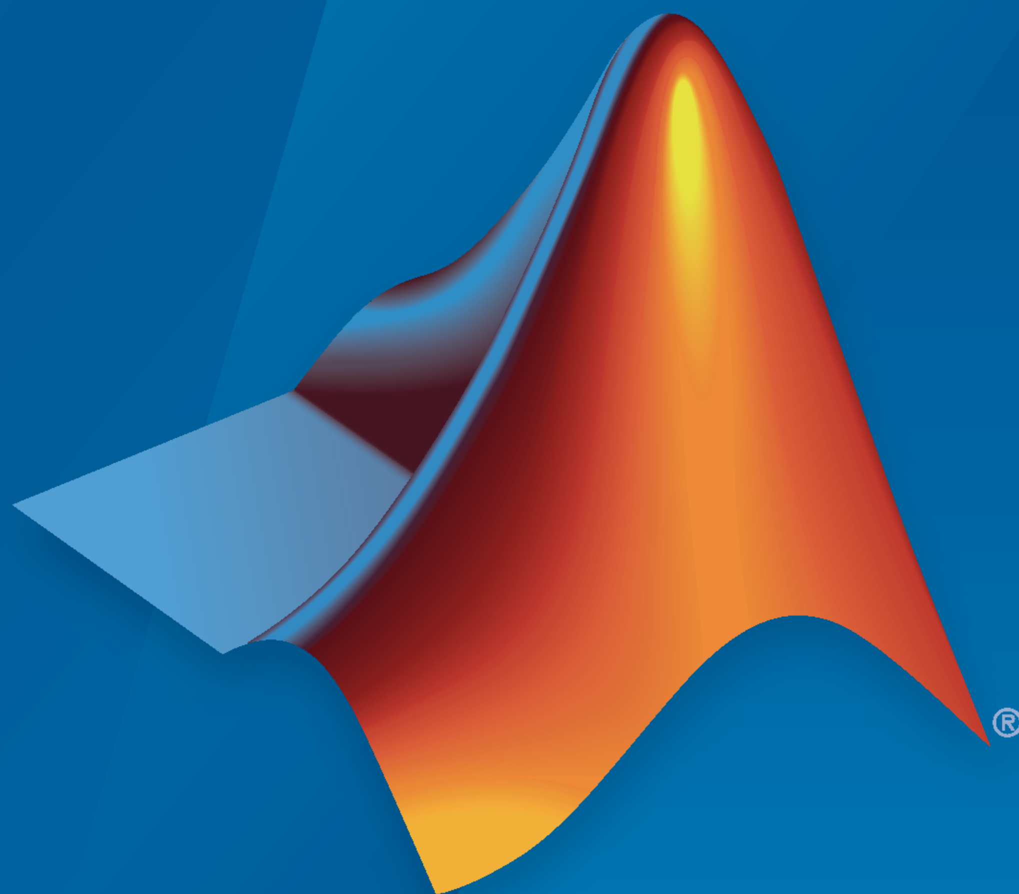
Communications Toolbox™

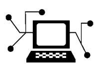
Getting Started Guide



MATLAB® & SIMULINK®

R2020a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Communications Toolbox™ Getting Started Guide

© COPYRIGHT 2011–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	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)
March 2015	Online only	Revised for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release 2016b)
March 2017	Online only	Revised for Version 6.4 (Release 2017a)
September 2017	Online only	Revised for Version 6.5 (Release 2017b)
March 2018	Online only	Revised for Version 6.6 (Release 2018a)
September 2018	Online only	Revised for Version 7.0 (Release 2018b)
March 2019	Online only	Revised for Version 7.1 (Release 2019a)
September 2019	Online only	Revised for Version 7.2 (Release 2019b)
March 2020	Online only	Revised for Version 7.3 (Release 2020a)

Contents

1 Introduction

Communications Toolbox Product Description **1-2**
Key Features **1-2**

Configure the Simulink Environment for Communications Models **1-3**
Communications Toolbox Simulink Model Template **1-3**
Block Characteristics **1-4**

Access the Block Libraries **1-5**

2 System Simulation

Examine 256-QAM Using Simulink **2-2**

Compute BER for QAM System with AWGN Using MATLAB **2-7**

Examine 16-QAM Using MATLAB **2-8**

Use Pulse Shaping on 16-QAM Signal **2-14**

Use Forward Error Correction on 16-QAM Signal **2-21**

QPSK and OFDM with MATLAB System Objects **2-24**

Accelerating BER Simulations Using the Parallel Computing Toolbox . . **2-27**

Iterative Design Workflow for Communication Systems **2-30**
Simulate a basic communications system **2-30**
Introduce convolutional coding and hard-decision Viterbi decoding **2-34**
Improve results using soft-decision decoding **2-37**
Use turbo coding to improve BER performance **2-40**
Apply a Rayleigh channel model **2-42**
Use OFDM-based equalization to correct multipath fading **2-44**
Use multiple antennas to further improve system performance **2-46**
Accelerate the simulation using MATLAB Coder **2-48**

What Is Different About Using Communications Toolbox Online? **2-51**

3 Visualization and Measurements

Scatter Plot and Eye Diagram with MATLAB Functions **3-2**

EVM and MER Measurements with Simulink **3-6**

ACPR and CCDF Measurements with MATLAB System Objects **3-13**
ACPR Measurements **3-13**
CCDF Measurements **3-15**

Introduction

- “Communications Toolbox Product Description” on page 1-2
- “Configure the Simulink Environment for Communications Models” on page 1-3
- “Access the Block Libraries” on page 1-5

Communications Toolbox Product Description

Design and simulate the physical layer of communications systems

Communications Toolbox provides algorithms and apps for the analysis, design, end-to-end simulation, and verification of communications systems. Toolbox algorithms including channel coding, modulation, MIMO, and OFDM enable you to compose and simulate a physical layer model of your standard-based or custom-designed wireless communications system.

The toolbox provides a waveform generator app, constellation and eye diagrams, bit-error-rate, and other analysis tools and scopes for validating your designs. These tools enable you to generate and analyze signals, visualize channel characteristics, and obtain performance metrics such as error vector magnitude (EVM). The toolbox includes SISO and MIMO statistical and spatial channel models. Channel profile options include Rayleigh, Rician, and WINNER II models. It also includes RF impairments, including RF nonlinearity and carrier offset and compensation algorithms, including carrier and symbol timing synchronizers. These algorithms enable you to realistically model link-level specifications and compensate for the effects of channel degradations.

Using Communications Toolbox with RF instruments or hardware support packages, you can connect your transmitter and receiver models to radio devices and verify your designs with over-the-air testing.

Key Features

- Algorithms for designing the physical layer of standard-based or custom-designed communications systems
- Waveform Generator app and analysis tools and measurement scopes, including a bit-error-rate app, constellation diagrams, and eye diagrams
- Channel models, including AWGN, multipath Rayleigh fading, Rician fading, MIMO multipath fading, and WINNER II spatial
- RF impairment models, including nonlinearity, phase noise, I/Q imbalance, thermal noise, and phase and frequency offsets
- Receiver components, including AGC, I/Q imbalance correction, DC blocking, and timing and carrier synchronization
- Hardware support packages for connecting waveforms to radio devices and verifying designs with over-the-air testing
- GPU-enabled algorithms for computationally intensive algorithms such as Turbo, LDPC, and Viterbi decoders

Configure the Simulink Environment for Communications Models

Communications Toolbox Simulink Model Template

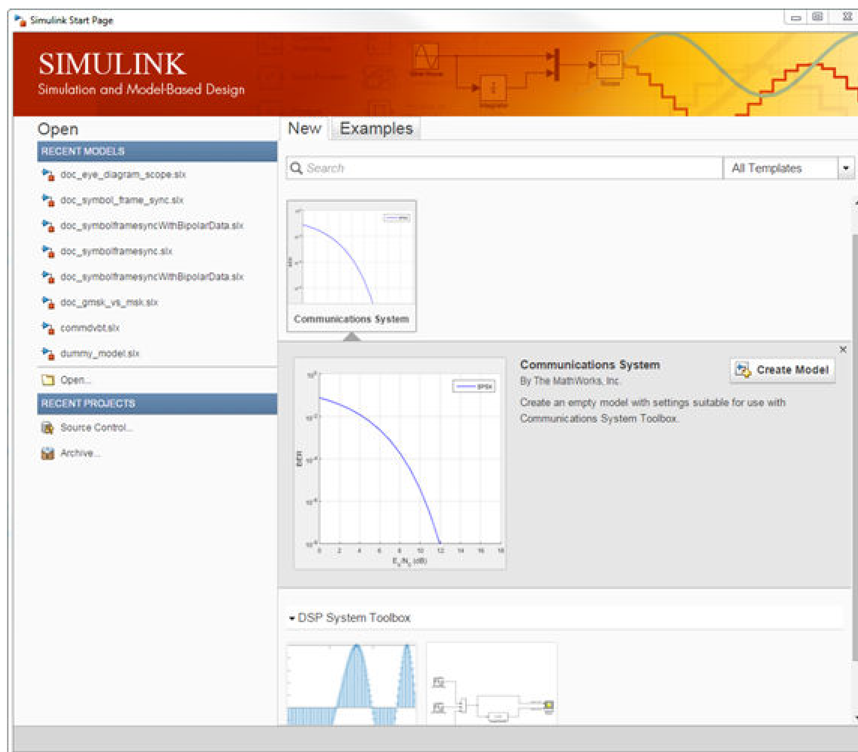
The Communications Toolbox Simulink model template lets you automatically configure the Simulink environment with the recommended settings for communications modeling. Communications Toolbox Simulink model templates enable reuse of settings, including configuration parameters. The model you create from the template uses best practices and takes advantage of previous solutions to common problems which helps you get started more quickly.

For more information on Simulink model templates, see “Build and Edit a Model Interactively” (Simulink).

Create Model Using the Communications Toolbox Simulink Model Template

To create a new blank model and open the library browser:

1. On the MATLAB® **Home** tab, click **Simulink**, and choose the **Communications** model template.
2. Click **Create Model** to create an empty model with settings suitable for use with Communications Toolbox. The new model opens. To access the library browser, click the **Library Browser** button on the model toolbar.



The new model using the template settings and contents appears in the Simulink Editor. The model is only in memory until you save it.

The Communications Toolbox Simulink Model Template

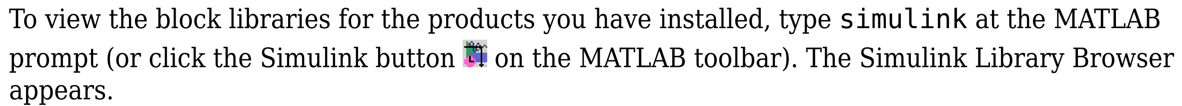
When you create a model by choosing the Communications Toolbox Simulink model template, the model is configured to use the settings recommended for communications modeling. Some of these settings are:

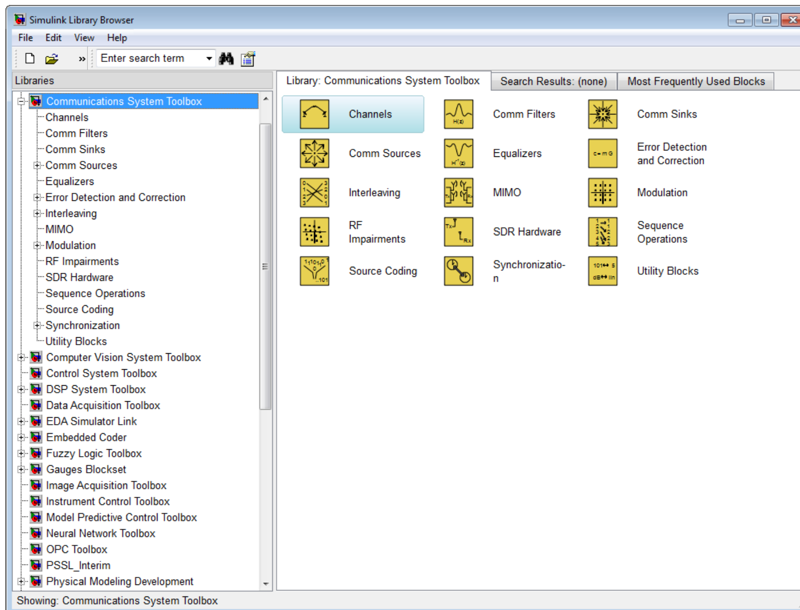
Configuration Parameter	Setting
'SingleTaskRateTransMsg'	'error'
'Solver'	'VariableStepDiscrete'
'EnableMultiTasking'	'Off'
'MaxStep'	'auto'
'StartTime'	'0.0'
'StopTime'	'inf'
'FixedStep'	'auto'
'SaveTime'	'off'
'SaveOutput'	'off'
'AlgebraicLoopMsg'	'error'
'RTWInlineParameters'	'on'
'BooleanDataType'	'off'
'UnnecessaryDatatypeConvMsg'	'none'
'LocalBlockOutputs'	'off'

Block Characteristics

You can type `showcommblockdatatypetable` at the MATLAB command line to generate a table showing characteristics of Simulink blocks in Communications Toolbox.

Access the Block Libraries

To view the block libraries for the products you have installed, type `simulink` at the MATLAB prompt (or click the Simulink button  on the MATLAB toolbar). The Simulink Library Browser appears.



Simulink Library Browser

The left pane displays the installed products, each of which has its own library of blocks. To open a library, click the **+** sign next to the product name in the left pane. This displays the contents of the library in the right pane.

You can find the blocks you need to build communications system models in the Communications Toolbox, DSP System Toolbox™, and Simulink libraries.

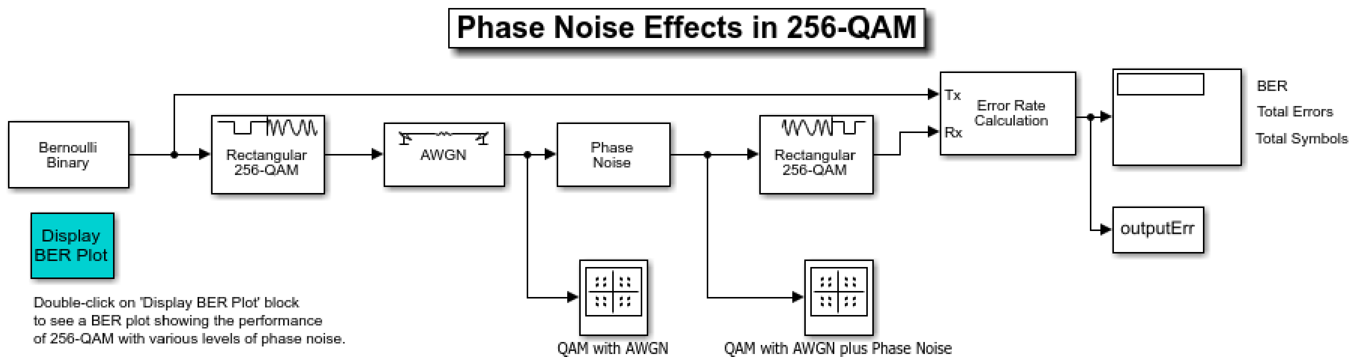
Alternatively, you can access the main Communications Toolbox block library by entering `commlib` at the MATLAB command line.

System Simulation

- “Examine 256-QAM Using Simulink” on page 2-2
- “Compute BER for QAM System with AWGN Using MATLAB” on page 2-7
- “Examine 16-QAM Using MATLAB” on page 2-8
- “Use Pulse Shaping on 16-QAM Signal” on page 2-14
- “Use Forward Error Correction on 16-QAM Signal” on page 2-21
- “QPSK and OFDM with MATLAB System Objects” on page 2-24
- “Accelerating BER Simulations Using the Parallel Computing Toolbox” on page 2-27
- “Iterative Design Workflow for Communication Systems” on page 2-30
- “What Is Different About Using Communications Toolbox Online?” on page 2-51

Examine 256-QAM Using Simulink

This example shows you the basic functionality of a communications system that models quadrature amplitude modulation (QAM), additive white Gaussian noise (AWGN) channel, and phase noise using Simulink®. The model displays constellation diagrams of the 256-QAM signal and performs error rate computations.



Copyright 2006-2019 The MathWorks, Inc.

Overview

The `cm_commphasenoise` model, simulates the effect of AWGN and phase noise on a 256-QAM signal. The Simulink model is a graphical representation for a mathematical model of a communications system that generates a random signal, modulates it using QAM, adds AWGN and phase noise to the signal, and demodulates the signal. The model also contains blocks to display the bit error rate and constellation diagrams of the modulated signal.

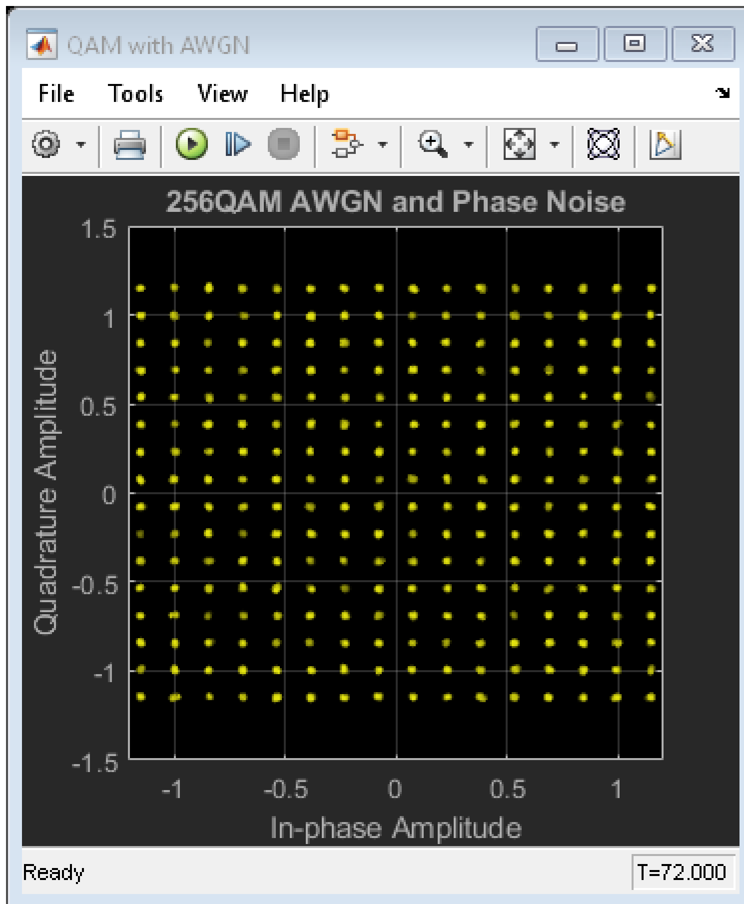
- The Bernoulli Binary Generator block generates a signal consisting of a sequence of 8-bit binary values in the range [0, 255].
- The Rectangular QAM Modulator Baseband block modulates the signal using baseband 256-ary QAM.
- The AWGN Channel block models a noisy channel by adding white Gaussian noise to the modulated signal.
- The Phase Noise block introduces noise in the angle of its complex input signal.
- The Rectangular QAM Demodulator Baseband block demodulates the signal.

