically.

The following visualization methods are currently available:

Impulse Response (IR)

This plot shows the magnitudes of two impulse responses: the multipath response (infinite bandwidth) and the bandlimited channel response.



The multipath response is represented by stems, each corresponding to one multipath component. The component with the smallest delay value is shown in red, and the component with the largest delay value is shown in blue. Components with intermediate delay values are shades between red and blue, becoming more blue for larger delays.

The bandlimited channel response is represented by the green curve. This response is the result of convolving the multipath impulse response, described above, with a sinc pulse of period, T, equal to the input signal's sample period.

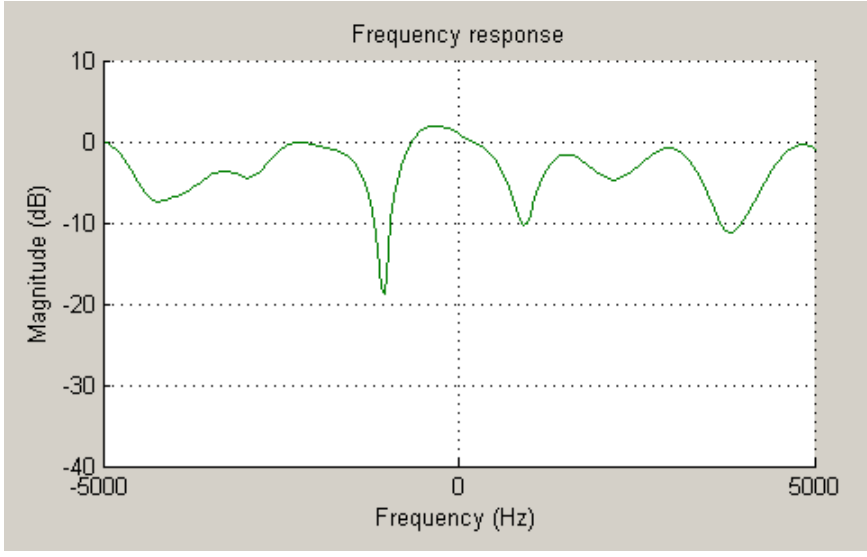
The solid green circles represent the channel filter response sampled at rate 1/T. The output of the channel filter is the convolution of the input signal (sampled at rate 1/T) with this discrete-time FIR channel filter response. For computational speed, the response is truncated.

The hollow green circles represent sample values not captured in the channel filter response that is used for processing the input signal.

Note that these impulse responses vary over time. You can use the slider to visualize how the impulse response changes over time for the current frame (i.e., input signal vector over time).

Frequency Response (FR)

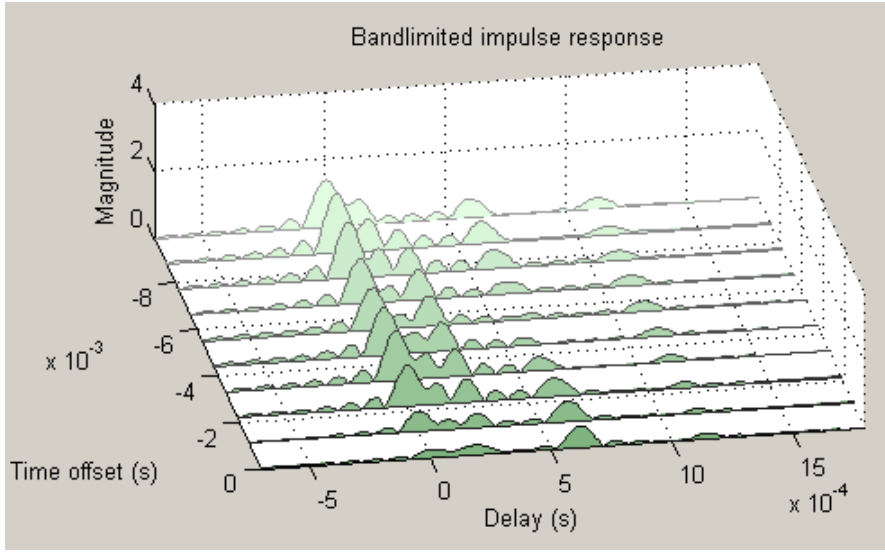
This plot shows the magnitude (in dB) of the frequency response of the multipath channel over the signal bandwidth.



As with the impulse response visualization, you can visualize how this frequency response changes over time.

IR Waterfall

This plot shows the evolution of the magnitude impulse response over time.

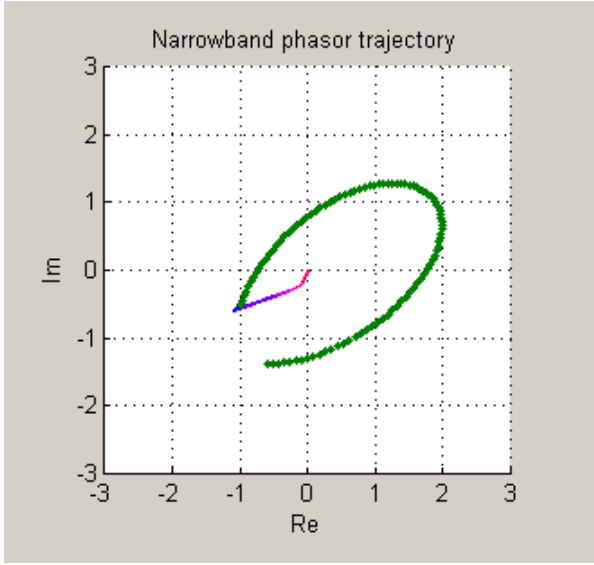


It shows 10 snapshots of the bandlimited channel impulse response within the last frame, with the darkest green curve showing the current response.

The time offset is the time of the channel snapshot relative to the current response time.

Phasor Trajectory

This plot shows phasors (vectors representing magnitude and phase) for each multipath component, using the same color code that was used for the impulse response plot.

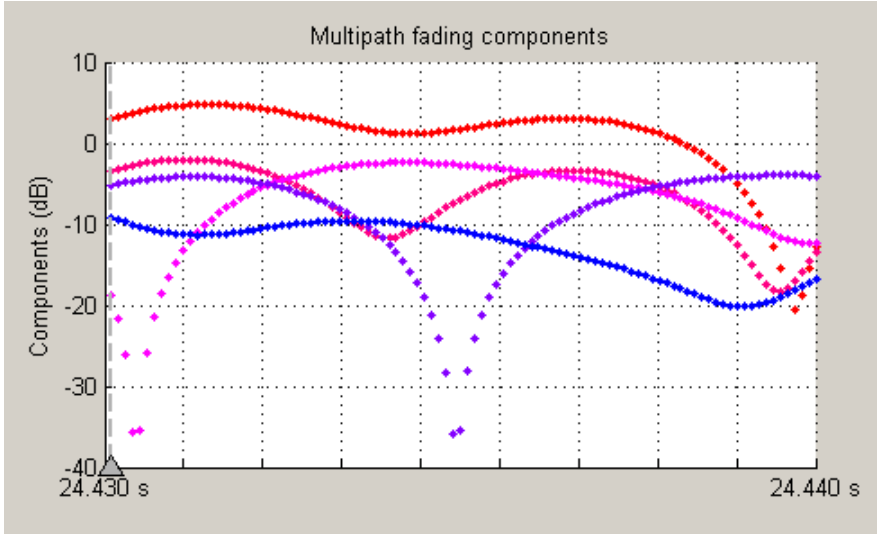


The phasors are connected end to end in order of path delay, and the trajectory of the resultant phasor is plotted as a green line. This resultant phasor is referred to as the *narrowband phasor*.

This plot can be used to determine the impact of the multipath channel on a narrowband signal. A narrowband signal is defined here as having a sample period much greater than the span of delays of the multipath channel (alternatively, a signal bandwidth much smaller than the coherence bandwidth of the channel). Thus, the multipath channel can be represented by a single complex gain, which is the sum of all the multipath component gains. When the narrowband phasor trajectory passes through or near the origin, it corresponds to a deep narrowband fade.