Additional blocks in the model can help you interpret the simulation.

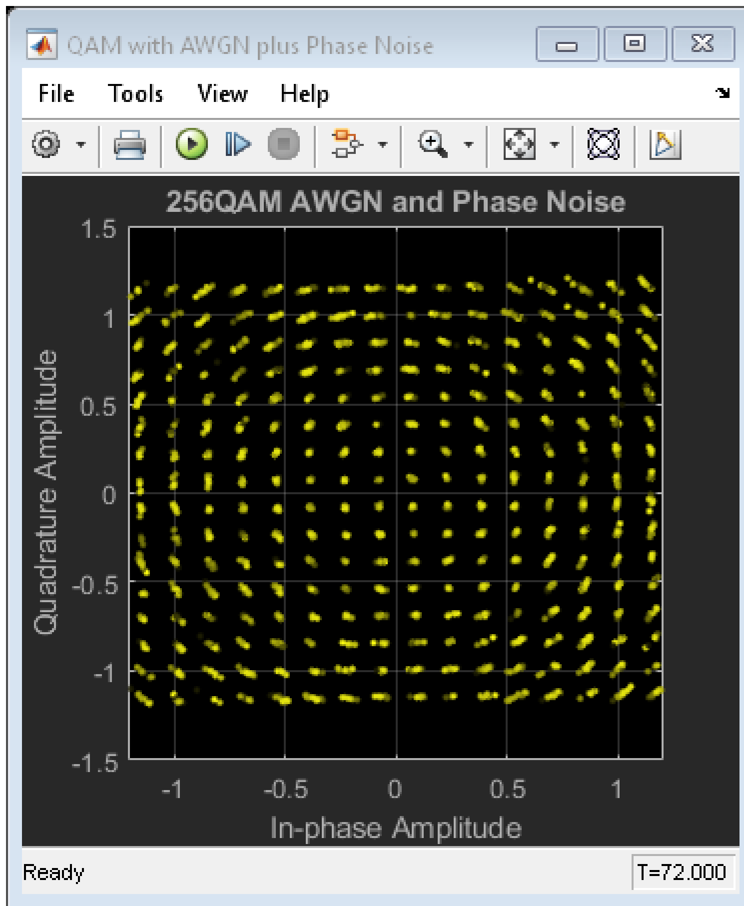
- The Constellation Diagram blocks, labeled `QAM with AWGN` and `QAM with AWGN plus Phase Noise`, display constellation diagrams of the signal with AWGN and phase noise added.
- The Error Rate Calculation block counts bits that differ between the received signal and transmitted signal.
- The To Workspace block, labeled `outputErr`, outputs the results to the workspace for use when plotting the results. The Display BER Plot block opens a bit error rate (BER) plot showing the Eb/N0 performance curves for 256-QAM trasnmission and reception at various levels of phase noise.

Digital Modulation

The model simulates QAM, which is a method for converting a digital signal to a complex signal. The model modulates the signal onto a sequence of complex numbers that lie on a lattice of points in the complex plane, known as the constellation of the signal. A plot of these points is called a *scatterplot* or *constellation diagram* of the signal. The constellation for a baseband 256-ary QAM with AWGN added is shown in this figure. The points in the constellation diagram do not lie exactly on the constellation shown in the figure because of the added noise.



This plot shows the constellation with AWGN and phase noise added. Phase noise alters the angle of the complex modulated signal, causing a radial displacement of constellation points.



Run the Simulation

The default model configuration has the run duration set to `inf`. The Error Rate Calculation block is configured to run until 100 errors occur. To stop the simulation before 100 errors occur, click **Stop** on the **Simulation** tab.

Display the Error Rate

The Display block displays the number of errors introduced by the AWGN channel and phase noise. When you run the simulation, three small boxes appear in the block, displaying the vector output from the Error Rate Calculation block.

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

Display a Phase Noise Plot

To display a figure that plots simulation results of BER verus Eb/N0 curves for a range of phase noise settings, double-click the Display BER Plot block in the model.

Further Exploration

You can control the way a Simulink block functions by setting its parameters. To view or change simulation parameters, double-click a block to open its block mask.

To change the amount of phase noise, open the Phase Noise block mask and enter a new value for the **Phase noise level (dBc/Hz)** parameter. Click **OK** to apply the new setting.

To change the amount of noise, open the AWGN Channel block mask and enter a new value for the **Eb/No (dB)** parameter. Decreasing this parameter value increases the noise level. Click **OK** to apply the new setting.

Reducing the phase noise and increasing the Eb/N0 removes noise from the model. Since the model is configured to run until 100 errors occur, running the simulation with little noise in the model results in a long simulation run time. To limit the maximum simulation run time, you can reduce the run duration from `inf` to a small value, such as `10`.

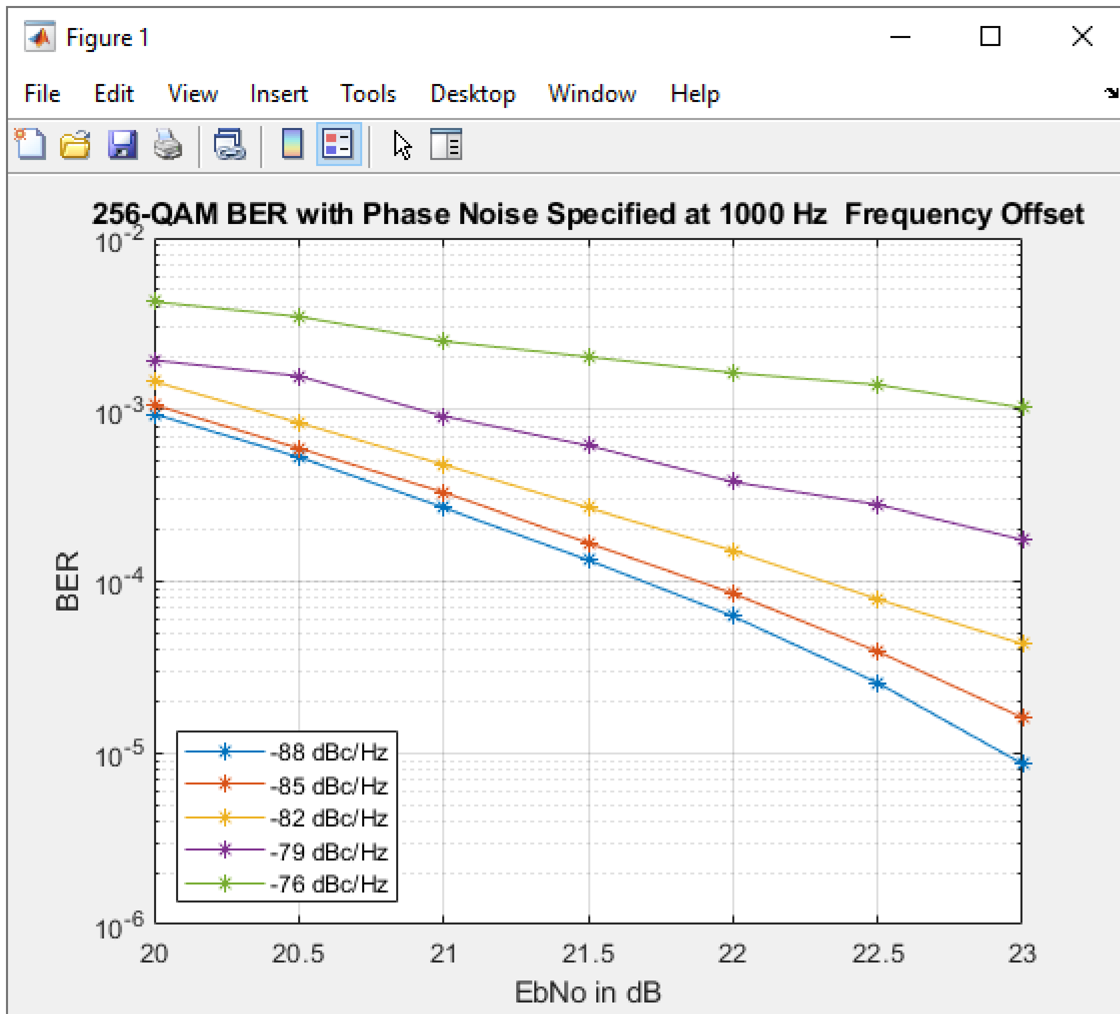
To produce new results, run the simulation using the modified settings.

Alternatively, you can enter a variable name in a parameter. Then at the MATLAB® command line set the value for that variable in the workspace. Setting parameters in the Command Window can be convenient if you need to run multiple simulations with different parameter values.

You can also use callback functions to configure your simulation. The default setting for several parameters in this model are set using the `PreLoadFcn` callback function. To access the callback functions, select **Model Settings** > **Model Properties** on the **Modeling** tab. In the **Model Properties** dialog, select the **Callbacks** tab. For more information on model properties and callback functions, see “Model Callbacks” (Simulink).

Plot BER at Different Noise Levels

The `plot_256qam_ber_curves.m` MATLAB® program file generated this BER plot by running multiple simulations with different values for the **Phase noise level (dBc/Hz)** and **Eb/No (dB)** parameters. Each curve is a plot of BER as a function of signal to noise ratio for a fixed amount of phase noise. For each plotted BER point, the simulation stopped when 1000 bit errors were reached or 1e8 bits were compared. Results vary from run to run due to the random nature of the input signal and simulation impairments.



See Also

Related Examples

- "Passband Modulation"
- "Configure the Simulink Environment for Communications Models" on page 1-3

Compute BER for QAM System with AWGN Using MATLAB

Communications Toolbox features build upon the MATLAB computational and visualization tools, enabling you to use higher level functions when simulating communications systems. This set of examples shows how to compute the bit error rate (BER) on a 16-QAM signal distorted by an AWGN channel.

- “Examine 16-QAM Using MATLAB” on page 2-8 — Shows a basic 16-QAM communications link
- “Use Pulse Shaping on 16-QAM Signal” on page 2-14 — Extends the basic 16-QAM communications link example to include pulse shape filtering
- “Use Forward Error Correction on 16-QAM Signal” on page 2-21 — Extends the 16-QAM communications link with pulse shaping example to include forward error correction (FEC)

Examine 16-QAM Using MATLAB

This example shows how to process a binary data stream by using a communications link that consists of a baseband modulator, channel, and demodulator. The example displays a portion of the random data in a stem plot, displays the transmitted and received signals in constellation diagrams, and computes the bit error rate (BER). To add a pulse shaping filter to the communications link, see the “Use Pulse Shaping on 16-QAM Signal” on page 2-14 example. To add forward error correction to the communications link with pulse shape filtering, see the “Use Forward Error Correction on 16-QAM Signal” on page 2-21 example.

Modulate Random Signal

The modulation scheme uses baseband 16-QAM, and the signal passes through an additive white Gaussian noise (AWGN) channel. The basic simulation operations use these Communications Toolbox™ and MATLAB® functions.

- `rng` — Controls the random number generation.
- `randi` — Generates a random binary data stream.
- `bi2de` — Converts the binary signal to an integer-valued signal.
- `qammod` — Modulates using 16-QAM.
- `comm.AWGNChannel` — Impairs the transmitted data using AWGN.
- `scatterplot` — Creates constellation diagrams.
- `qamdemod` — Demodulates using 16-QAM.
- `de2bi` — Converts the integer-valued signal to a binary signal.
- `biterr` — Computes the system BER.

Generate Random Binary Data Stream

The conventional format for representing a signal in MATLAB is a vector or matrix. The `randi` function creates a column vector containing the values of a binary data stream. The length of the binary data stream (that is, the number of rows in the column vector) is arbitrarily set to 30,000.

Define parameters.

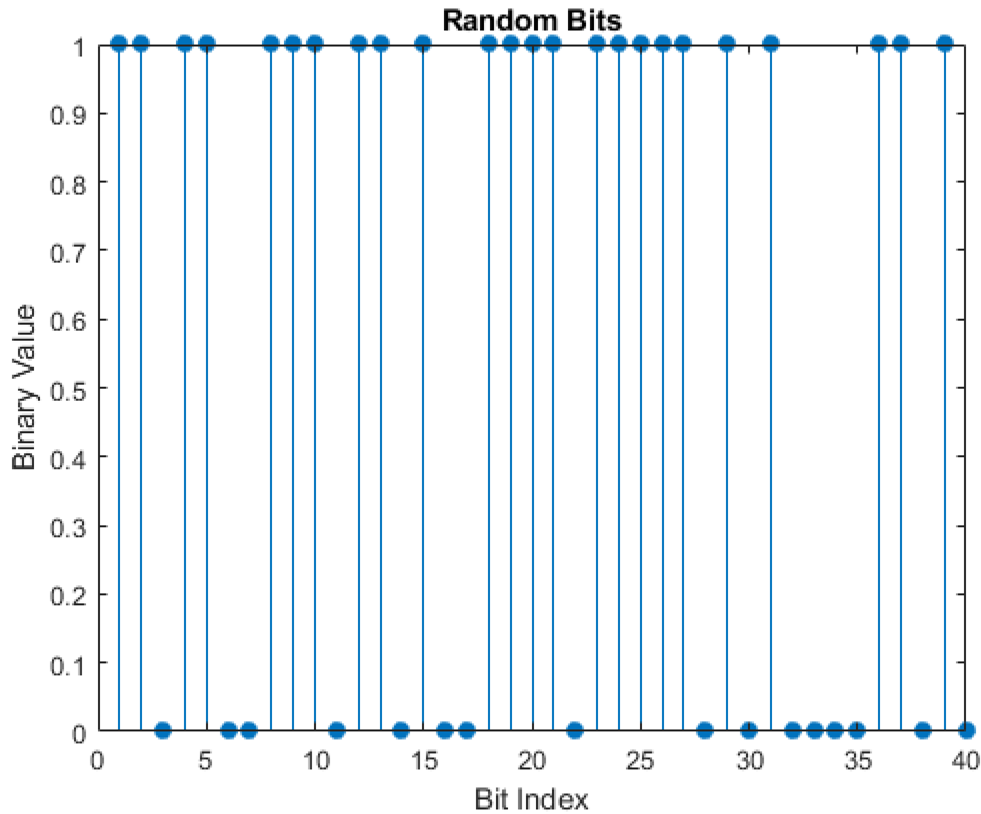
```
M = 16; % Modulation order (alphabet size or number of points in signal constellation)
k = log2(M); % Number of bits per symbol
n = 30000; % Number of bits to process
sps = 1; % Number of samples per symbol (oversampling factor)
```

Set the `rng` function to its default state, or any static seed value, so that the example produces repeatable results. Then use the `randi` function to generate random binary data.

```
rng default;
dataIn = randi([0 1],n,1); % Generate vector of binary data
```

Use a stem plot to show the binary values for the first 40 bits of the random binary data stream. Use the colon (`:`) operator in the call to the `stem` function to select a portion of the binary vector.

```
stem(dataIn(1:40),'filled');
title('Random Bits');
xlabel('Bit Index');
ylabel('Binary Value');
```



Convert Binary Signal to Integer-Valued Signal

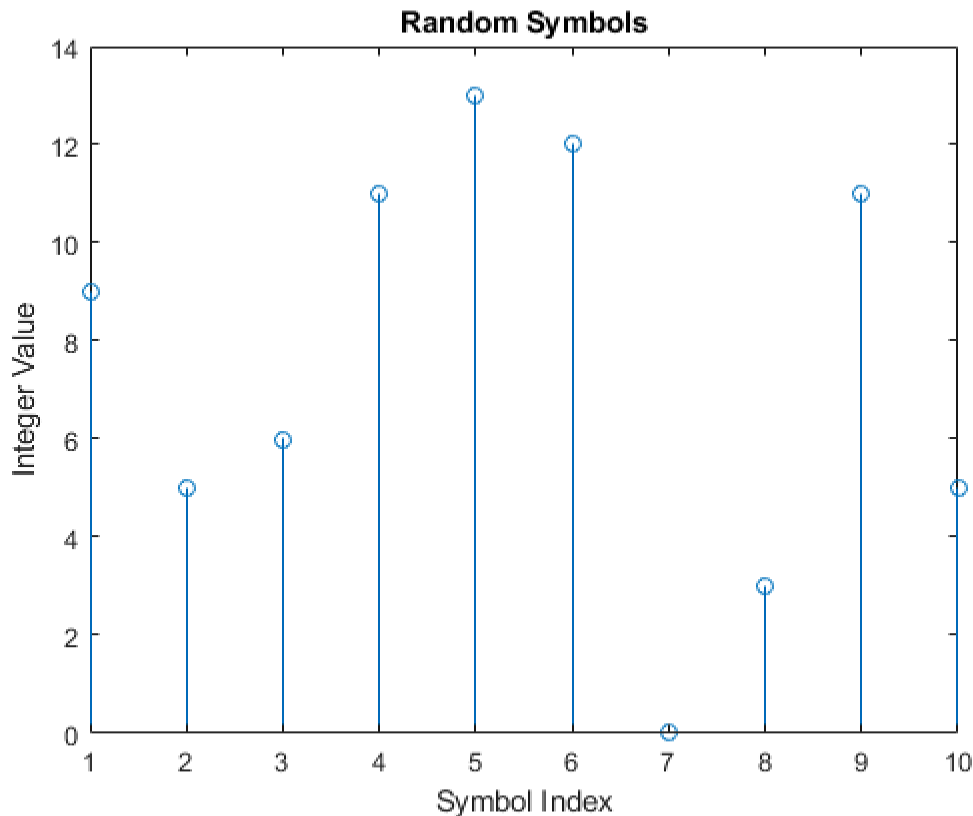
The default configuration for the `qammod` function expects integer-valued data as the input symbols to modulate. In this example, the binary data stream is preprocessed into integer values before using the `qammod` function. In particular, the `bi2de` function converts each 4-tuple to a corresponding integer in the range [0, (`M`–1)]. The modulation order, `M`, is 16 in this example.