Multipath Components

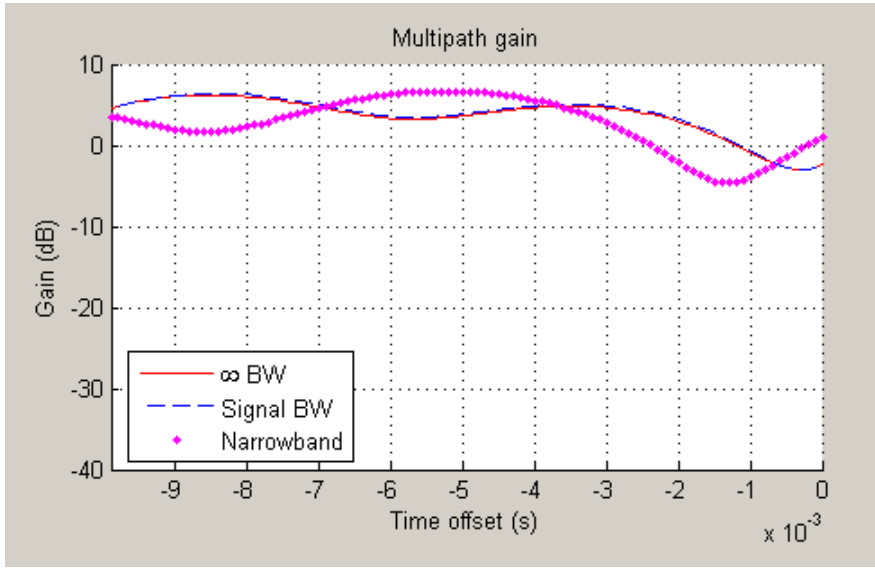
This plot shows the magnitudes of the multipath gains over time, using the same color code as that used for the multipath impulse response.



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

Multipath Gain

This plot shows the collective gains for the multipath channel for three signal bandwidths.



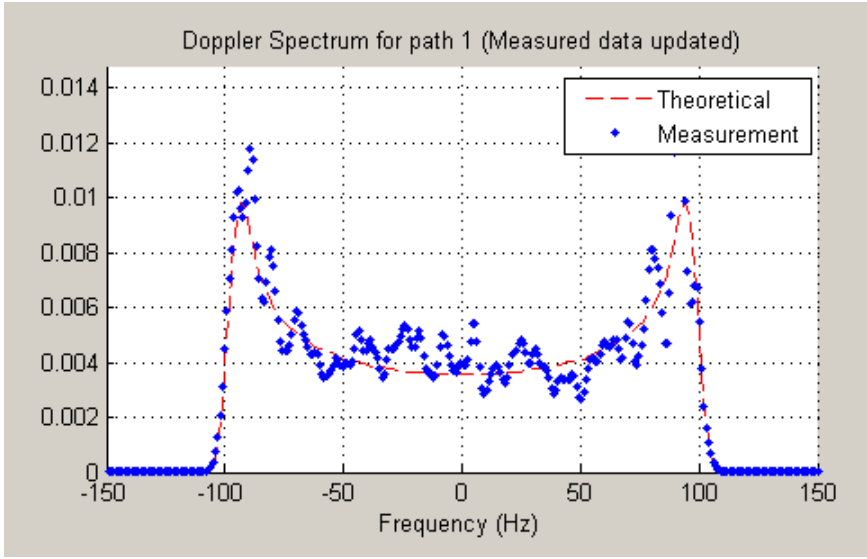
A collective gain is the sum of component magnitudes, as explained in the following:

- Narrowband (magenta dots): This is the magnitude of the narrowband phasor in the above trajectory plot. This curve is sometimes referred to as the *narrowband fading envelope*.
- Current signal bandwidth (dashed blue line): This is the sum of the magnitudes of the channel filter impulse response samples (the solid green dots in the impulse response plot). This curve represents the maximum signal energy that can be captured using a RAKE receiver. Its value (or metrics, such as theoretical BER, derived from it) is sometimes referred to as the *matched filter bound*.
- Infinite bandwidth (solid red line): This is the sum of the magnitudes of the multipath component gains.

In general, the variability of this multipath gain, or of the signal fading, decreases as signal bandwidth is increased, because multipath components become more resolvable. If the signal bandwidth curve roughly follows the narrowband curve, you might describe the signal as narrowband. If the signal bandwidth curve roughly follows the infinite bandwidth curve, you might describe the signal as wideband. With the right receiver, a wideband signal exploits the path diversity inherent in a multipath channel.

Doppler Spectrum

This plot shows up to two Doppler spectra.



The first Doppler spectrum, represented by the dashed red line, is a theoretical spectrum based on the Doppler filter response used in the multipath channel model. In the preceding plot, the theoretical Doppler spectrum used for the multipath channel model is known as the *Jakes spectrum*. Note that the plotted Doppler spectrum is normalized to have a total power of 1. This Doppler spectrum is used to determine a Doppler filter response. For practical purposes, the Doppler filter response is truncated, which has the effect of modifying the Doppler spectrum, as shown in the plot.

The second Doppler spectrum, represented by the blue dots, is determined by measuring the power spectrum of the multipath fading channel as the model generates path gains. This measurement is meaningful only after enough path gains have been generated. The title above the plot reports how many samples need to be processed through the channel before either the first Doppler spectrum or an updated spectrum can be plotted.

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

new Doppler spectrum will appear as soon as the processing of the current frame has ended.

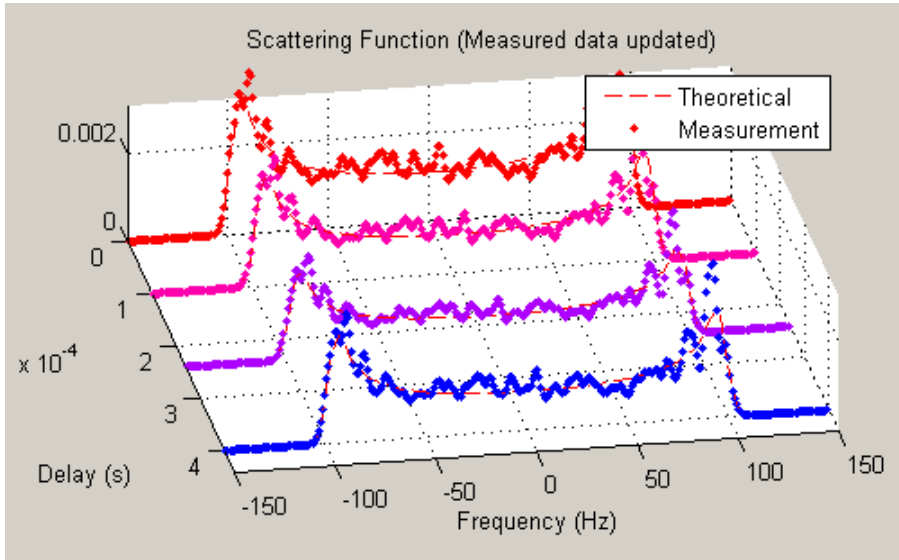
If the measured Doppler spectrum is a good approximation of the theoretical Doppler spectrum, the multipath channel model has generated enough fading gains to yield a reasonable representation of the channel statistics. For instance, if you want to determine the average BER of a communications link with a multipath channel and you want a statistically accurate measure of this average, you may want to ensure that the channel has processed enough samples to yield at least one Doppler spectrum measurement.

It is possible that a multipath channel (e.g., a Rician channel) can have both specular (line-of-sight) and diffuse components. In such a case, the Doppler spectrum would have both a line component and a wideband component. The channel visualization tool only shows the wideband component for the Doppler spectrum.

Unlike other visualizations, the Doppler spectrum visualization does not support animation. Because there is no intraframe data to plot, the visualization tool only updates the channel statistics at the end of each frame and therefore cannot pause in the middle of a frame. If you switch to the Doppler spectrum visualization from a different visualization that is in pause mode, the **Pause** button is subsequently disabled. Disabling pause avoids interaction problems between the Doppler spectrum visualization and other animation-style visualizations.

Scattering Function

This plot shows the Doppler spectra of each path versus the path delays, using the same color code as that used for the multipath impulse response.



The principle of operation of the Scattering Function plot is similar to that of the Doppler Spectrum plot. The main difference is that the Doppler spectra on this plot are not normalized as they are on the Doppler Spectrum plot, in order to better visualize the power delay profile.

Composite Plots

Several composite plots are also available. These are chosen by selecting the following from the **Visualization** pull-down menu:

- `IR and FR` for impulse response and frequency response plots.
- `Components and Gain` for multipath components and multipath gain plots.
- `Components and IR` for multipath components and impulse response plots.
- `Components, IR, and Phasor` for multipath components, impulse response, and phasor trajectory plots.

Visualize Samples Within a Frame

This example shows how to visualize samples within a frame through animation. The following lines of code create a Rayleigh channel and open the channel visualization tool:

```
% Create a fast fading channel
```

```
h = rayleighchan(1e-4, 100, [0 1.1e-4], [0 0]);

h.StoreHistory = 1;          % Allow states to be stored
y = filter(h, ones(100,1));  % Process samples through channel
plot(h);                     % Open channel visualization tool
```

After selecting a visualization option and a speed in the **Animation** menu, move the **Sample index** slider control all the way to the left and click **Resume**. The slider control moves by itself during animation. The sample index increments automatically to show which snapshot you are visualizing.

You can also move the slider control and glance through the samples of the frame as you like.

Animate Snapshots Across Frames

This example shows how to animate snapshots across frames. The following lines of code call the filter and plot methods within a loop to accomplish this:

```
Ts = 1e-4;   % Sample period (s)
fd = 100;    % Maximum Doppler shift

% Initialize DPSK modulator for M=4
hMod = comm.DPSKModulator(4);

% Path delay and gains
tau = [0.1 1.2 2.3 6.2 11.3]*Ts;
PdB = linspace(0, -10, length(tau)) - length(tau)/20;

nTrials = 10000;   % Number of trials
N = 100;           % Number of samples per frame

h = rayleighchan(Ts, fd, tau, PdB);  % Create channel object
h.NormalizePathGains = false;
h.ResetBeforeFiltering = false;
h.StoreHistory = 1;
h % Show channel object

% Channel fading simulation
for trial = 1:nTrials
    x = randi([0 3],10000,1);   % Random symbols
    dpskSig = step(hMod, x);    % Modulated symbols
    y = filter(h, dpskSig);     % Channel filter
```

```
    plot(h);                    % Plot channel response
    % The line below returns control to the command line in case
    % the GUI is closed while this program is still running
    if isempty(findobj('name', 'Multipath Channel')), break; end;
end
```

While the animation is running, you can move the slider control and change the sample index (which also makes the animation pause). After clicking **Resume**, the plot continues to animate.

The property `ResetBeforeFiltering` needs to be set to false so that the state information in the channel is not reset after the processing of each frame.

C Code Generation

Understanding C Code Generation

C Code Generation with the Simulink Coder Product

You can use the Communications System Toolbox, Simulink Coder, and Embedded Coder® products together to generate code that you can use to implement your model for a practical application. For instance, you can create an executable from your Simulink model to run on a target chip. This chapter introduces you to the basic concepts of code generation using these tools. For more information on code generation, see "Compiler or IDE Selection and Configuration".

Shared Library Dependencies

In general, the code you generate from Communications System Toolbox blocks is portable ANSI® C code. After you generate the code, you can deploy it on another machine. For more information on how to do so, see "Relocate Code to Another Development Environment" in the Simulink Coder documentation.

There are a few Communications System Toolbox blocks that generate code with limited portability. These blocks use precompiled shared libraries, such as DLLs, to support I/O for specific types of devices and file formats. To find out which blocks use precompiled shared libraries, open the Communications System Toolbox Block Support Table. You can identify blocks that use precompiled shared libraries by checking the footnotes listed in the **Code Generation Support** column of the table. All blocks that use shared libraries have the following footnote:

```
Host computer only. Excludes Real-Time Windows (RTWIN) target.
```

Simulink Coder provides functions to help you set up and manage the build information for your models. For example, one of the "Build Information Methods" that Simulink Coder provides is `getNonBuildFiles`. This function allows you to identify the shared libraries required by blocks in your model. If your model contains any blocks that use precompiled shared libraries, you can install those libraries on the target system. The folder that you install the shared libraries in must be on the system path. The target system does not need to have MATLAB installed, but it does need to be supported by MATLAB.

Highly Optimized Generated ANSI C Code

Communications System Toolbox blocks generate highly optimized ANSI C code. This C code is often suitable for embedded applications, and includes the following optimizations:

- **Function reuse (run-time libraries)** — The generated code reuses common algorithmic functions via calls to shared utility functions. Shared utility functions are highly optimized ANSI/ISO C functions that implement core algorithms such as FFT and convolution.
- **Parameter reuse (Simulink Coder run-time parameters)** — In many cases, if there are multiple instances of a block that all have the same value for a specific parameter, each block instance points to the same variable in the generated code. This process reduces memory requirements.
- **Blocks have parameters that affect code optimization** — Various blocks, such as the FFT and Sine Wave blocks, have parameters that enable you to optimize the simulation for memory or for speed. These optimizations also apply to code generation.
- **Other optimizations** — Use of contiguous input and output arrays, reusable inputs, overwritable arrays, and inlined algorithms provide smaller generated C code that is more efficient at run time.

C Code Generation from MATLAB

What is C Code Generation from MATLAB?

Code generation from MATLAB is a restricted subset of the MATLAB language that provides optimizations for:

- Generating efficient, production-quality C/C++ code and MEX files for deployment in desktop and embedded applications. For embedded targets, the subset restricts MATLAB semantics to meet the memory and data type requirements of the target environments.
- Accelerating fixed-point algorithms

Code generation from MATLAB supports the Communications System Toolbox functions listed in "Communications System Toolbox". You must have the DSP System Toolbox software installed to use this feature. To generate C code, you must have the MATLAB Coder software. If you have the Fixed-Point Designer, you can use `fiaccel` to generate MEX code for fixed-point applications.

In order to use Communications System Toolbox you must have a Signal Processing Toolbox license. There are a number of differences between the use of Signal Processing Toolbox functions with code generation from MATLAB and the use of these functions in the Signal Processing Toolbox software. These differences are summarized in "Specifying Inputs in Code Generation for MATLAB" and illustrated in "Code Generation Examples".

To follow the examples in this documentation:

- To generate C/C++ code with `codegen`, install the MATLAB Coder software, the Signal Processing Toolbox, the DSP System Toolbox, and a C compiler. For the Windows® platform, MATLAB supplies a default C compiler. Run `mex -setup` at the MATLAB command prompt to set up the C compiler.
- Change to a folder where you have write permission.

C Code Generation with System Objects and Functions

The following System objects and functions support code generation in MATLAB via the MATLAB Coder product. To use the `codegen` function, you must have a MATLAB Coder license.