Perform a bit-to-symbol mapping by first reshaping the data into binary `k`-tuples, where `k` is the number of bits per symbol defined by $k = log_2(M)$. Then, use the `bi2de` function to convert each 4-tuple to an integer value.

```
dataInMatrix = reshape(dataIn,length(dataIn)/k,k);
dataSymbolsIn = bi2de(dataInMatrix);
```

Plot the first 10 symbols in a stem plot.

```
figure; % Create new figure window.
stem(dataSymbolsIn(1:10));
title('Random Symbols');
xlabel('Symbol Index');
ylabel('Integer Value');
```



Modulate Using 16-QAM

Use the `qammod` function to apply 16-QAM modulation to the `dataSymbolsIn` column vector for natural-encoded and Gray-encoded binary bit-to-symbol mappings.

```
dataMod = qammod(dataSymbolsIn,M,'bin'); % Binary coding with phase offset of zero
dataModG = qammod(dataSymbolsIn,M); % Gray coding with phase offset of zero
```

The modulation operation outputs complex column vectors containing values that are elements of the 16-QAM signal constellation. Later in this example constellation diagrams show the natural and Gray binary symbol mapping.

For more information on modulation functions, see “Digital Modulation”. For an example that uses Gray coding with phase-shift keying (PSK) modulation, see “Gray Coded 8-PSK”.

Add White Gaussian Noise

The modulated signal passes through the channel by using the `awgn` function with the specified signal-to-noise ratio (SNR). Convert the ratio of energy per bit to noise power spectral density (E_b/N_0) to an SNR value for use by the `awgn` function. The `sps` variable is not significant in this example but makes extending the example to use pulse shaping easier. For more information, see the “Use Pulse Shaping on 16-QAM Signal” on page 2-14 example.

Calculate the SNR when the channel has an E_b/N_0 of 10 dB.

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

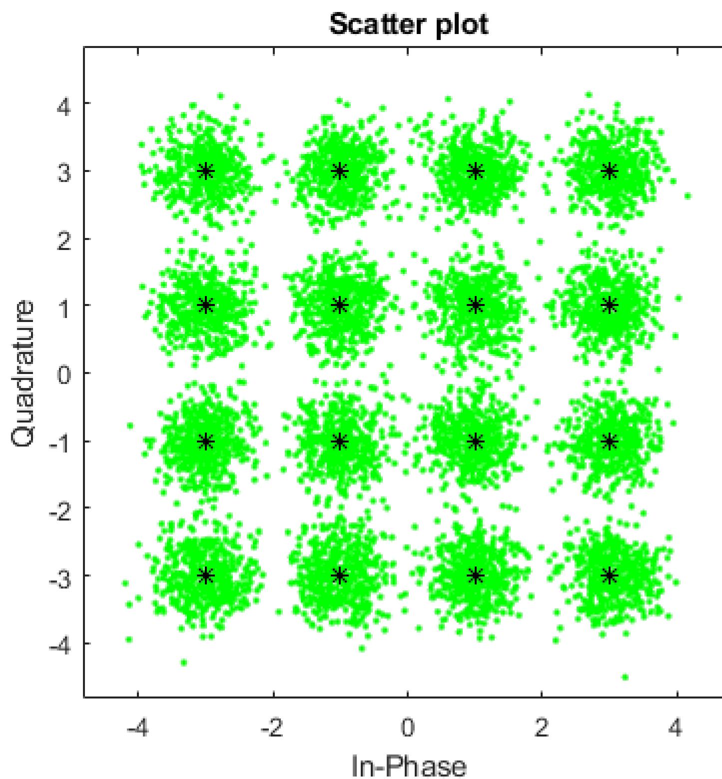
Pass the signal through the AWGN channel for the binary and Gray coded symbol mappings.

```
receivedSignal = awgn(dataMod,snr,'measured');
receivedSignalG = awgn(dataModG,snr,'measured');
```

Create Constellation Diagram

Use the `scatterplot` function to display the in-phase and quadrature components of the modulated signal, `dataMod`, and the noisy signal received after the channel. The effects of AWGN are present in the constellation diagram.

```
sPlotFig = scatterplot(receivedSignal,1,0,'g.');
hold on
scatterplot(dataMod,1,0,'k*',sPlotFig)
```



Demodulate 16-QAM

Use the `qamdemod` function to demodulate the received data and output integer-valued data symbols.

```
dataSymbolsOut = qamdemod(receivedSignal,M,'bin');
dataSymbolsOutG = qamdemod(receivedSignalG,M);
```

Convert Integer-Valued Signal to Binary Signal

Use the `de2bi` function to convert the data symbols from the QAM demodulator, `dataSymbolsOut`, into a binary matrix, `dataOutMatrix`, with dimensions N_{sym}-by-$N_{bits/sym}$. In the matrix, N_{sym} is the total number of QAM symbols, and $N_{bits/sym}$ is the number of bits per symbol. For 16-QAM, $N_{bits/sym}$

= 4. Then convert the matrix into a column vector of length equal to the number of input bits, 30,000. Repeat the process for the Gray-encoded data symbols, `dataSymbolsOutG`.

Reverse the bit-to-symbol mapping performed earlier in this example.

```
dataOutMatrix = de2bi(dataSymbolsOut,k);
dataOut = dataOutMatrix(:); % Return data in column vector
dataOutMatrixG = de2bi(dataSymbolsOutG,k);
dataOutG = dataOutMatrixG(:); % Return data in column vector
```

Compute System BER

The `biterr` function calculates the bit error statistics from the original binary data stream, `dataIn`, and the received data streams, `dataOut` and `dataOutG`. Gray coding significantly reduces the BER.

Use the error rate function to compute the error statistics. Use the `fprintf` function to display the results.

```
[numErrors,ber] = biterr(dataIn,dataOut);
fprintf('\nThe binary coding bit error rate is %5.2e, based on %d errors.\n', ...
    ber,numErrors)
```

```
The binary coding bit error rate is 2.40e-03, based on 72 errors.
```

```
[numErrorsG,berG] = biterr(dataIn,dataOutG);
fprintf('\nThe Gray coding bit error rate is %5.2e, based on %d errors.\n', ...
    berG,numErrorsG)
```

```
The Gray coding bit error rate is 1.33e-03, based on 40 errors.
```

Plot Signal Constellations

The constellation diagram shown previously plotted the points in the QAM constellation, but it did not indicate the mapping between symbol values and the constellation points. In this section, the constellation diagram indicates the mappings for natural-encoding and Gray-encoding of binary data to constellation points.

Show Natural and Gray Coded Binary Symbol Mapping for 16-QAM Constellation

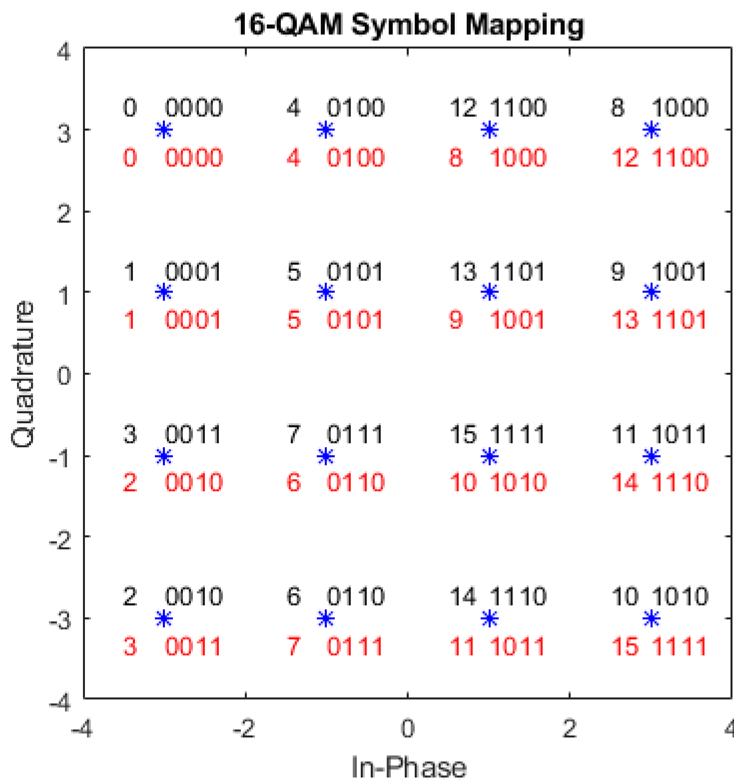
Apply 16-QAM modulation to complete sets of constellation points by using natural-coded binary symbol mapping and Gray-coded symbol mapping.

```
M = 16; % Modulation order
x = (0:15); % Integer input
symbin = qammod(x,M,'bin'); % 16-QAM output (natural-coded binary)
symgray = qammod(x,M,'gray'); % 16-QAM output (Gray-coded)
```

Use the `scatterplot` function to plot the constellation diagram and annotate it with natural (red) and Gray (black) binary representations of the constellation points.

```
scatterplot(symgray,1,0,'b*');
for k = 1:M
    text(real(symgray(k)) - 0.0,imag(symgray(k)) + 0.3,...
        dec2base(x(k),2,4));
     text(real(symgray(k)) - 0.5,imag(symgray(k)) + 0.3,...
         num2str(x(k)));

    text(real(symbin(k)) - 0.0,imag(symbin(k)) - 0.3,...
```

```
        dec2base(x(k),2,4),'Color',[1 0 0]);
    text(real(symbin(k)) - 0.5,imag(symbin(k)) - 0.3,...
        num2str(x(k)),'Color',[1 0 0]);
end
title('16-QAM Symbol Mapping')
axis([-4 4 -4 4])
```



Examine Plots

Using Gray-coded symbol mapping improves BER performance because the Gray-coded signal constellation points differ by only one bit from each adjacent neighboring point. Where as with natural binary symbol mapping, some of the adjacent constellation points. For example, the natural-coded binary values for 1 (`0 0 0 1`) and 2 (`0 0 1 0`) differ by two bits (the third and forth bits).

See Also

Related Examples

- “Use Pulse Shaping on 16-QAM Signal” on page 2-14
- “Use Forward Error Correction on 16-QAM Signal” on page 2-21

Use Pulse Shaping on 16-QAM Signal

This example extends the "Examine 16-QAM Using MATLAB" on page 2-8 example to perform pulse shaping and matched filtering by using a pair of square-root raised cosine (RRC) filters. The `rcosdesign` function creates the filters. BER performance can be improved by adding forward error correction (FEC) to the communication link. To add FEC to the communications link with pulse shape filtering example, see the "Use Forward Error Correction on 16-QAM Signal" on page 2-21 example.

This example shows how to process a binary data stream by using a communications link that consists of a baseband modulator, channel, demodulator, and pulse shaping and matching filters. The example displays a portion of the random data in a stem plot, displays the transmitted and received signals in constellation diagrams, and computes the bit error rate (BER).

Establish Simulation Framework

Define simulation parameters for a 16-QAM modulation scheme with raised cosine filtering, and an AWGN channel.

```
M = 16; % Modulation order
k = log2(M); % Number of bits per symbol
numBits = 3e5; % Number of bits to process
sps = 4; % Number of samples per symbol (oversampling factor)
```

Create RRC Filter

Set the RRC filter parameters.

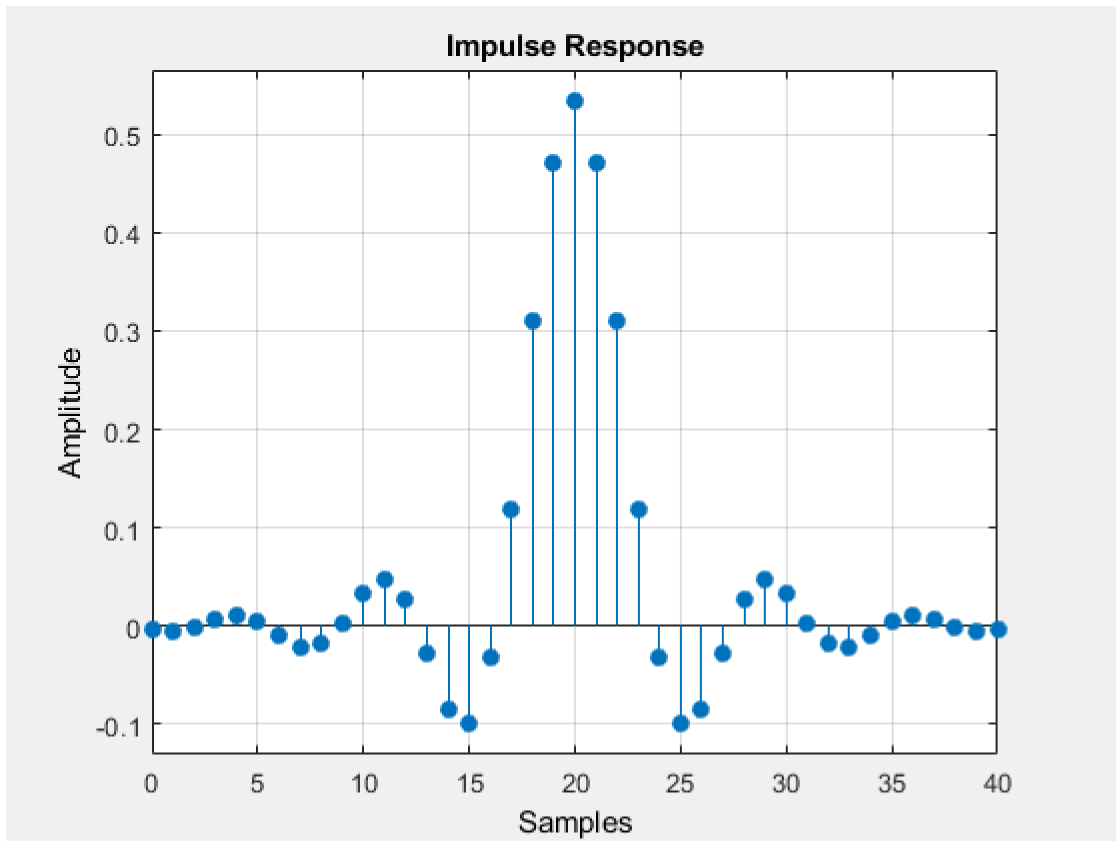
```
filtlen = 10; % Filter length in symbols
rolloff = 0.25; % Filter rolloff factor
```

Use the `rcosdesign` function to create an RRC filter.

```
rrcFilter = rcosdesign(rolloff,filtlen,sps);
```

Use the FVTool to display the RRC filter impulse response.

```
fvtool(rrcFilter,'Analysis','Impulse')
```



Compute System BER

Use the `randi` function to generate random binary data. Set the `rng` function to its default state, or any static seed value, so that the example produces repeatable results.

```
rng default; % Use default random number generator
dataIn = randi([0 1],numBits,1); % Generate vector of binary data
```

Reshape the input vector into a matrix of 4-bit binary data. Then, use the `bi2de` function to convert the data into integer symbols.

```
dataInMatrix = reshape(dataIn,length(dataIn)/k,k); % Reshape data into binary 4-tuples
dataSymbolsIn = bi2de(dataInMatrix); % Convert to integers
```

Apply 16-QAM modulation using the `qammod` function.

```
dataMod = qammod(dataSymbolsIn,M);
```

Use the `upfirdn` function to upsample the signal by the oversampling factor and apply the RRC filter. The `upfirdn` function pads the upsampled signal with zeros at the end to flush the filter. Then, the function applies the filter.

```
txFiltSignal = upfirdn(dataMod,rrcFilter,sps,1);
```

Using the number of bits per symbol (`k`) and the number of samples per symbol (`sps`), convert the ratio of energy per bit to noise power spectral density (`EbNo`) to an SNR value for use by the `awgn` function.

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

Pass the filtered signal through an AWGN channel.

```
rxSignal = awgn(txFiltSignal,snr,'measured');
```

Use the `upfirdn` function on the received signal to downsample and filter the signal. Downsample by using the same oversampling factor applied for upsampling the transmitted signal. Filter by using the same RRC filter applied to the transmitted signal.