Name	Remarks and Limitations
Input and Output	
`comm.BarkerCode`	"System Objects in MATLAB Code Generation"
`comm.GoldSequence`	"System Objects in MATLAB Code Generation"
`comm.HadamardCode`	"System Objects in MATLAB Code Generation"
`comm.KasamiSequence`	"System Objects in MATLAB Code Generation"
`comm.WalshCode`	"System Objects in MATLAB Code Generation"
`comm.PNSequence`	"System Objects in MATLAB Code Generation"
`lteZadoffChuSeq`	—
Signal and Delay Management	
`bi2de`	—
`de2bi`	—
Display and Visual Analysis	
`comm.ConstellationDiagram`	"System Objects in MATLAB Code Generation"
`dsp.ArrayPlot`	"System Objects in MATLAB Code Generation"
`dsp.SpectrumAnalyzer`	"System Objects in MATLAB Code Generation"
`dsp.TimeScope`	"System Objects in MATLAB Code Generation"
Source Coding	
`comm.DifferentialDecoder`	"System Objects in MATLAB Code Generation"
`comm.DifferentialEncoder`	"System Objects in MATLAB Code Generation"
Cyclic Redundancy Check Coding	
`comm.CRCDetector`	"System Objects in MATLAB Code Generation"
`comm.CRCGenerator`	"System Objects in MATLAB Code Generation"
`comm.HDLCRCDetector`	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
comm.HDLCRCGenerator	“System Objects in MATLAB Code Generation”
BCH Codes	
comm.BCHDecoder	“System Objects in MATLAB Code Generation”
comm.BCHEncoder	“System Objects in MATLAB Code Generation”
Reed-Solomon Codes	
comm.RSDecoder	“System Objects in MATLAB Code Generation”
comm.RSEncoder	“System Objects in MATLAB Code Generation”
comm.HDLRSDecoder	“System Objects in MATLAB Code Generation”
comm.HDLRSEncoder	“System Objects in MATLAB Code Generation”
LDPC Codes	
comm.LDPCDecoder	“System Objects in MATLAB Code Generation”
comm.LDPCEncoder	“System Objects in MATLAB Code Generation”
Convolutional Coding	
comm.APPDecoder	“System Objects in MATLAB Code Generation”
comm.ConvolutionalEncoder	“System Objects in MATLAB Code Generation”
comm.TurboDecoder	“System Objects in MATLAB Code Generation”
comm.TurboEncoder	“System Objects in MATLAB Code Generation”
comm.ViterbiDecoder	“System Objects in MATLAB Code Generation”
istrellis	—
poly2trellis	—
Signal Operations	
comm.Descrambler	“System Objects in MATLAB Code Generation”
comm.Scrambler	“System Objects in MATLAB Code Generation”
Interleaving	
comm.AlgebraicDeinterleaver	“System Objects in MATLAB Code Generation”
comm.AlgebraicInterleaver	“System Objects in MATLAB Code Generation”
comm.BlockDeinterleaver	“System Objects in MATLAB Code Generation”
comm.BlockInterleaver	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
comm.ConvolutionalDeinterleaver	"System Objects in MATLAB Code Generation"
comm.ConvolutionalInterleaver	"System Objects in MATLAB Code Generation"
comm.HelicalDeinterleaver	"System Objects in MATLAB Code Generation"
comm.HelicalInterleaver	"System Objects in MATLAB Code Generation"
comm.MatrixDeinterleaver	"System Objects in MATLAB Code Generation"
comm.MatrixInterleaver	"System Objects in MATLAB Code Generation"
comm.MatrixHelicalScanDeinterleaver	"System Objects in MATLAB Code Generation"
comm.MatrixHelicalScanInterleaver	"System Objects in MATLAB Code Generation"
comm.MultiplexedDeinterleaver	"System Objects in MATLAB Code Generation"
comm.MultiplexedInterleaver	"System Objects in MATLAB Code Generation"
Frequency Modulation	
comm.FSKDemodulator	"System Objects in MATLAB Code Generation"
comm.FSKModulator	"System Objects in MATLAB Code Generation"
Phase Modulation	
comm.BPSKDemodulator	"System Objects in MATLAB Code Generation"
comm.BPSKModulator	"System Objects in MATLAB Code Generation"
comm.DBPSKDemodulator	"System Objects in MATLAB Code Generation"
comm.DBPSKModulator	"System Objects in MATLAB Code Generation"
comm.DPSKDemodulator	"System Objects in MATLAB Code Generation"
comm.DPSKModulator	"System Objects in MATLAB Code Generation"
comm.DQPSKDemodulator	"System Objects in MATLAB Code Generation"
comm.DQPSKModulator	"System Objects in MATLAB Code Generation"
comm.OQPSKDemodulator	"System Objects in MATLAB Code Generation"
comm.OQPSKModulator	"System Objects in MATLAB Code Generation"
comm.PSKDemodulator	"System Objects in MATLAB Code Generation"
comm.PSKModulator	"System Objects in MATLAB Code Generation"
comm.QPSKDemodulator	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
comm.QPSKModulator	“System Objects in MATLAB Code Generation”
Amplitude Modulation	
comm.GeneralQAMDemodulator	“System Objects in MATLAB Code Generation”
comm.GeneralQAMModulator	“System Objects in MATLAB Code Generation”
comm.PAMDemodulator	“System Objects in MATLAB Code Generation”
comm.PAMModulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMDemodulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMModulator	“System Objects in MATLAB Code Generation”
Continuous Phase Modulation	
comm.CPFSKDemodulator	“System Objects in MATLAB Code Generation”
comm.CPFSKModulator	“System Objects in MATLAB Code Generation”
comm.CPMDemodulator	“System Objects in MATLAB Code Generation”
comm.CPMModulator	“System Objects in MATLAB Code Generation”
comm.GMSKDemodulator	“System Objects in MATLAB Code Generation”
comm.GMSKModulator	“System Objects in MATLAB Code Generation”
comm.MSKDemodulator	“System Objects in MATLAB Code Generation”
comm.MSKModulator	“System Objects in MATLAB Code Generation”
Trellis Coded Modulation	
comm.GeneralQAMTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.GeneralQAMTCMModulator	“System Objects in MATLAB Code Generation”
comm.PSKTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.PSKTCMModulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMTCMModulator	“System Objects in MATLAB Code Generation”
Orthogonal Frequency-Division Modulation	
comm.OFDMDemodulator	“System Objects in MATLAB Code Generation”
comm.OFDMModulator	“System Objects in MATLAB Code Generation”
Filtering	

Name	Remarks and Limitations
comm.IntegrateAndDumpFilter	"System Objects in MATLAB Code Generation"
comm.RaisedCosineReceiveFilter	"System Objects in MATLAB Code Generation"
comm.RaisedCosineTransmitFilter	"System Objects in MATLAB Code Generation"
Carrier Phase Synchronization	
comm.CPMCarrierPhaseSynchronizer	"System Objects in MATLAB Code Generation"
comm.PSKCarrierPhaseSynchronizer	"System Objects in MATLAB Code Generation"
Timing Phase Synchronization	
comm.EarlyLateGateTimingSynchronizer	"System Objects in MATLAB Code Generation"
comm.GardnerTimingSynchronizer	"System Objects in MATLAB Code Generation"
comm.GMSKTimingSynchronizer	"System Objects in MATLAB Code Generation"
comm.MSKTimingSynchronizer	"System Objects in MATLAB Code Generation"
comm.MuellerMullerTimingSynchronizer	"System Objects in MATLAB Code Generation"
Synchronization Utilities	
comm.DiscreteTimeVCO	"System Objects in MATLAB Code Generation"
Equalization	
comm.MLSEEqualizer	"System Objects in MATLAB Code Generation"
MIMO	
comm.LTEMIMOChannel	"System Objects in MATLAB Code Generation"
comm.MIMOChannel	"System Objects in MATLAB Code Generation"
comm.OSTBCCombiner	"System Objects in MATLAB Code Generation"
comm.OSTBCEncoder	"System Objects in MATLAB Code Generation"
comm.SphereDecoder	"System Objects in MATLAB Code Generation"
Channel Modeling and RF Impairments	
comm.AGC	"System Objects in MATLAB Code Generation"
comm.AWGNChannel	"System Objects in MATLAB Code Generation"
comm.BinarySymmetricChannel	"System Objects in MATLAB Code Generation"
comm.IQImbalanceCompensator	"System Objects in MATLAB Code Generation"
comm.LTEMIMOChannel	"System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
comm.MemorylessNonlinearity	“System Objects in MATLAB Code Generation”
comm.MIMOChannel	“System Objects in MATLAB Code Generation”
comm.PhaseFrequencyOffset	“System Objects in MATLAB Code Generation”
comm.PhaseNoise	“System Objects in MATLAB Code Generation”
comm.RayleighChannel	“System Objects in MATLAB Code Generation”
comm.RicianChannel	“System Objects in MATLAB Code Generation”
comm.ThermalNoise	“System Objects in MATLAB Code Generation”
comm.PSKCoarseFrequencyEstimator	“System Objects in MATLAB Code Generation”
comm.QAMCoarseFrequencyEstimator	“System Objects in MATLAB Code Generation”
iqcoef2imbal	—
iqimbal2coef	—
Measurements and Analysis	
comm.ACPR	“System Objects in MATLAB Code Generation”
comm.CCDF	“System Objects in MATLAB Code Generation”
comm.ErrorRate	“System Objects in MATLAB Code Generation”
comm.EVM	“System Objects in MATLAB Code Generation”
comm.MER	“System Objects in MATLAB Code Generation”

15

HDL Code Generation

What is HDL Code Generation?

You can use MATLAB and Simulink for rapid prototyping of hardware designs. Communication System Toolbox blocks and System objects provide support for HDL code generation when used with HDL Coder™. HDL Coder generates target-independent synthesizable Verilog and VHDL code for FPGA programming or ASIC prototyping and design.

Blocks With HDL Support

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

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 With HDL Support

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`
- `comm.RectangularQAMDemodulator`
- `comm.RectangularQAMModulator`

16

Simulation Acceleration

Simulation Acceleration Using GPUs

In this section...

GPU-Based System objects

GPU-based System objects look and behave much like the other System objects in the Communications System 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 System 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 `step` method 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 `step` method 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.

```
hTEnc = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

Create a BPSK Modulator System object.

```
hMod = comm.BPSKModulator;
```

Create an AWGN Channel System object.

```
hChan = 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.

```
hTDec = comm.gpu.TurboDecoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
      'NumIterations', 4);
```

Create an Error Rate System object.

```
hError = comm.ErrorRate;
```

Run the simulation by using the step method to process data.

```
for frmIdx = 1:8
 data = randi(s, [0 1], frmLen, 1);
 encodedData = step(hTEnc, data);
 modSignal = step(hMod, encodedData);
 receivedSignal = step(hChan, modSignal);
```

Convert the received signal to log-likelihood ratios for decoding.

```
receivedBits  = step(hTDec, (-2/(noiseVar/2))*real(receivedSignal));
```

Compare original the data to the received data and then calculate the error rate results.

```
errorStats = step(hError, 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;

hEnc = comm.LDPCEncoder;
gDec = comm.gpu.LDPCDecoder;
hDec = comm.LDPCDecoder;


msg = randi([0 1], 32400,2);

for ii=1:numframes,
    encout(:,ii) = step(hEnc, msg(:,ii));
end

%single ended to bipolar (for LLRs)
encout = 1-2*encout;

%Decode on the CPU
for ii=1:numframes;
    cout(:,ii) = step(hDec, encout(:,ii));
end

%Mulitframe decode on the GPU
gout = step(gDec, 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 GPU-based Viterbi Decoder System object. For a Viterbi Decoder, the frame size of your system cannot be inferred from an object property. Therefore, the `NumFrames` property defines the number of frames present in the input data.

```
numframes = 10;

hEnc = comm.ConvolutionalEncoder('TerminationMethod', 'Terminated');
hVit = comm.ViterbiDecoder('TerminationMethod', 'Terminated');

%Create a GPU Viterbi Decoder, using NumFrames property.
```

```
gVit = comm.gpu.ViterbiDecoder('TerminationMethod', 'Terminated', ...
                               'NumFrames', numframes );

msg = randi([0 1], 200, numframes);

for ii=1:numframes,
    convEncOut(:,ii) = 1-2*step(hEnc, msg(:,ii));
end

%Decode on the CPU
for ii=1:numframes;
    cVitOut(:,ii) = step(hVit, convEncOut(:,ii));
end

%Decode on the GPU
gVitOut = step(gVit, convEncOut(:));

isequal(gVitOut, cVitOut(:))
```

gpuArray and Regular MATLAB Numerical Arrays

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” in the Parallel Computing Toolbox documentation.

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 a GPU System object passes data to the `step` method in the form of a gpuArray, 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 to the step method.

In general, you should try to minimize the amount of data transfer between the CPU and the GPU in your simulation.

Pass gpuArray to Input of step Method

This example shows how to pass a gpuArray to the input of the `step` method, reducing latency.

```
h = comm.gpu.PSKModulator;
x = randi([0 7], 1000, 1, 'single');
gx = gpuArray(x);

o = step(h,x);
class(o)

release(h); %allow input types to change

go = step(h,gx);
class(go)
```

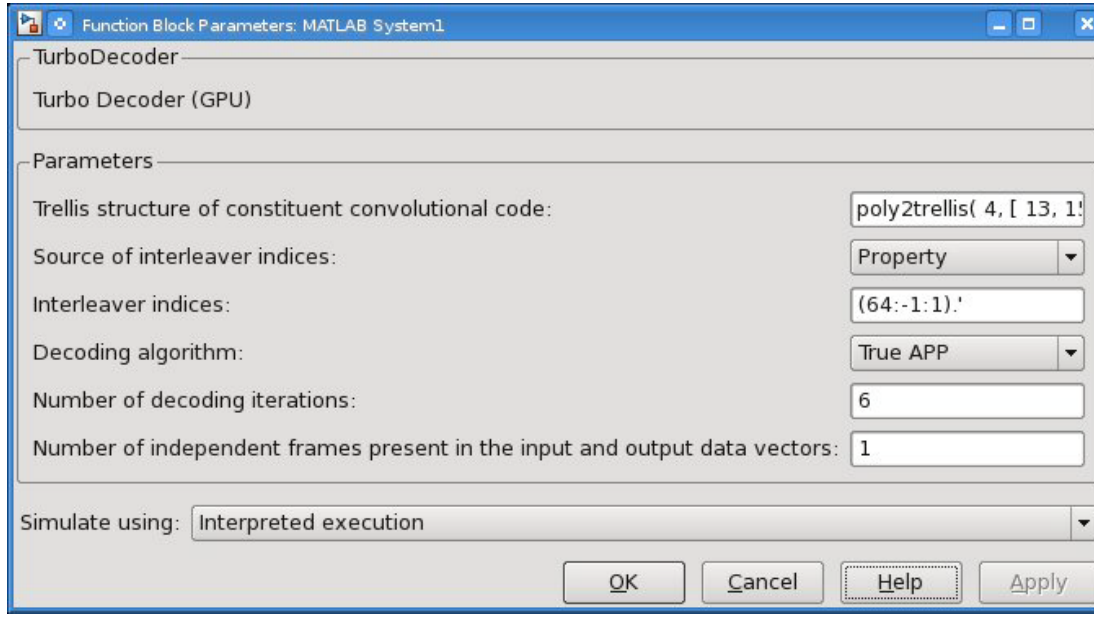
System Block Support for GPU System Objects

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`.



Define New System Objects

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

1. Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

   ```
   edit AddOne.m
   ```

2. Subclass your object from `matlab.System`. Insert this line as the first line of your file.

   ```
   classdef AddOne < matlab.System
   ```

3. Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access to `protected` because it is an internal method that users do not directly call or run.

   All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle.

   In this example, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables from occurring.

   By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

   ```
   methods (Access = protected)
      function y = stepImpl(~,x)
         y = x + 1;
      end
   end
   ```

Note: Instead of manually creating your class definition file, you can use an option on the **New > System Object** menu to open a template. The **Basic** template opens a simple

System object template. The **Advanced** template includes more advanced features of System objects, such as backup and restore. The **Simulink Extension** template includes additional customizations of the System object for use in the Simulink MATLAB System block. You then can edit the template file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic System Object

```
classdef AddOne < matlab.System
% ADDONE Compute an output value one greater than the input value