Each filtering operation delays the signal by half of the filter length in symbols, `filtlen`/2. So, the total delay from transmit and receive filtering equals the filter length, `filtlen`. For the BER computation, the transmitted and received signals must be the same size and you must account for the delay between the transmitted and received signal. Remove the first `filtlen` symbols in the decimated signal to account for the cumulative delay of the transmit and receive filtering operations. Remove the last `filtlen` symbols in the decimated signal to ensure the number of samples in the demodulator output matches the number of samples in the modulator input.

```
rxFiltSignal = upfirdn(rxSignal,rrcFilter,1,sps); % Downsample and filter
rxFiltSignal = rxFiltSignal(filtlen + 1:end - filtlen); % Account for delay
```

Use the `qamdemod` function to demodulate the received filtered signal.

```
dataSymbolsOut = qamdemod(rxFiltSignal,M);
```

Convert the recovered integer symbols into binary data by using the `de2bi` function.

```
dataOutMatrix = de2bi(dataSymbolsOut,k);
dataOut = dataOutMatrix(:); % Return data in column vector
```

Determine the number of errors and the associated BER by using the `biterr` function.

```
[numErrors,ber] = biterr(dataIn,dataOut);
fprintf('\nFor an EbNo setting of %3.1f dB, the bit error rate is %5.2e, based on %d errors.\n',
    EbNo,ber,numErrors)
```

```
For an EbNo setting of 10.0 dB, the bit error rate is 1.83e-03, based on 550 errors.
```

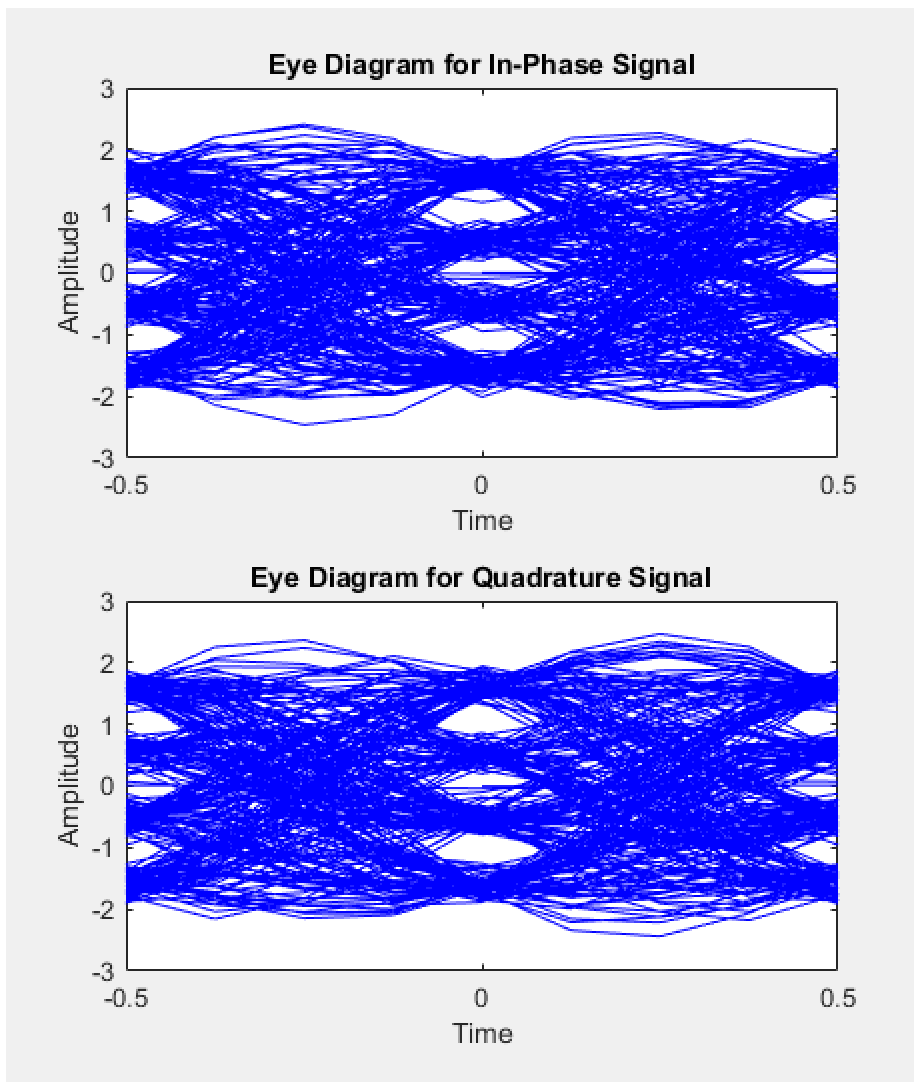
Visualize Filter Effects

To visualize the filter effects in an eye diagram, reduce the E_b/N_0 setting and regenerate the received data. Visualizing a high SNR signal with no other multipath effects, you can use eye diagrams to highlight the intersymbol interference (ISI) reduction at the output for the matched pair of pulse shaping RRC filters. The RRC filter does not have zero-ISI until it is paired with the second RRC filter to form in cascade a raised cosine filter as a matched pair of filters.

```
EbNo = 20;
snr = EbNo + 10*log10(k) - 10*log10(sps);
rxSignal = awgn(txFiltSignal,snr,'measured');
rxFiltSignal = upfirdn(rxSignal,rrcFilter,1,sps); % Downsample and filter
rxFiltSignal = rxFiltSignal(filtlen + 1:end - filtlen); % Account for delay
```

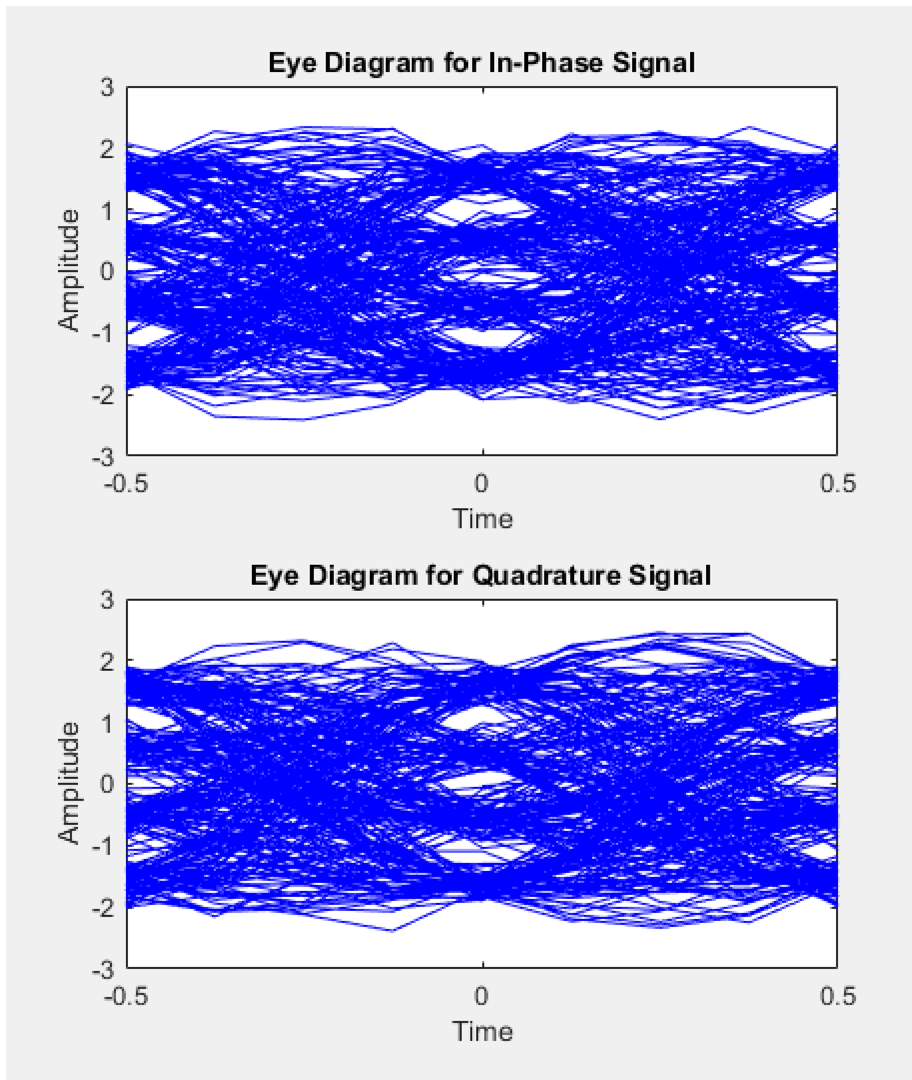
Create an eye diagram for a portion of the filtered noiseless signal to visualize the effect of the pulse shaping. The transmitted signal has RRC filtering and shows ISI as a narrowing of the *eye-opening*.

```
eyediagram(txFiltSignal(1:2000),sps*2);
```



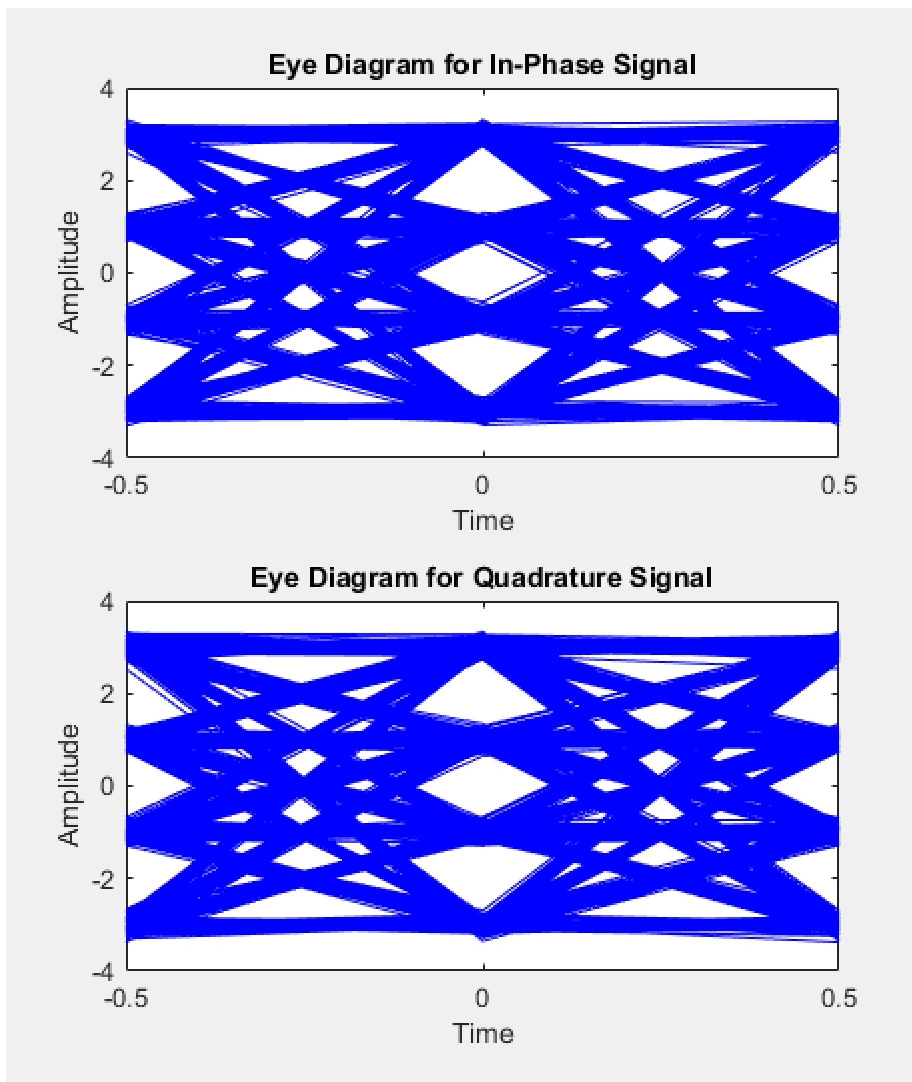
Displaying the eye diagram of the signal after the channel noise shows the signal with RRC filtering and noise. The noise level causes further narrowing of the eye diagram eye-opening.

```
eyediagram(rxSignal(1:2000),sps*2);
```



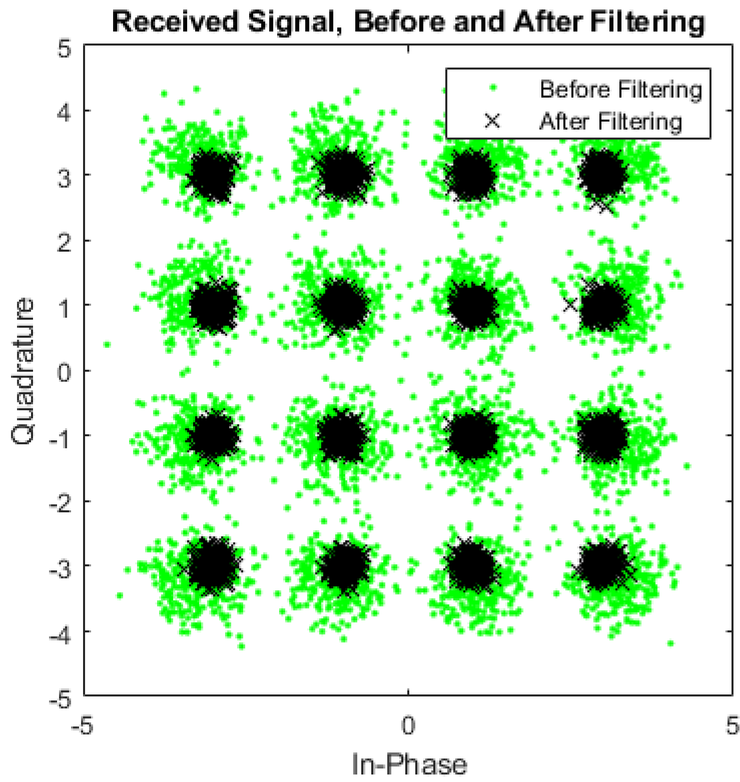
Displaying the eye diagram of the signal after the matched receive filtering is applied shows the signal with raised cosine filtering. The wider eye diagram eye-openings, the signal has less ISI with raised cosine filtering as compared to the signal with RRC filtering.

```
eyediagram(rxFiltSignal(1:2000),2);
```



Create a constellation diagram of the received signal before and after filtering. Scale the received signal by the square root of the number of samples per symbol to normalize the transmit and receive power levels.

```
scatplot = scatterplot(sqrt(sps)*...
    rxSignal(1:sps*5e3),...
    sps,0,'g.');
hold on;
scatterplot(rxFiltSignal(1:5e3),1,0,'kx',scatplot);
title('Received Signal, Before and After Filtering');
legend('Before Filtering','After Filtering');
axis([-5 5 -5 5]); % Set axis ranges
hold off;
```



See Also

Related Examples

- "Examine 16-QAM Using MATLAB" on page 2-8
- "Use Forward Error Correction on 16-QAM Signal" on page 2-21

Use Forward Error Correction on 16-QAM Signal

This example extends the "Use Pulse Shaping on 16-QAM Signal" on page 2-14 example to show bit error rate (BER) performance improvement when using forward error correction (FEC) coding.

This example shows how to process a binary data stream by using a communications link that consists of a baseband modulator, channel, demodulator, pulse shaping and matching filters, and error correction.

Establish Simulation Framework

In this example, to achieve a more accurate BER estimate, the number of bits to process is increased from the value used in the "Use Pulse Shaping on 16-QAM Signal" on page 2-14 example. Other simulation variables match the settings in that example.

Define simulation parameters for a 16-QAM modulation scheme with raised cosine filtering and an AWGN channel.

```
M = 16; % Modulation order
k = log2(M); % Number of bits per symbol
numBits = 1e6; % Number of bits to process
sps = 4; % Number of samples per symbol (oversampling factor)
filtlen = 10; % Filter length in symbols
rolloff = 0.25; % Filter rolloff factor
```

Generate Random Data

Set the `rng` function to its default state, or any static seed value, so that the example produces repeatable results. Then, use the `randi` function to generate random binary data.

```
rng default;                     % Use default random number generator
dataIn = randi([0 1],numBits,1); % Generate vector of binary data
```

Apply Convolutional Encoding

To correct errors arising from the noisy channel, apply convolutional coding to the data before transmission and Viterbi decoding to the received data. The decoder uses a hard decision algorithm, which means each received data bit is interpreted as either `0` or `1`.

Define a convolutional coding trellis for a rate 2/3 code by using the `poly2trellis` function. The defined trellis represents the convolutional code that the `convenc` function uses for encoding the binary vector, `dataIn`.

```
constrlen = [5 4]; % Code constraint length
genpoly = [23 35 0; 0 5 13] % Generator polynomials
```

```
genpoly = 2×3

    23    35     0
     0     5    13
```

```
tPoly = poly2trellis(constrlen,genpoly);
codeRate = 2/3;
```

Encode the input data by using the `tPoly` trellis.

```
dataEnc = convenc(dataIn,tPoly);
```

Modulate Data

Reshape the input vector into a matrix of 4-bit binary data. Then, use the `bi2de` function to convert the encoded binary data to an integer format.

```
dataEncMatrix = reshape(dataEnc, ...
    length(dataEnc)/k,k); % Reshape data into binary 4-tuples
dataSymbolsIn = bi2de(dataEncMatrix); % Convert to integers
```

Use the `qammod` function to apply 16-QAM modulation.

```
dataMod = qammod(dataSymbolsIn,M);
```

Apply Raised Cosine Filtering

Use the `rcosdesign` function to create an RRC filter.

```
rrcFilter = rcosdesign(rolloff,filtlen,sps);
```

Use the `upfirdn` function to upsample the signal by the oversampling factor and apply the RRC filter. The `upfirdn` function pads the upsampled signal with zeros at the end to flush the filter. Then, the function applies the filter.

```
txSignal = upfirdn(dataMod,rrcFilter,sps,1);
```

Apply AWGN Channel

Using the number of bits per symbol (`k`) and the number of samples per symbol (`sps`), convert the ratio of energy per bit to noise power spectral density (`EbNo`) to an SNR value for use by the `awgn` function. When converting the E_b/N_0 to SNR, you must account for the number of information bits per symbol. With no FEC applied, each symbol corresponded to `k` bits. With FEC applied, each symbol corresponds to (`k` × `codeRate`) information bits. For the 2/3 code rate and 16-QAM transmissions used in this example, three symbols correspond to 12 coded bits and 8 uncoded (information) bits.

```
EbNo = 10;
snr = EbNo+10*log10(k*codeRate)-10*log10(sps);
```

Pass the filtered signal through an AWGN channel.

```
rxSignal = awgn(txSignal,snr,'measured');
```

Receive and Demodulate Signal

Filter the received signal by using the RRC filter. Remove a portion of the signal to account for the filter delay.

```
rxFiltSignal = upfirdn(rxSignal,rrcFilter,1,sps); % Downsample and filter
rxFiltSignal = rxFiltSignal(filtlen + 1:end - filtlen); % Account for delay
```

Use the `qamdemod` function to demodulate the received filtered signal.

```
dataSymbolsOut = qamdemod(rxFiltSignal,M);
```

Apply Viterbi Decoding

Use the `de2bi` function to convert the recovered integer symbols into binary data.

```
dataOutMatrix = de2bi(dataSymbolsOut,k);
codedDataOut = dataOutMatrix(:); % Return data in column vector
```

Use the `vitdec` function, configured for hard decisions and continuous operation mode, to decode the convolutionally encoded data. The continuous operation mode maintains the internal state when the decoder is repeatedly invoked, such as when receiving frames of data operating in a loop. The continuous operation mode also adds delay to the system. Although this example does not use a loop, the '`cont`' mode is used for the purpose of illustrating how to compensate for the delay in this decoding operation.

```
traceBack = 16; % Traceback length for decoding
numCodeWords = floor(length(codedDataOut)*2/3); % Number of complete codewords
dataOut = vitdec(codedDataOut(1:numCodeWords*3/2), ...
          tPoly,traceBack,'cont','hard'); % Decode data
```

Compute System BER

Use the `biterr` function to compute the number of errors and the BER by comparing `dataIn` and `dataOut`. The delay introduced by the transmit and receive RRC filters is already accounted for in the recovered data, but the decoder delay is not accounted for yet. The continuous operation mode of the Viterbi decoder incurs a delay with a duration in bits equal to the traceback length, `traceBack`, times the number of input streams at the encoder. For the 2/3 code rate used in this example, the encoder has two input streams, so the delay is 2×`traceBack` bits. As a result, the first 2×`traceBack` bits in the decoded vector, `dataOut`, are zeros. When computing the BER, discard the first 2×`traceBack` bits in `dataOut` and the last 2×`traceBack` bits in the original vector, `dataIn`.

```
decDelay = 2*traceBack; % Decoder delay, in bits
[numErrors,ber] = ...
   biterr(dataIn(1:end - decDelay),dataOut(decDelay + 1:end));
fprintf('\nThe bit error rate is %5.2e, based on %d errors.\n', ...
    ber,numErrors)
```

```
The bit error rate is 1.00e-04, based on 100 errors.
```

For the same E_b/N_0 of 10 dB, less errors occur when using FEC and the BER improves from approximately 2×10-3 to 1×10-4.