  % All methods occur inside a methods declaration.
  % The stepImpl method has protected access
  methods (Access = protected)

    function y = stepImpl(~,x)
      y = x + 1;
    end
  end
end
```

Related Examples

- “Change Number of Step Inputs or Outputs” on page 17-4

Change Number of Step Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you specify the inputs and outputs to the `stepImpl` method, you do not need to specify the `getNumInputsImpl` and `getNumOutputsImpl` methods. If you have a variable number of inputs or outputs (using varargin or varargout), include the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively, in your class definition file.

Note: You should only use `getNumInputsImpl` or `getNumOutputsImpl` methods to change the number of System object inputs or outputs. Do not use any other handle objects within a System object to change the number of inputs or outputs.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to specify two inputs and two outputs. You do not need to implement associated `getNumInputsImpl` or `getNumOutputsImpl` methods.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(~,x1,x2)
      y1 = x1 + 1
      y2 = x2 + 1;
   end
end
```

Update the Algorithm and Associated Methods

Update the `stepImpl` method to use varargin and varargout. In this case, you must implement the associated `getNumInputsImpl` and `getNumOutputsImpl` methods to specify two or three inputs and outputs.

```
methods (Access = protected)
   function varargout = stepImpl(obj,varargin)
      varargout{1} = varargin{1}+1;
      varargout{2} = varargin{2}+1;
      if (obj.numInputsOutputs = 3)
         varargout{3} = varargin{3}+1;
      end
```

```
    end

    function validatePropertiesImpl(obj)
        if ~((obj.numInputsOutputs == 2) ||...
             (obj.numInputsOutputs == 3))
            error('Only 2 or 3 input and outputs allowed.');
        end
    end

    function numIn = getNumInputsImpl(obj)
        numIn = 3;
        if (obj.numInputsOutputs == 2)
            numIn = 2;
        end
    end

    function numOut = getNumOutputsImpl(obj)
        numOut = 3;
        if (obj.numInputsOutputs == 2)
            numOut = 2;
        end
    end
end
```

Use this syntax to run the algorithm with two inputs and two outputs.

```
x1 = 3;
x2 = 7;
[y1,y2] = step(AddOne,x1,x2);
```

To change the number of inputs or outputs, you must release the object before rerunning it.

```
release(AddOne)
x1 = 3;
x2 = 7;
x3 = 10
[y1,y2,y3] = step(AddOne,x1,x2,x3);
```

Complete Class Definition File with Multiple Inputs and Outputs

```
 classdef AddOne < matlab.System
% ADDONE Compute output values one greater than the input values

    % This property is nontunable and cannot be changed
```

```
% after the setup or step method has been called.
properties (Nontunable)
   numInputsOutputs = 3;   % Default value
  end

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods (Access = protected)
   function varargout = stepImpl(obj,varargin)
      if (obj.numInputsOutputs = 2)
         varargout{1} = varargin{1}+1;
         varargout{2} = varargin{2}+1;
      else
         varargout{1} = varargin{1}+1;
         varargout{2} = varargin{2}+1;
         varargout{3} = varargin{3}+1;
      end
   end

   function validatePropertiesImpl(obj)
      if ~((obj.numInputsOutputs == 2) ||
           (obj.numInputsOutputs == 3))
         error('Only 2 or 3 input and outputs allowed.');
      end
   end

   function numIn = getNumInputsImpl(obj)
      numIn = 3;
      if (obj.numInputsOutputs == 2)
         numIn = 2;
      end
   end

   function numOut = getNumOutputsImpl(obj)
      numOut = 3;
      if (obj.numInputsOutputs == 2)
         numOut = 2;
      end
   end
end
```

```
end
```

Related Examples

- “Validate Property and Input Values” on page 17-8
- “Define Basic System Objects” on page 17-2

More About

- “System Object Input Arguments and ~ in Code Examples” on page 17-45

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using `set.`*`PropertyName`* syntax. In this case, the *`PropertyName`* is `Increment`.

```
methods
   % Validate the properties of the object
   function set.Increment(obj,val)
       if val >= 10
         error('The increment value must be less than 10');
       end
       obj.Increment = val;
   end
 end
```

This example shows how to validate the value of two interdependent properties using the `validatePropertiesImpl` method. In this case, the `UseIncrement` property value must be `true` and the `WrapValue` property value must be less than the `Increment` property value.

```
methods (Access = protected)
   function validatePropertiesImpl(obj)
       if obj.UseIncrement && obj.WrapValue < obj.Increment
         error('Wrap value must be less than increment value');
       end
   end
end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access = protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
```

```
end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
% ADDONE Compute an output value by incrementing the input value

  % All properties occur inside a properties declaration.
  % These properties have public access (the default)
  properties (Logical)
    UseIncrement = true
  end

  properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
  end

  methods
    % Validate the properties of the object
    function set.Increment(obj, val)
        if val >= 10
          error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
  end

  methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue < obj.Increment
          error('Wrap value must be less than increment value');
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
          error('Input must be numeric');
        end
    end

    function out = stepImpl(obj,in)
      if obj.UseIncrement
        out = in + obj.Increment;
```

```
      else
        out = in + 1;
      end
    end
  end
end
```

Note: All inputs default to variable-size inputs. See “Change Input Complexity or Dimensions” for more information.

Related Examples

- “Define Basic System Objects” on page 17-2

More About

-
- “Property Set Methods”
- “System Object Input Arguments and ~ in Code Examples” on page 17-45

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you call the step method.

Define Public Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable string, `default.bin`. Users cannot change *nontunable* properties after the `setup` method has been called. Refer to the Methods Timing section for more information.

```
properties (Nontunable)
  Filename = 'default.bin'
end
```

Define Private Properties to Initialize

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as *hidden* to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access = private)
  pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once during the first call to the `step` method. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
  function setupImpl(obj)
    obj.pFileID = fopen(obj.Filename,'wb');
    if obj.pFileID < 0
       error('Opening the file failed');
```

```
    end
  end
end
```

Although not part of setup, you should close files when your code is done using them. You use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step  method has been called.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
```

```
end
```

Related Examples

- “Release System Object Resources” on page 17-29
- “Define Property Attributes” on page 17-18

More About

-

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (`MyFile` in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. varargin indicates all of the object's public properties.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:});
   end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename ='default.bin' % the name of the file to create
        Access = 'wb' % The file access string (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
          setProperties(obj,nargin,varargin{:});
        end
```

```
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
end
```

See Also

`nargin`

Related Examples

- “Define Property Attributes” on page 17-18
- “Release System Object Resources” on page 17-29

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

`pCount` is an internal counter property of the System object `obj`. The user calls the `reset` method, which calls the `resetImpl` method. In this example , `pCount` resets to 0.

Note: When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
% Counter System object™ that increments a counter

   properties (Access = private)
      pCount
   end

   methods (Access = protected)
      % In step, increment the counter and return
      % its value as an output
      function c = stepImpl(obj)
          obj.pCount = obj.pCount + 1;
          c = obj.pCount;
      end

      % Reset the counter to zero.
      function resetImpl(obj)
         obj.pCount = 0;
      end
   end
```

```
end
```

See for more information.

More About

-

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

System object users cannot change nontunable properties after the `setup` or `step` method has been called. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
   InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter `1` or `0` or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use `1` or `0` as a default value.

```
properties (Logical)
   Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `pCount` is constrained to accept only real, positive integers. You cannot use sparse values. The following restriction applies to a property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
   Count
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute . Properties with this attribute display their state values when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or fi value, but not a scaled double fi value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, `Constant`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess = Public` by default
- Value set only using the `setupImpl` method or when the System object is locked during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
   Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
% Counter Increment a counter to a maximum value

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
```

```
    % The inital value of the counter
    InitialValue = 0
end
properties (Nontunable, PositiveInteger)
    % The maximum value of the counter
    MaxValue = 3
end

properties (Logical)
    % Whether to increment the counter
    Increment = true
end

properties (DiscreteState)
    % Count state variable
    Count
end

methods (Access = protected)
    % In step, increment the counter and return its value
    % as an output

    function c = stepImpl(obj)
        if obj.Increment && (obj.Count < obj.MaxValue)
            obj.Count = obj.Count + 1;
        else
            disp(['Max count, ' num2str(obj.MaxValue) ',reached'])
        end
        c = obj.Count;
    end

    % Setup the Count state variable
    function setupImpl(obj)
        obj.Count = 0;
    end

    % Reset the counter to one.
    function resetImpl(obj)
        obj.Count = obj.InitialValue;
    end
end
```

```
end
```