More About Delays

The decoding operation in this example incurs a delay that causes the output of the decoder to lag the input. Timing information does not appear explicitly in the example, and the length of the delay depends on the specific operations being performed. Delays occur in various communications system operations, including convolutional decoding, convolutional interleaving and deinterleaving, equalization, and filtering. To find out the duration of the delay caused by specific functions or operations, see the specific documentation for those functions or operations. For more information on delays, see “Delays of Convolutional Interleavers” and “Fading Channels”.

See Also

Related Examples

- “Examine 16-QAM Using MATLAB” on page 2-8
- “Use Pulse Shaping on 16-QAM Signal” on page 2-14

QPSK and OFDM with MATLAB System Objects

This example shows how to simulate a basic communication system in which the signal is first QPSK modulated and then subjected to Orthogonal Frequency Division Multiplexing. The signal is then passed through an additive white Gaussian noise channel prior to being demultiplexed and demodulated. Lastly, the number of bit errors are calculated. The example showcases the use of MATLAB® System objects™.

Set the simulation parameters.

```
M = 4;                 % Modulation alphabet
k = log2(M);           % Bits/symbol
numSC = 128;           % Number of OFDM subcarriers
cpLen = 32;            % OFDM cyclic prefix length
maxBitErrors = 100;    % Maximum number of bit errors
maxNumBits = 1e7;      % Maximum number of bits transmitted
```

Construct System objects needed for the simulation: QPSK modulator, QPSK demodulator, OFDM modulator, OFDM demodulator, AWGN channel, and an error rate calculator. Use name-value pairs to set the object properties.

Set the QPSK modulator and demodulator so that they accept binary inputs.

```
qpskMod = comm.QPSKModulator('BitInput',true);
qpskDemod = comm.QPSKDemodulator('BitOutput',true);
```

Set the OFDM modulator and demodulator pair according to the simulation parameters.

```
ofdmMod = comm.OFDMModulator('FFTLength',numSC,'CyclicPrefixLength',cpLen);
ofdmDemod = comm.OFDMDemodulator('FFTLength',numSC,'CyclicPrefixLength',cpLen);
```

Set the `NoiseMethod` property of the AWGN channel object to `Variance` and define the `VarianceSource` property so that the noise power can be set from an input port.

```
channel = comm.AWGNChannel('NoiseMethod','Variance', ...
    'VarianceSource','Input port');
```

Set the `ResetInputPort` property to `true` to enable the error rate calculator to be reset during the simulation.

```
errorRate = comm.ErrorRate('ResetInputPort',true);
```

Use the `info` function of the `ofdmMod` object to determine the input and output dimensions of the OFDM modulator.

```
ofdmDims = info(ofdmMod)
```

```
ofdmDims = struct with fields:
    DataInputSize: [117 1]
       OutputSize: [160 1]
```

Determine the number of data subcarriers from the `ofdmDims` structure variable.

```
numDC = ofdmDims.DataInputSize(1)
```

```
numDC = 117
```

Determine the OFDM frame size (in bits) from the number of data subcarriers and the number of bits per symbol.

```
frameSize = [k*numDC 1];
```

Set the SNR vector based on the desired Eb/No range, the number of bits per symbol, and the ratio of the number of data subcarriers to the total number of subcarriers.

```
EbNoVec = (0:10)';
snrVec = EbNoVec + 10*log10(k) + 10*log10(numDC/numSC);
```

Initialize the BER and error statistics arrays.

```
berVec = zeros(length(EbNoVec),3);
errorStats = zeros(1,3);
```

Simulate the communication link over the range of Eb/No values. For each Eb/No value, the simulation runs until either `maxBitErrors` are recorded or the total number of transmitted bits exceeds `maxNumBits`.

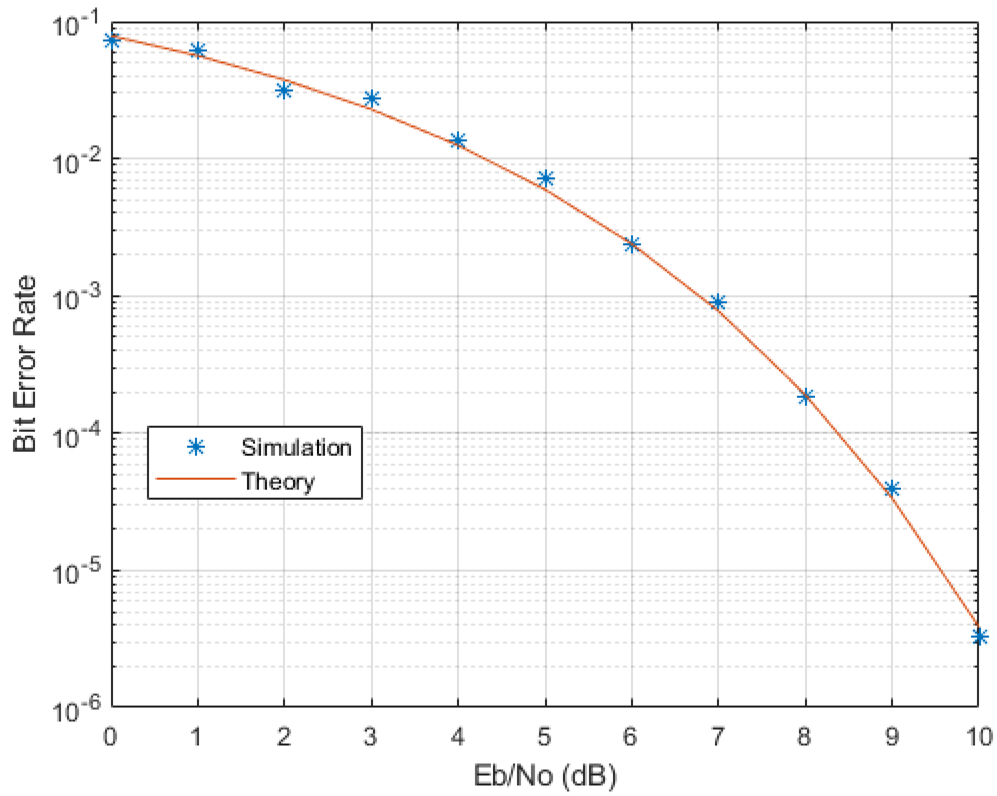
```
for m = 1:length(EbNoVec)
    snr = snrVec(m);

    while errorStats(2) <= maxBitErrors && errorStats(3) <= maxNumBits
        dataIn = randi([0,1],frameSize);              % Generate binary data
        qpskTx = qpskMod(dataIn);                     % Apply QPSK modulation
        txSig = ofdmMod(qpskTx);                      % Apply OFDM modulation
        powerDB = 10*log10(var(txSig));               % Calculate Tx signal power
        noiseVar = 10.^(0.1*(powerDB-snr));           % Calculate the noise variance
        rxSig = channel(txSig,noiseVar);              % Pass the signal through a noisy channel
        qpskRx = ofdmDemod(rxSig);                    % Apply OFDM demodulation
        dataOut = qpskDemod(qpskRx);                  % Apply QPSK demodulation
        errorStats = errorRate(dataIn,dataOut,0);     % Collect error statistics
    end

    berVec(m,:) = errorStats;                         % Save BER data
    errorStats = errorRate(dataIn,dataOut,1);         % Reset the error rate calculator
end
```

Use the `berawgn` function to determine the theoretical BER for a QPSK system.

```
berTheory = berawgn(EbNoVec,'psk',M,'nondiff');
```

Plot the theoretical and simulated data on the same graph to compare results.

```
figure
semilogy(EbNoVec,berVec(:,1),'*')
hold on
semilogy(EbNoVec,berTheory)
legend('Simulation','Theory','Location','Best')
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
grid on
hold off
```



Observe that there is good agreement between the simulated and theoretical data.

Accelerating BER Simulations Using the Parallel Computing Toolbox

This example uses Parallel Computing Toolbox to accelerate a simple, QPSK bit error rate (BER) simulation. The system consists of a QPSK modulator, a QPSK demodulator, an AWGN channel, and a bit error rate counter. In this example, four parallel processors are used.

Set the simulation parameters.

```
EbNoVec = 5:8;      % Eb/No values in dB
totalErrors = 200;  % Number of bit errors needed for each Eb/No value
totalBits = 1e7;    % Total number of bits transmitted for each Eb/No value
```

Allocate memory to the arrays used to store the data generated by the function, `helper_qpsk_sim_with_awgn`.

```
[numErrors, numBits] = deal(zeros(length(EbNoVec),1));
```

Run the simulation and determine the execution time. Only one processor will be used to determine baseline performance. Accordingly, observe that the normal for-loop is employed.

```
tic

for idx = 1:length(EbNoVec)
    errorStats = helper_qpsk_sim_with_awgn(EbNoVec, idx, ...
        totalErrors, totalBits);
    numErrors(idx) = errorStats(idx,2);
    numBits(idx) = errorStats(idx,3);
end

simBaselineTime = toc;
```

Calculate the BER.

```
ber1 = numErrors ./ numBits;
```

Rerun the simulation for the case in which Parallel Computing Toolbox is available. Create a pool of workers.

```
pool = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Determine the number of available workers from the `NumWorkers` property of `pool`. The simulation runs the range of E_b/N_0 values over each worker rather than assigning a single E_b/N_0 point to each worker as the former method provides the biggest performance improvement.

```
numWorkers = pool.NumWorkers;
```

Determine the length of `EbNoVec` for use in the nested `parfor` loop. For proper variable classification, the range of a for-loop nested in a `parfor` must be defined by constant numbers or variables.

```
lenEbNoVec = length(EbNoVec);
```

Allocate memory to the arrays used to store the data generated by the function, `helper_qpsk_sim_with_awgn`.

```
[numErrors, numBits] = deal(zeros(length(EbNoVec),numWorkers));
```

Run the simulation and determine the execution time.

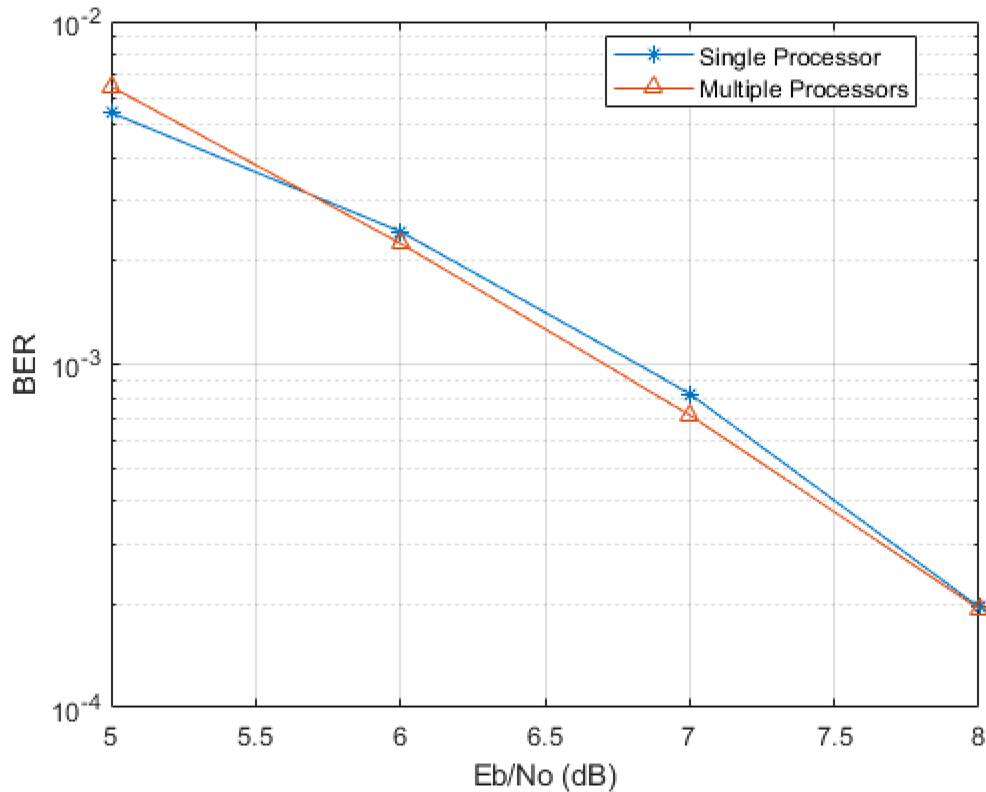
```
tic

parfor n = 1:numWorkers

    for idx = 1:lenEbNoVec
        errorStats = helper_qpsk_sim_with_awgn(EbNoVec, idx, ...
            totalErrors/numWorkers, totalBits/numWorkers);
        numErrors(idx,n) = errorStats(idx,2);
        numBits(idx,n) = errorStats(idx,3);
    end

end

simParallelTime = toc;
```

Calculate the BER. In this case, the results from multiple processors must be combined to generate the aggregate BER.

```
ber2 = sum(numErrors,2) ./ sum(numBits,2);
```

Compare the BER values to verify that the same results are obtained independent of the number of workers.

```
semilogy(EbNoVec',ber1,'-*',EbNoVec',ber2,'-^')
legend('Single Processor','Multiple Processors','location','best')
xlabel('Eb/No (dB)')
ylabel('BER')
grid
```



You can see that the BER curves are essentially the same with any variance being due to differing random number seeds.

Compare the execution times for each method.

```
fprintf(['\nSimulation time = %4.1f sec for one worker\n', ...
    'Simulation time = %4.1f sec for multiple workers\n'], ...
    simBaselineTime, simParallelTime)
```