More About

- “Class Attributes”
- “Property Attributes”
-

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the `isInactivePropertyImpl` method to hide a property from displaying. If the `isInactiveProperty` method returns `true` to the property you pass in, then that property does not display.

```
methods (Access = protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

Complete Class Definition File with Hidden Inactive Property

```
classdef Counter < matlab.System
  % Counter Increment a counter

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
    % Allow the user to set the initial value
    UseRandomInitialValue = true
    InitialValue = 0
  end

  % The private count variable, which is tunable by default
  properties (Access = private)
    pCount
  end

  methods (Access = protected)
    % In step, increment the counter and return its value
    % as an output
    function c = stepImpl(obj)
      obj.pCount = obj.pCount + 1;
```

```
      c = obj.pCount;
    end

    % Reset the counter to either a random value or the initial
    % value.
    function resetImpl(obj)
      if obj.UseRandomInitialValue
        obj.pCount = rand();
      else
        obj.pCount = obj.InitialValue;
      end
    end

    % This method controls visibility of the object's properties
    function flag = isInactivePropertyImpl(obj,propertyName)
      if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
      else
        flag = false;
      end
    end
  end
end
```

Limit Property Values to Finite String Set

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

String sets use two related properties. You first specify the user-visible property name and default string value. Then, you specify the associated hidden property by appending "Set" to the property name. You must use a capital "S" in "Set."

In the "Set" property, you specify the valid string values as a cell array of the `matlab.system.Stringset` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
   Color = 'blue'
end

properties (Hidden,Transient)
   ColorSet = matlab.system.StringSet({'red','blue','green'});
end
```

Complete Class Definition File with String Set

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ illustrates the use of StringSets

   properties
     Color = 'blue'
   end

   properties (Hidden,Transient)
     % Let them choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]),randn([2,1]), ...
           'Color',obj.Color(1));
```

```
    end
    function releaseImpl(obj)
        cla(Whiteboard.getWhiteboard());
        hold on
    end
  end

  methods (Static)
    function a = getWhiteboard()
        h = findobj('tag','whiteboard');
        if isempty(h)
          h = figure('tag','whiteboard');
          hold on
        end
        a = gca;
    end
  end
end
```

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk  = Whiteboard;

% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color  = 'blue';

% Take a few steps
for i=1:3
  hGreenInk.step();
  hBlueInk.step();
end

%% Clear the whiteboard
hBlueInk.release();

%% Display System object used in this example
```

```
type('Whiteboard.m');
```

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods  (Access = protected)
   function processTunedPropertiesImpl(obj)
      obj.pLookupTable = obj.MiddleC * ...
        (1+log(1:obj.NumNotes)/log(12));
   end
end
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
  % TuningFork Illustrate the processing of tuned parameters
  %

  properties
    MiddleC = 440
    NumNotes = 12
  end

  properties (Access = private)
    pLookupTable
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.MiddleC = 440;
      obj.pLookupTable = obj.MiddleC * ...
         (1+log(1:obj.NumNotes)/log(12));
    end
```

```
    function hz = stepImpl(obj,noteShift)
      % A noteShift value of 1 corresponds to obj.MiddleC
      hz = obj.pLookupTable(noteShift);
    end

    function processTunedPropertiesImpl(obj)
      % Generate a lookup table of note frequencies
      obj.pLookupTable = obj.MiddleC * ...
         (1+log(1:obj.NumNotes)/log(12));
    end
  end
end
```

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
   function releaseImpl(obj)
      cla(Whiteboard.getWhiteboard());
      hold on
   end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ shows the use of StringSets
%
   properties
     Color = 'blue'
   end

   properties (Hidden)
     % Let user choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]), randn([2,1]), ...
           'Color',obj.Color(1));
     end

     function releaseImpl(obj)
         cla(Whiteboard.getWhiteboard());
         hold on
```

```
    end

  methods (Static)
    function a = getWhiteboard()
        h = findobj('tag','whiteboard');
        if isempty(h)
          h = figure('tag','whiteboard');
          hold on
        end
        a = gca;
    end
  end
end
```

Related Examples

- "Initialize Properties and Setup One-Time Calculations" on page 17-11

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the `pFir` and `pIir` property values.

```
properties (Nontunable, Access = private)
   pFir % Store the FIR filter
   pIir % Store the IIR filter
end

methods
   function obj = Filter(varargin)
      setProperties(obj,nargin,varargin{:});
      obj.pFir = FIR(obj.zero);
      obj.pIir = IIR(obj.pole);
   end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
% Filter System object with a single pole and a single zero
%
% This System object illustrates composition by
% composing an instance of itself.
%

    properties (Nontunable)
      zero = 0.01
      pole = 0.5
    end

    properties (Nontunable,Access = private)
      pZero % Store the FIR filter
      pPole % Store the IIR filter
```

```
    end

    methods
        function obj = Filter(varargin)
            setProperties(obj,nargin,varargin{:});
            % Create instances of FIR and IIR as
            % private properties
            obj.pZero = Zero(obj.zero);
            obj.pPole = Pole(obj.pole);
        end
    end

    methods (Access = protected)
        function setupImpl(obj,x)
            setup(obj.pZero,x);
            setup(obj.pPole,x);
        end

        function resetImpl(obj)
            reset(obj.pZero);
            reset(obj.pPole);
        end

        function y = stepImpl(obj,x)
            y = step(obj.pZero,x) + step(obj.pPole,x);
        end
        function releaseImpl(obj)
            release(obj.pZero);
            release(obj.pPole);
        end
    end
end
```

Class Definition File for IIR Component of Filter

```
classdef Pole < matlab.System

  properties
    Den = 1
  end

  properties (Access = private)
    tap = 0
  end
```

```
  methods
    function obj = Pole(varargin)
      setProperties(obj,nargin,varargin{:},'Den');
    end
  end

  methods (Access = protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Den;
      obj.tap = y;
    end
  end

end
```

Class Definition File for FIR Component of Filter

```
classdef Zero < matlab.System

  properties
    Num = 1
  end

  properties (Access = private)
    tap = 0
  end

  methods
    function obj = Zero(varargin)
      setProperties(obj,nargin,varargin{:},'Num');
    end
  end

  methods (Access = protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Num;
      obj.tap = x;
    end
  end

end
```

See Also

`nargin`

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
   matlab.system.mixin.FiniteSource
```

2 Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
  function bDone = isDoneImpl(obj)
     bDone = obj.NumSteps==2
  end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
  % RunTwice System object that runs exactly two times
  %
  properties (Access = private)
    NumSteps
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.NumSteps = 0;
    end

    function y = stepImpl(obj)
      if ~obj.isDone()
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
      else
        y = 0;
      end
    end

    function bDone = isDoneImpl(obj)
```

```
      bDone = obj.NumSteps==2;
    end
  end
end
```

More About

- “What Are Mixin Classes?” on page 17-46
- “Subclassing Multiple Classes”
- “System Object Input Arguments and ~ in Code Examples” on page 17-45

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods (Access = protected)
   function s = saveObjectImpl(obj)
      s = saveObjectImpl@matlab.System(obj);
      s.child = matlab.System.saveObject(obj.child);
      s.protected = obj.protected;
      s.pdependentprop = obj.pdependentprop;
      if isLocked(obj)
         s.state = obj.state;
      end
   end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

  properties (Access = private)
    child
    pdependentprop
  end

  properties (Access = protected)
    protected = rand;
  end

  properties (DiscreteState = true)
    state
  end

  properties (Dependent)
    dependentprop
  end
```

```
methods
  function obj = MySaveLoader(varargin)
    obj@matlab.System();
    setProperties(obj,nargin,varargin{:});
  end
end

methods (Access = protected)
  function setupImpl(obj)
    obj.state = 42;
  end

  function out = stepImpl(obj,in)
    obj.state = in;
    out = obj.state;
  end
end


% Serialization
methods (Access = protected)
  function s = saveObjectImpl(obj)
    % Call the base class method
    s = saveObjectImpl@matlab.System(obj);

    % Save the child System objects
    s.child = matlab.System.saveObject(obj.child);

    % Save the protected & private properties
    s.protected = obj.protected;
    s.pdependentprop = obj.pdependentprop;

    % Save the state only if object locked
    if isLocked(obj)
      s.state = obj.state;
    end
  end

  function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
```

```
        obj.protected = s.protected;
        obj.pdependentprop = s.pdependentprop;

        % Load the state only if object locked
        if wasLocked
          obj.state = s.state;
        end

        % Call base class method to load public properties
        loadObjectImpl@matlab.System(obj,s,wasLocked);
      end
    end
end
```

Related Examples

- “Load System Object” on page 17-39

Load System Object

This example shows how to load a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to assign the child object struct data to the associated object property. Assign protected and dependent property data to the associated object properties. If the object was locked when it was saved, assign the object's state to the associated property. Load the saved public properties with the `loadObjectImpl` method.