```
Simulation time = 67.2 sec for one worker
Simulation time = 17.9 sec for multiple workers
```

In this case where four processor cores were used, the speed improvement factor was approximately four.

Iterative Design Workflow for Communication Systems

In this section...

"Simulate a basic communications system" on page 2-30
"Introduce convolutional coding and hard-decision Viterbi decoding" on page 2-34
"Improve results using soft-decision decoding" on page 2-37
"Use turbo coding to improve BER performance" on page 2-40
"Apply a Rayleigh channel model" on page 2-42
"Use OFDM-based equalization to correct multipath fading" on page 2-44
"Use multiple antennas to further improve system performance" on page 2-46
"Accelerate the simulation using MATLAB Coder" on page 2-48

This example illustrates a design workflow that represents the iterative steps for creating a wireless communications system with the Communications Toolbox. Because Communications Toolbox supports both MATLAB and Simulink, this example showcases design paths using MATLAB code and Simulink blocks. As you progress through the workflow, you may follow the design path for MATLAB, for Simulink, or for both products.

The workflow begins with a simple communications system and performs bit error rate (BER) simulations to gauge system performance. BER simulations are based on simulating a communications system with a given signal-to-noise ratio (En/No), and then calculating the corresponding bit error rate measurement to determine the number of errors in the transmitted signal. The lower the BER measurement at a given signal-to-noise ratio, the better the system performance.

This workflow starts with a simple communications system, and iteratively adds the algorithmic components necessary to build a more complicated system. These additional components include:

- Convolutional Encoding and Viterbi Decoding
- Turbo Coding
- Multipath Fading Channels
- OFDM-Based Transmission
- Multiple-Antenna Techniques

As you add components to the system, the workflow includes bit error calculations so that you can progressively examine system performance. For some components, theoretical or performance benchmarks are available. In these cases, the workflow shows both the theoretical and measured performance metric.

Simulate a basic communications system

This workflow starts with a simple QPSK modulator system that transmits a signal through an AWGN channel and calculates the bit error rate to evaluate system performance.

In MATLAB

1 CD to the following MATLAB folder:

   matlab\help\toolbox\comm\examples

2. Type `edit doc_design_iteration_basic_m` at the MATLAB command line.

   MATLAB opens a file you will use in this example. Notice that this code employs four System objects from Communications Toolbox: `comm.PSKModulator`, `comm.AWGN`, `comm.PSKDemodulator`, and `comm.ErrorRate`. For each EbNo value, the code runs in a while loop until either the specified number of errors are observed or the maximum number of bits are processed. Notice that the code executes each System object™ by calling the `step` method. The code outputs BER, defined as the ratio of the observed number of errors per number of bits processed. The subsequent MATLAB functions that this example uses have a similar structure.
3. Type `bertool` at the MATLAB command line to open the Bit Error Rate Analysis Tool.
4. When the BERTool application appears, click the **Theoretical** tab.

   The first plot that you will generate is a theoretical curve.
5. Enter `0:9` for the **EbNo range**.

   EbNo is the ratio of noise power energy per bit. The higher the value, the better the system performance. This simulation will run using different values for the ratio, between `0` and `9`.
6. Select 4 for **Modulation order**.

   The modulation order defines the number of symbols to transmit. Here, each symbol is made up of two bits.
7. Click **Plot**.

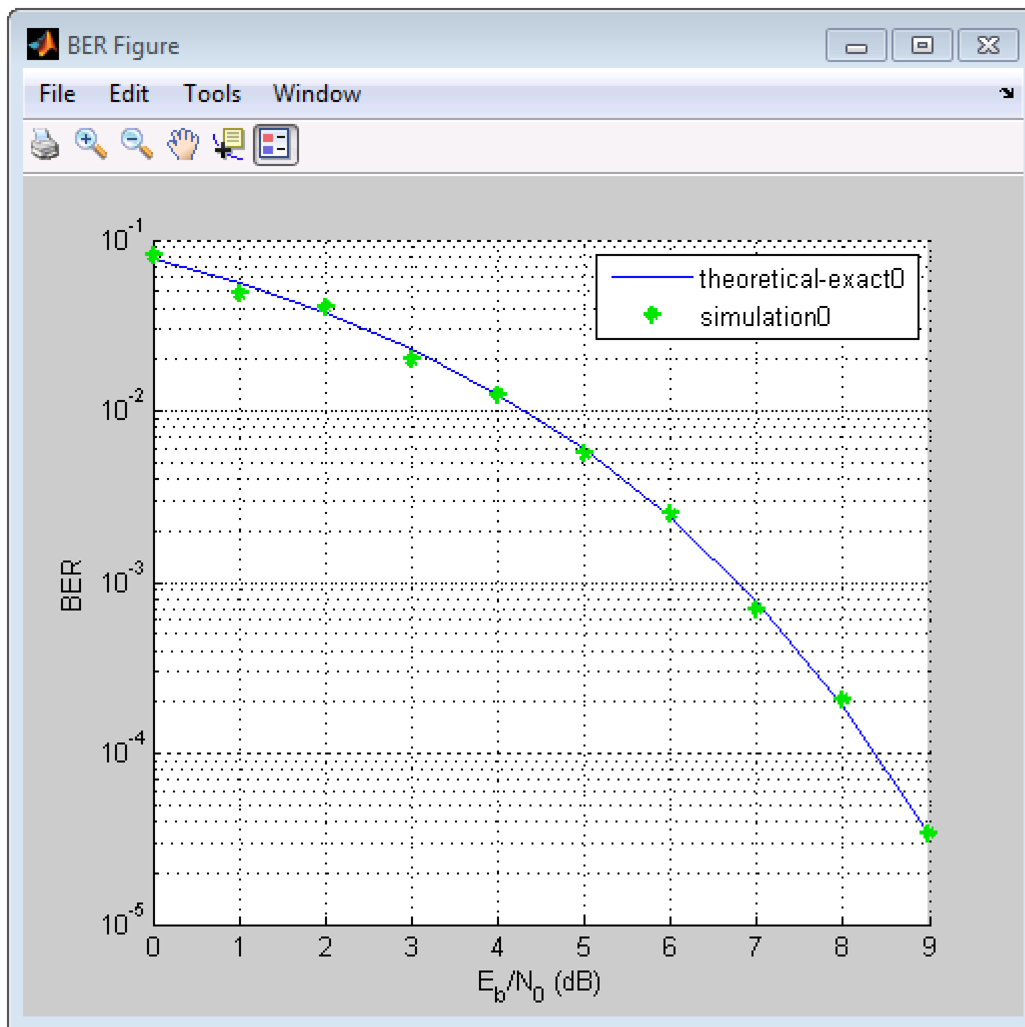
   The BERTool application generates the theoretical BER curve.
8. Click the **Monte Carlo** tab.

   Monte Carlo techniques use random sampling to compute data. Therefore, the plot for the second simulation uses random sampling.
9. Enter `0:9` for the **EbNo range**.
10. Enter `ber` for the **BER variable name**.
11. Enter `200` for the **Number of errors**.

    The **Number of errors** is one of the stop criteria for the simulation.
12. Enter `1e7` for the **Number of bits**.

    The **Number of bits** is also a stop criteria for the simulation. The simulation stops when it transmits the number of bits you specify for this parameter. In this example, the simulation either stops when it transmits 10 million bits or when it detects 200 errors.
13. Click the **Browse** button.
14. Navigate to `matlab/help/toolbox/comm/examples`, and select `doc_design_iteration_basic_m.m`.
15. Click **Run**.

    BERTool runs the simulation and generates simulation points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



Every function with two output variables and three input variables can be called using BERTool. This is how you interpret the three input variables:

- The first variable is a scalar number that corresponds to EbNo.
- The second variable is the stopping criterion based on the maximum number of errors to observe before stopping the simulation.
- The third variable is the stop criterion based on the maximum number of bits to process before observe before stopping the simulation.

In Simulink

1. Type `bertool` at the MATLAB command line to open the Bit Error Rate Analysis Tool.
2. When the BERTool application appears, click the **Theoretical** tab.
3. Enter `0:9` for the **EbNo range**.

   EbNo is the ratio of noise power energy per bit. The higher the value, the better the system performance. This simulation will run using different values for the ratio, between `0` and `9`.
4. Select 4 for **Modulation order**.

The modulation order defines the number of symbols to transmit. Here, each symbol is made up of two bits.

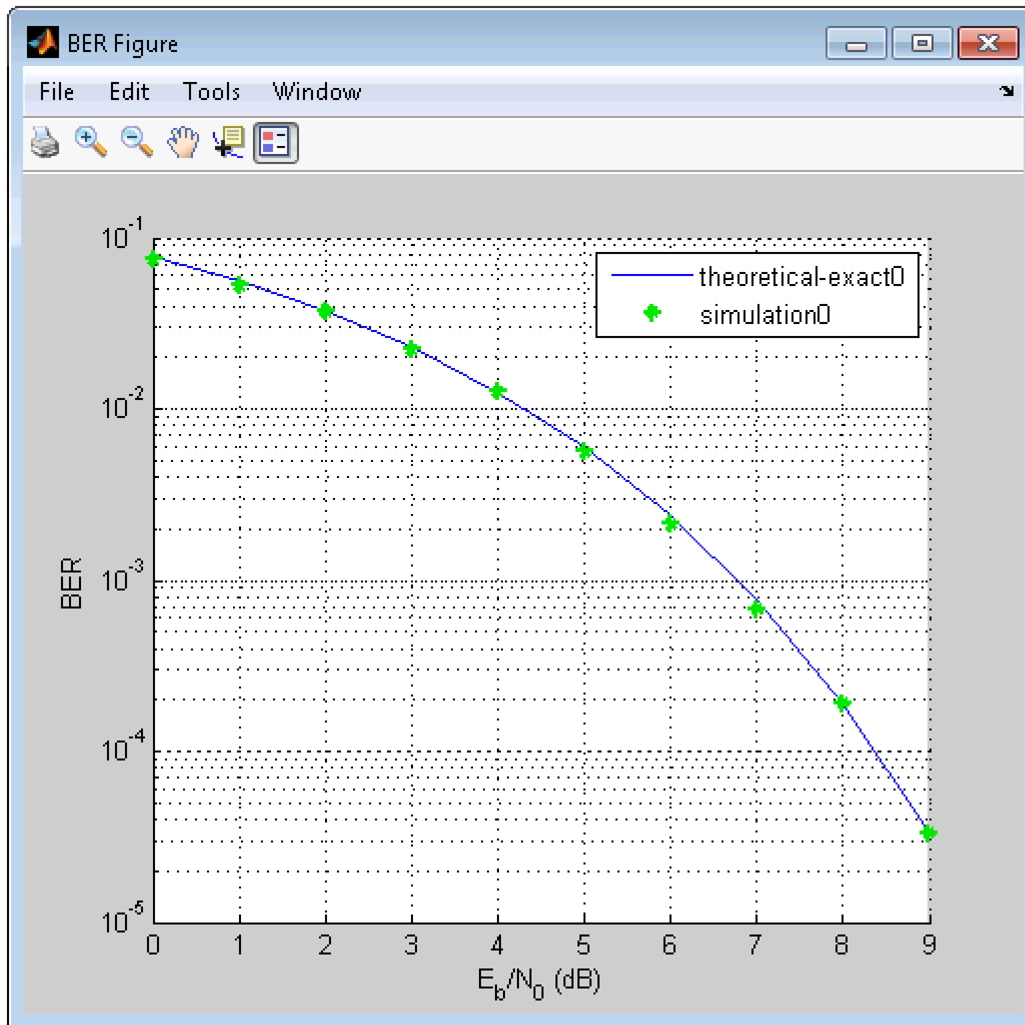
5. Click **Plot**.

   The BERTool application generates the theoretical BER curve.
6. Click the **Monte Carlo** tab.
7. Enter `0:9` or the **EbNo range**.
8. Enter `ber` for the **BER variable name**.
9. Enter `200` for the **Number of errors**.

   The **Number of errors** is one of the stop criteria for the simulation. The simulation stops when it reaches either the **Number of errors** or the **Number of bits**.
10. Enter `1e7` for the **Number of bits**.

    The **Number of bits** is also a stop criteria for the simulation. The simulation stops when it transmits the number of bits you specify for this parameter or when it reaches the **Number of errors**. In this example, the simulation either stops when it transmits 10 million bits or when it detects 200 errors.
11. Click the **Browse** button, select **All Files** for the **Files of type** field.
12. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_basic.slx` and click **Run**.

    BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



Introduce convolutional coding and hard-decision Viterbi decoding

Modify the basic communications model to include forward error correction. Adding forward error correction to the basic communications model improves system performance. In forward error correction, the transmitter sends redundant bits, along with the message bits, through a wireless channel. When the receiver accepts the transmitted signal, it uses the redundancy bits to detect and correct errors that the channel may have introduced.

This section of the design workflow adds a convolutional encoder and a Viterbi decoder to the communication system. This communications system uses hard-decision Viterbi decoding. In hard-decision Viterbi decoding, the demodulator maps the received signal to bits, and then passes the bits to the Viterbi decoder for error correction.

In MATLAB

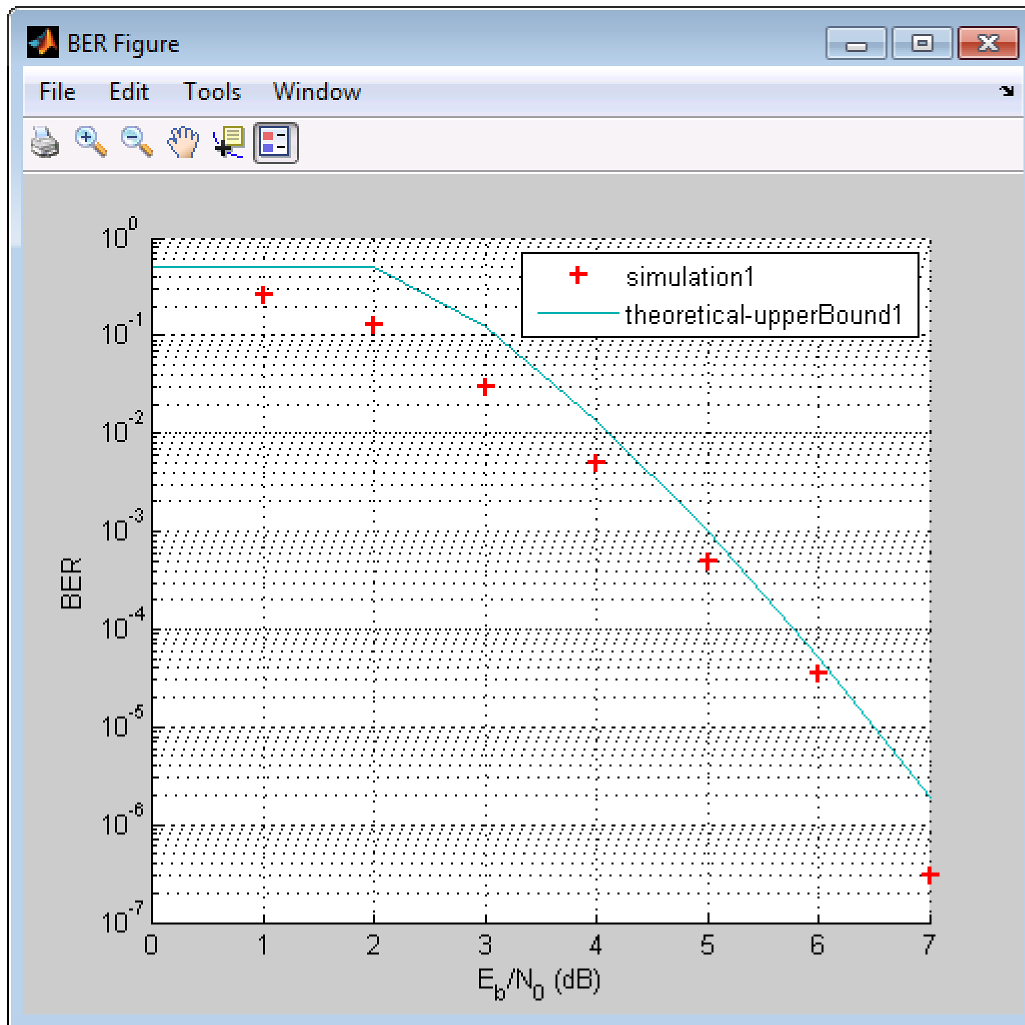
In this iteration of the design workflow, the MATLAB file you use starts from where the one in the previous section ended. This file adds two additional System objects to the communications system, `comm.ConvolutionalEncoder` and `comm.ViterbiDecoder`. The overall structure of the code doesn't change; it simply contains additional functionality.

1. Access the BERTool application.
2. Clear the **Plot** check boxes for the two plots BERTool generated in the previous step.
3. Click **Theoretical**.
4. Enter `0:7` for the **EbNo range**.
5. Select **Convolutional** for the **Channel Coding**.
6. Select **Hard** for the **Decision method**.

   This example uses hard-decision Viterbi decoding. The demodulator maps the received signal to bits, and then passes the bits to the Viterbi decoder for error correction.
7. Click **Plot**.

   The BERTool application generates the theoretical BER curve.
8. Click **Monte Carlo**.
9. Enter `0:7` for the **EbNo range**.
10. Enter `200` for the **Number of errors**.
11. Enter `1e7` for the **Number of bits**.
12. Click the **Browse** button.
13. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_m.m` and click **Open**.
14. Click **Run**.

    BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



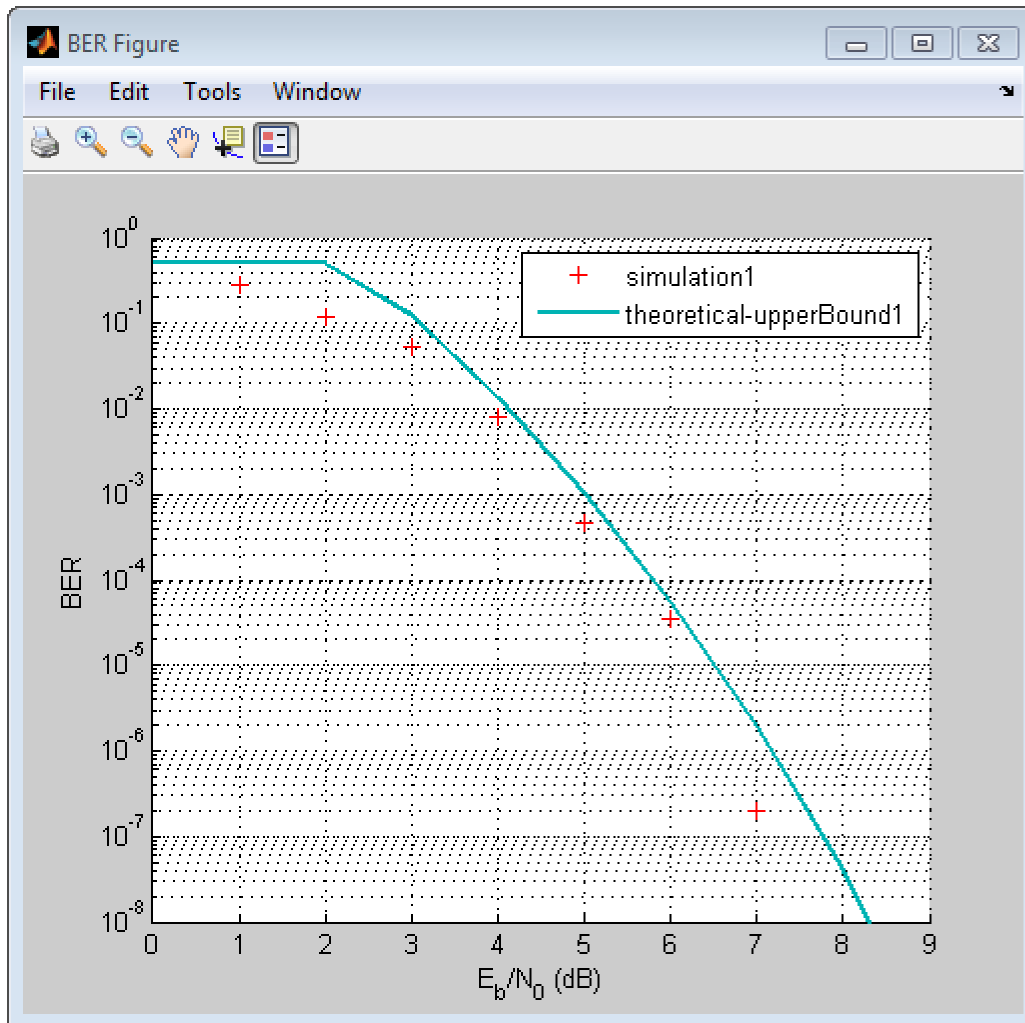
In Simulink

1. Access the BERTool application.
2. Click the **Theoretical** tab.
3. Enter `0:7` for the **EbNo range**.
4. Select **Convolutional** for the **Channel Coding**.
5. Select **Hard** for the **Decision method**.

   This example uses hard-decision Viterbi decoding. The demodulator maps the received signal to bits, and then passes the bits to the Viterbi decoder for error correction.
6. Click **Plot**.

   The BERTool application generates the theoretical BER curve.
7. Click the **Monte Carlo** tab.
8. Enter `0:7` for the **EbNo range**.
9. Enter `200` for the **Number of errors**.
10. Enter `1e7` for the **Number of bits**.
11. Click the **Browse** button, select **All Files** for the **Files of type** field.
12. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi.slx` and click **Open**.

13 click **Run**.BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



Improve results using soft-decision decoding

Use soft-decision decoding to improve BER performance. The previous section of this workflow uses hard-decision demodulation and hard-decision Viterbi decoding – processes that map symbols to bits. This section of the workflow uses soft-decision demodulation and soft-decision Viterbi decoding. In soft-decision demodulation, the received symbols are not mapped to bits. Instead, the symbols are mapped to log-likelihood ratios. When the Viterbi decoder processes log-likelihood ratios (LLR), the BER performance of the system improves.

In MATLAB

1. Access the BERTool application.
2. Clear the **Plot** check boxes for the two plots BERTool generated in the previous step.
3. Click **Theoretical**.
4. Enter `0:5` for the **EbNo range**.
5. Select **Soft** for the **Decision method**.

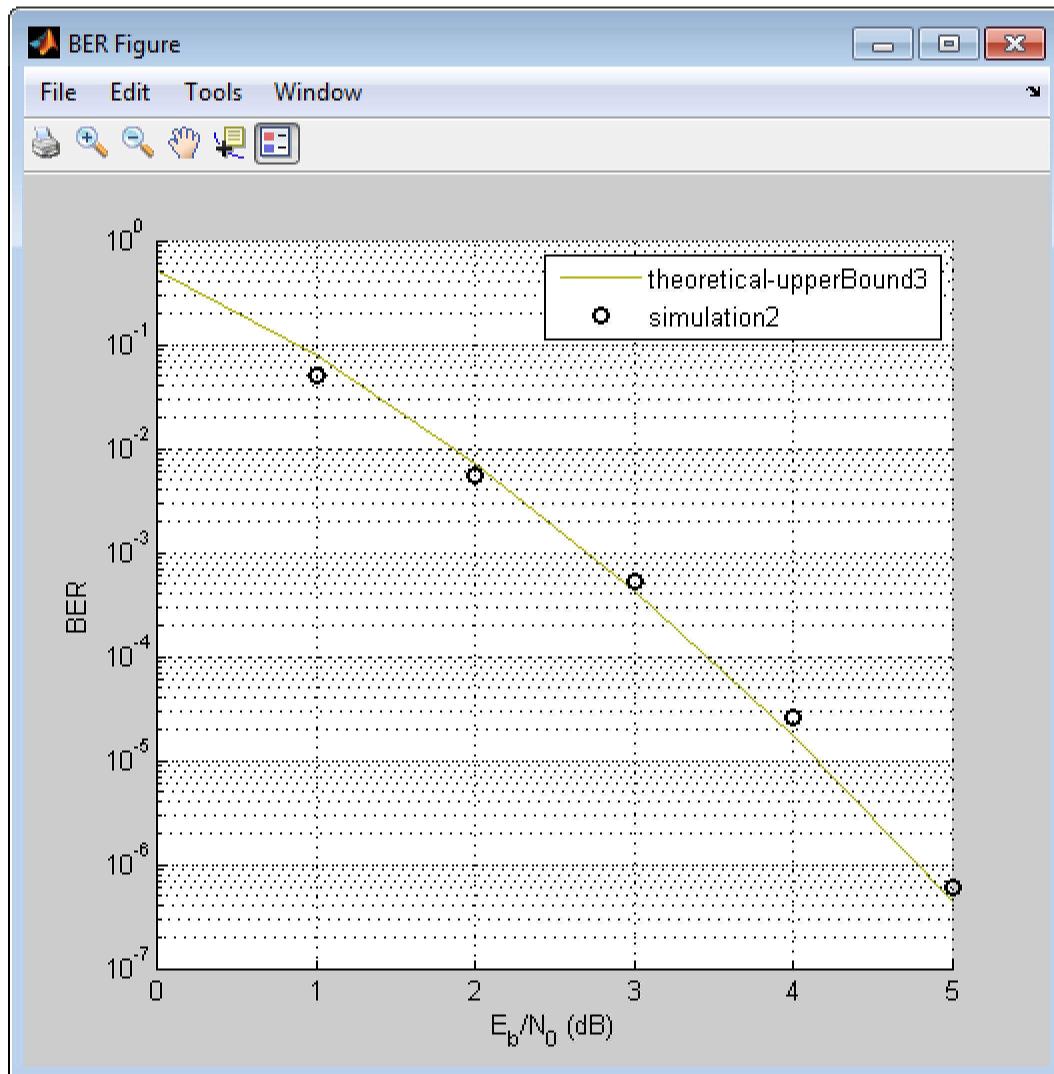
This example uses soft-decision Viterbi decoding. The demodulator maps the received signal to log likelihood ratios, improving BER performance results.

6 Click **Plot**.

   The BERTool application generates the theoretical BER curve.

7 Click **Monte Carlo**.
8 Enter `0:5` for the **EbNo range**.
9 Enter `200` for the **Number of errors**.
10 Enter `1e7` for the **Number of bits**.
11 Click the **Browse** button.
12 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_soft_m.m` and click **Run**.

   BERTool runs the simulation and generates the actual simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.

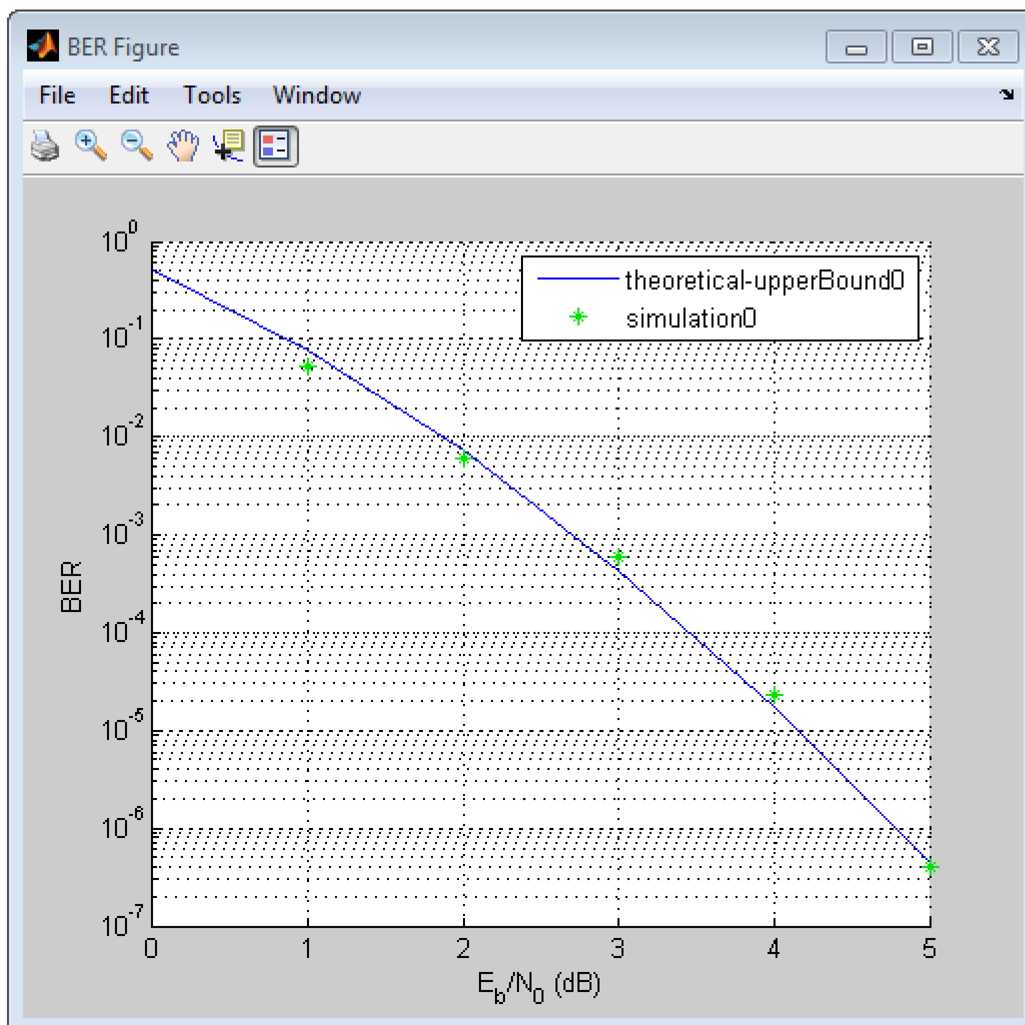


In Simulink

1. Access the BERTool application.
2. Clear the **Plot** check boxes for the two plots BERTool generated in the previous step.
3. Click **Theoretical**.
4. Enter `0:5`for the **EbNo range**.
5. Select **Soft** for the **Decision method**.

   This example uses soft-decision Viterbi decoding. The demodulator maps the received signal to log likelihood ratios, improving BER performance results.
6. Click **Plot**.

   The BERTool application generates the theoretical BER curve.
7. Click **Monte Carlo**.
8. Enter `0:5` for the **EbNo range**.
9. Enter `200` for the **Number of errors**.
10. Enter `1e7` for the **Number of bits**.
11. Click the **Browse** button, select **All Files** for the **Files of type** field.
12. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_soft.slx` and click **Run**.