```
methods (Access = protected)
    function loadObjectImpl(obj,s,wasLocked)
      obj.child = matlab.System.loadObject(s.child);
      obj.protected = s.protected;
      obj.pdependentprop = s.pdependentprop;
      if wasLocked
        obj.state = s.state;
      end
      loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
   end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

  properties (Access = private)
    child
    pdependentprop
  end

  properties (Access = protected)
    protected = rand;
  end

  properties (DiscreteState = true)
    state
  end

  properties (Dependent)
```

```
    dependentprop
  end

  methods
    function obj = MySaveLoader(varargin)
      obj@matlab.System();
      setProperties(obj,nargin,varargin{:});
    end
  end

  methods (Access = protected)
    function setupImpl(obj)
      obj.state = 42;
    end

    function out = stepImpl(obj,in)
      obj.state = in;
      out = obj.state;
    end
  end


  % Serialization
  methods (Access = protected)
    function s = saveObjectImpl(obj)
      % Call the base class method
      s = saveObjectImpl@matlab.System(obj);

      % Save the child System objects
      s.child = matlab.System.saveObject(obj.child);

      % Save the protected & private properties
      s.protected = obj.protected;
      s.pdependentprop = obj.pdependentprop;

      % Save the state only if object locked
      if isLocked(obj)
        s.state = obj.state;
      end
    end

    function loadObjectImpl(obj,s,wasLocked)
      % Load child System objects
      obj.child = matlab.System.loadObject(s.child);
```

```
      % Load protected and private properties
      obj.protected = s.protected;
      obj.pdependentprop = s.pdependentprop;

      % Load the state only if object locked
      if wasLocked
        obj.state = s.state;
      end

      % Call base class method to load public properties
      loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
  end
end
```

Related Examples

- “Save System Object” on page 17-36

Clone System Object

This example shows how to clone a System object.

Clone System Object

You can define your own clone method, which is useful for copying objects without saving their state. The default `cloneImpl` method copies both a System object™ and its current state. If an object is locked, the default `cloneImpl` creates a cloned object that is also locked. An example of when you may want to write your own clone method is for cloning objects that handle resources. These objects cannot allocate resources twice and you would not want to save their states. To write your clone method, use the `saveObject` and `loadObject` methods to perform the clone within the `cloneImpl` method.

```
methods (Access = protected)
   function obj2 = cloneImpl(obj1)
      s = saveObject (obj1);
      obj2 = loadObject(s);
   end
end
```

Complete Class Definition File with Clone

```
classdef PassThrough < matlab.System
  methods (Access = protected)
    function y = stepImpl(~,u)
        y = u;
    end
    function obj2 = cloneImpl(obj1)
        s = matlab.System.saveObject(obj1);
        obj2 = matlab.System.loadObject(s);
    end
  end
end
```

Define System Object Information

This example shows how to define information to display for a System object.

Define System Object Info

You can define your own `info` method to display specific information for your System object. The default `infoImpl` method returns an empty struct. This `infoImpl` method returns detailed information when the `info` method is called using `info(x,'details')` or only count information if it is called using `info(x)`.

```
methods (Access = protected)
   function s = infoImpl(obj,varargin)
      if nargin>1 && strcmp('details',varargin(1))
         s = struct('Name','Counter',...
             'Properties', struct('CurrentCount', ...
              obj.pCount,'Threshold',obj.Threshold));
      else
         s = struct('Count',obj.pCount);
      end
   end
end
```

Complete Class Definition File with InfoImpl

```
classdef Counter < matlab.System
   % Counter Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = 0;
      end

      function resetImpl(obj)
         obj.Count = 0;
      end
```

```
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function s = infoImpl(obj,varargin)
            if nargin>1 && strcmp('details',varargin(1))
                s = struct('Name','Counter',...
                'Properties', struct('CurrentCount', ...
                obj.pCount,'Threshold',obj.Threshold));
            else
                s = struct('Count',obj.pCount);
            end
    end
end
```

System Object Input Arguments and ~ in Code Examples

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In many examples, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables.

What Are Mixin Classes?

Mixin classes are partial classes that you can combine in various combinations to form desired behaviors using multiple inheritance. System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

The following mixin classes are available for use with System objects.

- `matlab.system.mixin.CustomIcon` — Defines a block icon for System objects in the MATLAB System block
- `matlab.system.mixin.FiniteSource` — Adds the `isDone` method to System objects that are sources
- `matlab.system.mixin.Nondirect` — Allows the System object, when used in the MATLAB System block, to support nondirect feedthrough by making the runtime callback functions, `output` and `update` available
- `matlab.system.mixin.Propagates` — Enables System objects to operate in the MATLAB System block using the interpreted execution

Best Practices for Defining System Objects

A System object is a specialized kind of MATLAB object that is optimized for iterative processing. Use System objects when you need to call the `step` method multiple times or process data in a loop. When defining your own System object, use the following suggestions to help your code run efficiently.

- Define all one-time calculations in the `setupImpl` method and cache the results in a private property. Use the `stepImpl` method for repeated calculations.
- For parameters that do not change, define them in a locked object as `Nontunable` properties.
- If the number of System object inputs does not change, do not implement the `getNumInputsImpl` method. Also do not implement the `getNumInputsImpl` method when you explicitly list the inputs in the `stepImpl` method instead of using `varargin`. The same caveats apply to the outputs, `getNumOutputsImpl` and `varargout`.
- Variables that do not need to retain their values between calls should have local scope for that method.
- If properties are accessed more than once in the `stepImpl` method, or in the `updateImpl` and `outputImpl` methods, cache those properties as local variables inside the method. Iterative calculations using cached local variables run faster than calculations that must access the properties of an object. When the calculations for the method complete, you can save the local cached results back to the properties of that System object. Copy frequently used tunable properties into private properties.
- For best practices for including System objects in code generation, see “System Objects in MATLAB Code Generation”.

Orphan Pages

Modulation

"Digital Baseband Modulation"— Baseband digital modulation, including basic concepts, Frequency Modulation (FM), Phase Modulation (PM), Amplitude Modulation (AM), Continuous Phase Modulation (CPM), and Trellis-Coded Modulation (TCM).

"Analog Passband Modulation"— AM and FM analog passband modulation.

Utility Blocks

Block Name	Block Description
Align Signals Block	Align two signals by finding delay between them
Bipolar to Unipolar Converter	Map bipolar signal into unipolar signal in range [0, *M*-1]
Bit to Integer Converter	Map vector of bits to corresponding vector of integers
Complex Phase Difference	Output phase difference between two complex input signals
Complex Phase Shift	Shift phase of complex input signal by second input value
Data Mapper	Map integer symbols from one coding scheme to another
EVM Measurement	Calculate vector magnitude difference between ideal reference signal and measured signal
Find Delay Block	Find delay between two signals
Integer to Bit Converter	Map vector of integers to vector of bits
MER Measurement	Measure signal-to-noise ratio (SNR) in digital modulation applications
Unipolar to Bipolar Converter	Map unipolar signal in range [0, *M*-1] into bipolar signal