When you plot the soft-decision theoretical curve, you will observe BER curve improvements of about 2 dB relative to the hard-decision decoding. Notice that the simulation results also reflects a similar BER improvement.

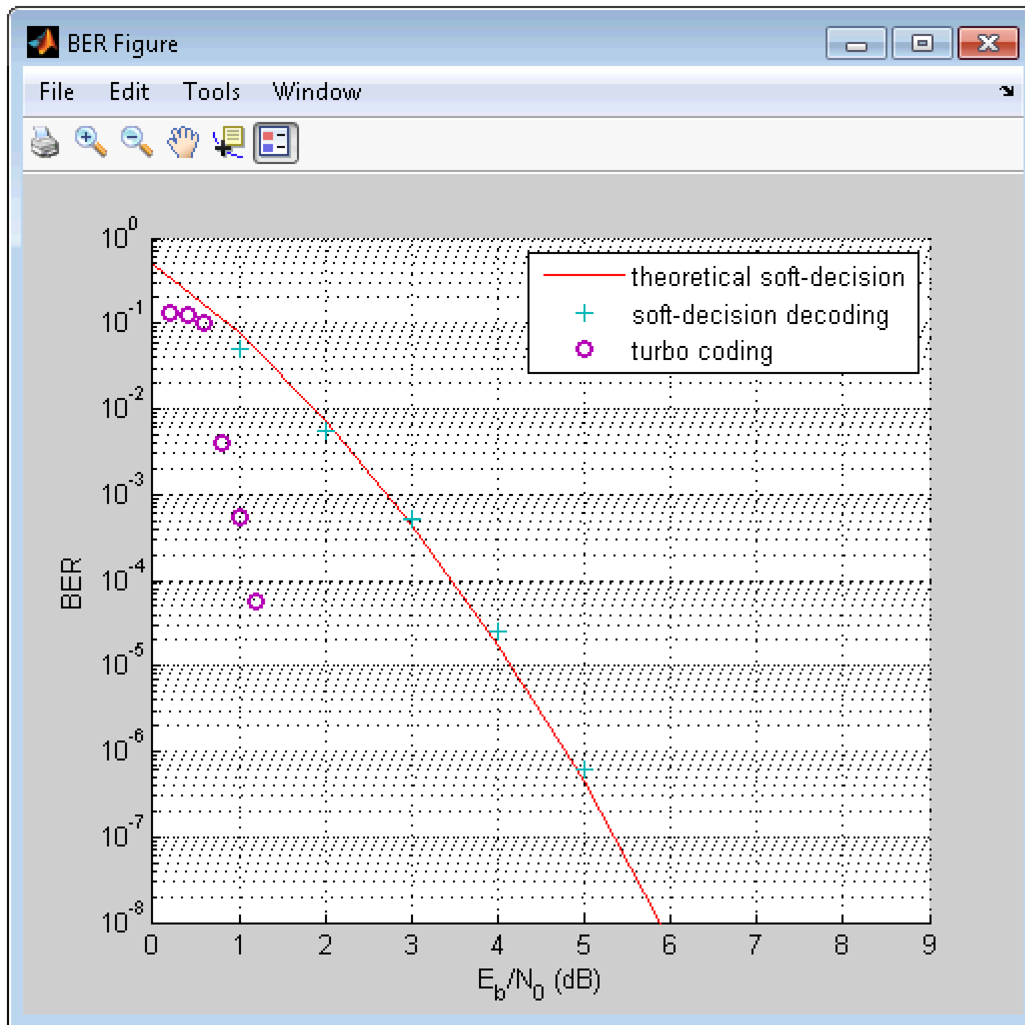
Use turbo coding to improve BER performance

Turbo codes substantially improve BER performance over soft-decision Viterbi decoding. Turbo coding uses two convolutional encoders in parallel at the transmitter and two a posteriori probability (APP) decoders in series at the receiver. This example uses a rate 1/3 turbo coder. For each input bit, the output has 1 systematic bit and 2 parity bits, for a total of three bits. Turbo coders achieve BER performances at much lower SNR values than convolutional encoders. As a result, this iteration uses a lower range of EbNo values than the previous section.

In MATLAB

1. Access the BERTool application.
2. Click the **Monte Carlo** tab.
3. Enter `0:0.2:1.2` for the **EbNo range**.
4. Enter `200` for the **Number of errors**.
5. Enter `1e7` for the **Number of bits**.
6. Click the **Browse** button.
7. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_zTurbo_soft_m.m` and click **Run**.

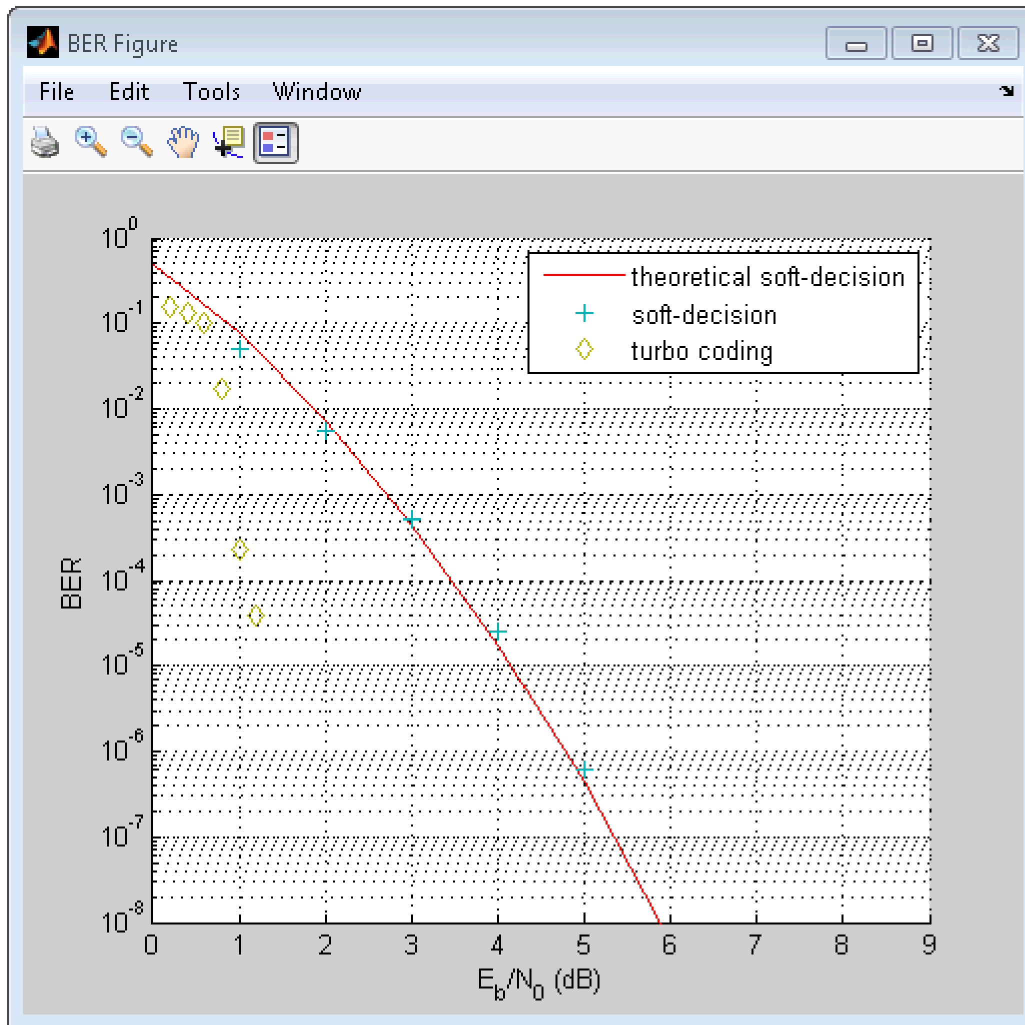
   BERTool runs the simulation and generates simulated points along the BER curve.



In Simulink

1. Access the BERTool application.
2. Clear the **Plot** check boxes for the last plot BERTool generated in the previous section.
3. Click the **Monte Carlo** tab.
4. Enter `0:0.2:1.2` for the **EbNo range**.
5. Enter `200` for the **Number of errors**.
6. Enter `1e7` for the **Number of bits**.
7. Click the **Browse** button, select **All Files** for the **Files of type** field.
8. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_turbo.slx` and click **Run**.

   BERTool runs the simulation and generates simulated points along the BER curve.



Apply a Rayleigh channel model

The previous design iterations model narrowband communications systems that can be adequately represented using an AWGN channel. However, high data rate communications systems require a wideband channel. Wideband communications channels are highly susceptible to the effects of multipath propagation, which introduces intersymbol interference (ISI). Therefore, you must model wideband channels as multipath fading channels. This iteration of the design workflow uses a multipath fading Rayleigh channel, which assumes no direct line-of-sight between the transmitter and receiver.

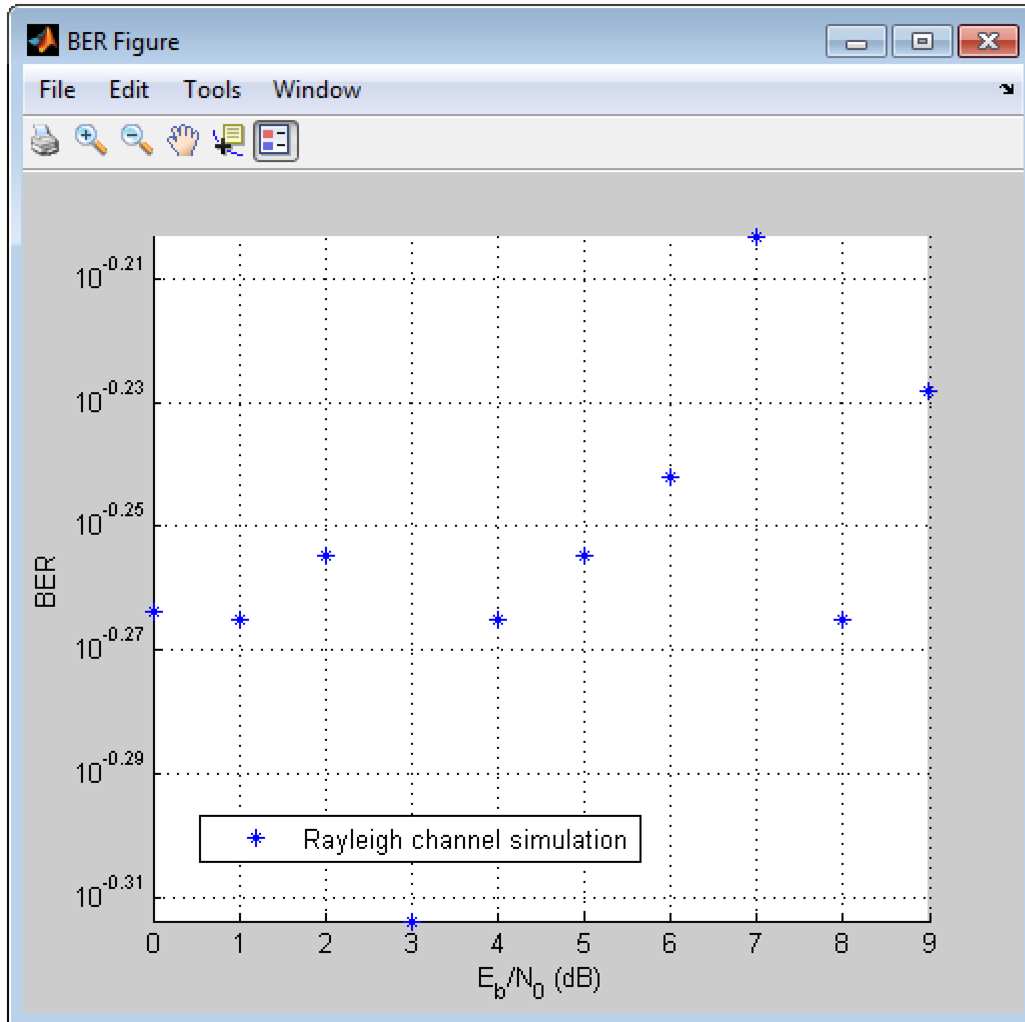
In MATLAB

1. Access the BERTool application.
2. Clear the **Plot** check box for the plot BERTool generated in the previous step.
3. Click **Monte Carlo**.
4. Enter `0:9` for the **EbNo range**.
5. Enter `200` for the **Number of errors**.
6. Enter `1e7` for the **Number of bits**.

7 Click the **Browse** button.

8 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_rayleigh_m.m` and click **Run**.

BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



In the presence of multipath fading, the BER performance reduces to that of a binary channel with a consistent value of one-half. To correct the effect of multipath fading, you must add equalization to the communications system.

In Simulink

1 Access the BERTool application.

2 Clear the **Plot** check box to clear the plot BERTool generated in the previous step.

3 Click **Monte Carlo**.

4 Enter `0:7` for the **EbNo range**.

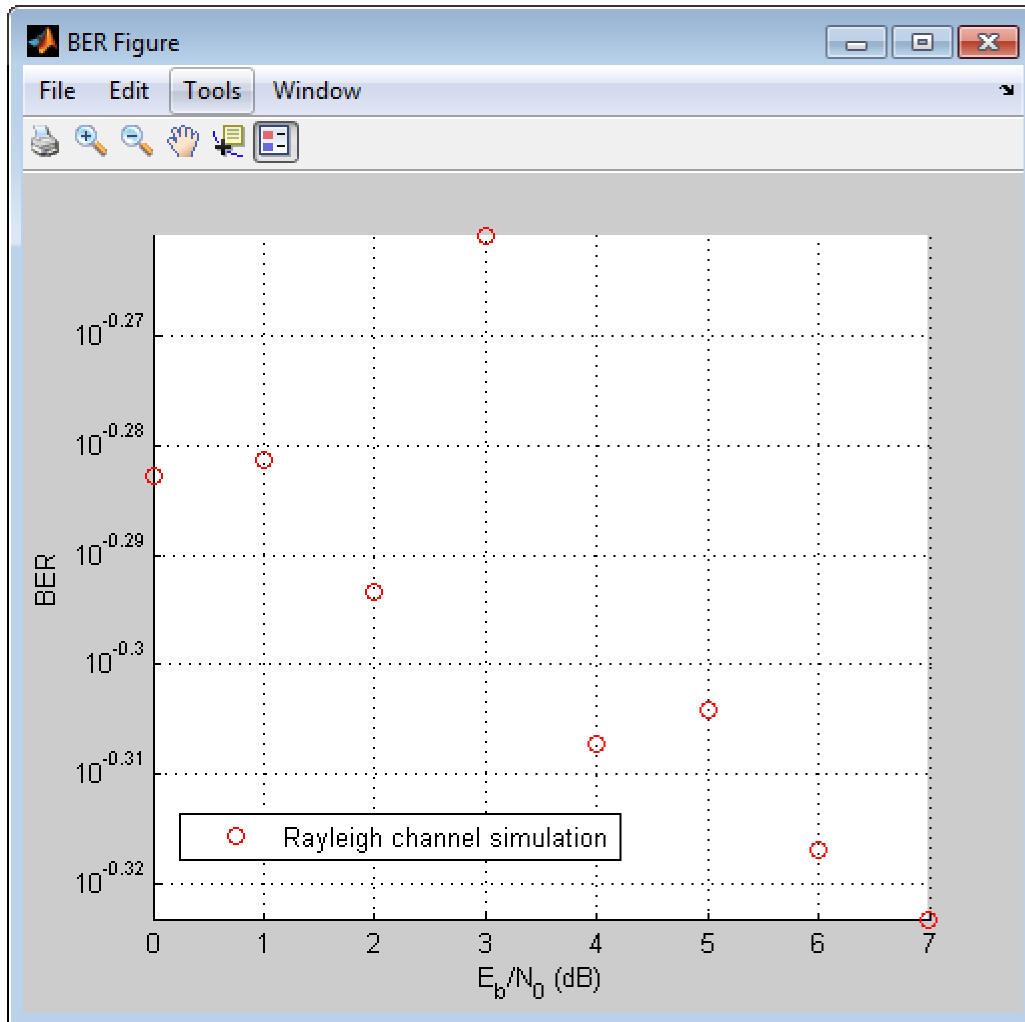
5 Enter `200` for the **Number of errors**.

6 Enter `1e7` for the **Number of bits**.

7 Click the **Browse** button, select **All Files** for the **Files of type** field.

8 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_rayleigh.slx` and click **Run**.

BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



In the presence of multipath fading, the BER performance reduces to that of a binary channel with a consistent value of one-half. To correct the effect of multipath fading, you must add equalization to the communications system.

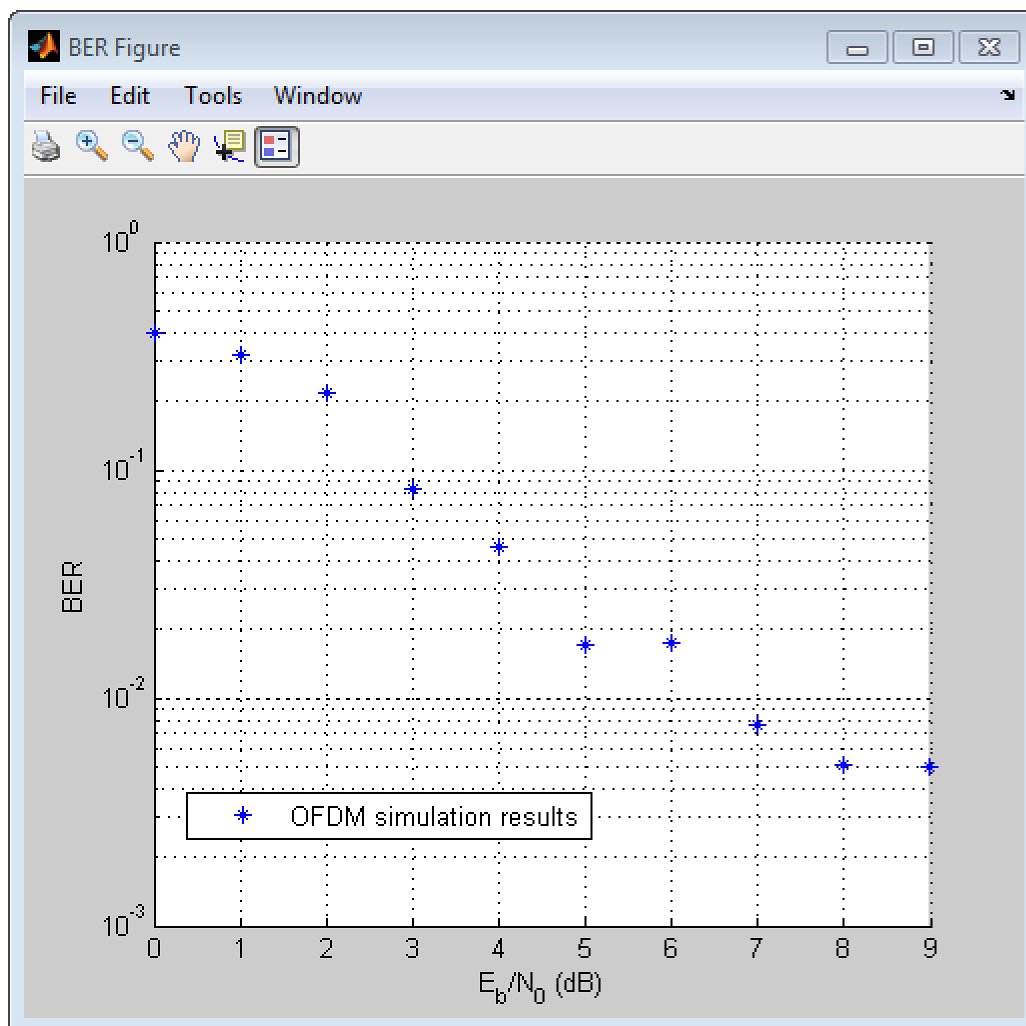
Use OFDM-based equalization to correct multipath fading

Use orthogonal frequency-division multiplexing (OFDM) to compensate for the multipath fading effect introduced by the Rayleigh fading channel. OFDM transmission schemes provides an effective way to perform frequency domain equalization. This design iteration introduces an OFDM transmitter, an OFDM receiver, and a frequency domain equalizer to the communications system.

In MATLAB

1 Access the BERTool application.
2 Clear the **Plot** check boxes for the simulation plot generated in the previous step.
3 Click the **Monte Carlo** tab.
4 Enter `0:9` for the **EbNo range**.
5 Enter `6000` for the **Number of errors**.
6 Enter `1e7` for the **Number of bits**.
7 Click the **Browse** button.
8 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_Rayleigh_OFDM_m.m` and click **Run**.

   BERTool runs the simulation and generates simulated points along the BER curve.

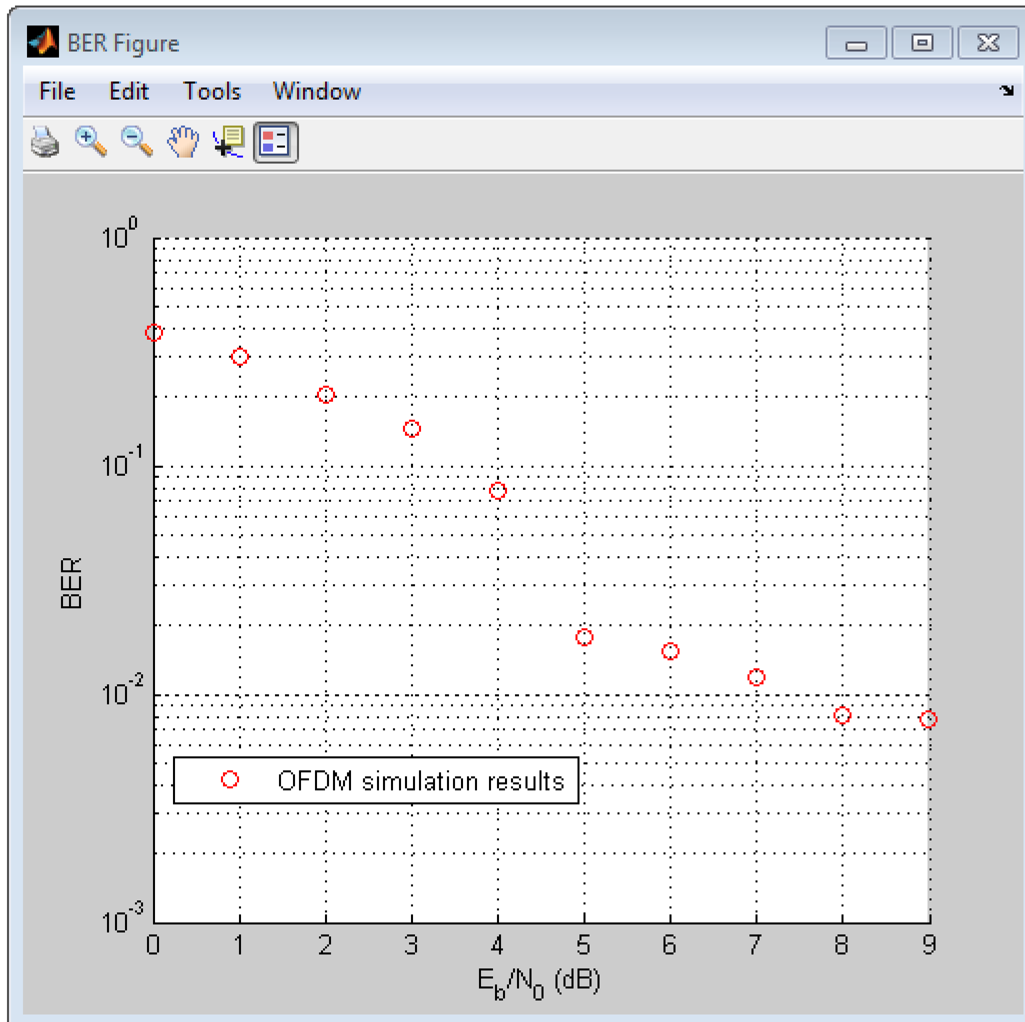


In Simulink

1 Access the BERTool application.
2 Clear the **Plot** check boxes for the plots BERTool generated in the previous step.
3 Click the **Monte Carlo** tab.

4 Enter `0:9` for the **EbNo range**.
5 Enter `6000` for the **Number of errors**.
6 Enter `5e7` for the **Number of bits**.
7 Click the **Browse** button, select **All Files** for the **Files of type** field.
8 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_rayleigh_OFDM.slx` and click **Run**.

BERTool runs the simulation and generates simulated points. Compare the simulation BER curve with the theoretical BER curve.



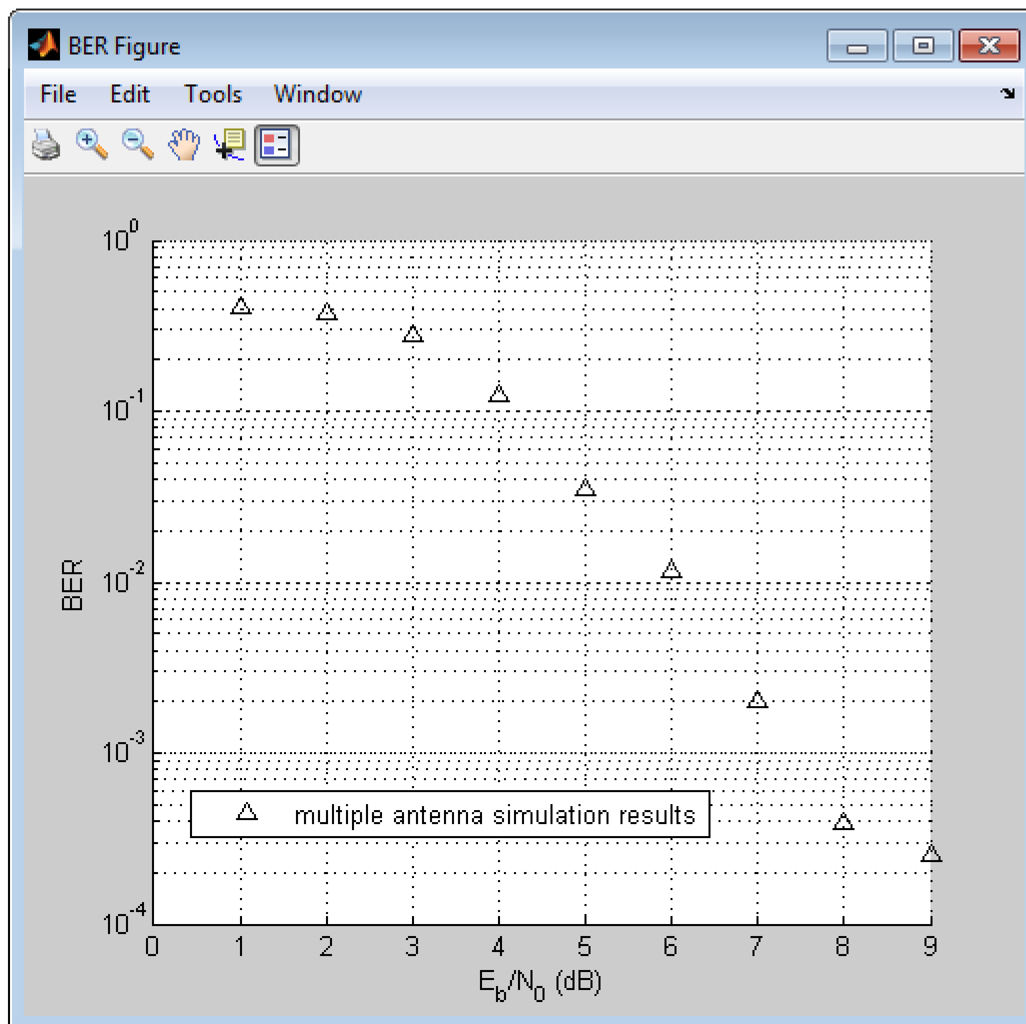
Use multiple antennas to further improve system performance

Simultaneously transmitting copies of a signal using multiple antennas can significantly increase the likelihood that the receiver correctly recovers the transmitted signal. This phenomenon is known as transmit diversity. However, this performance improvement comes at the expense of introducing additional computational complexity in the receiver.

In MATLAB

1. Access the BERTool application.
2. Clear the **Plot** check box to clear the simulation plot generated in the previous step.
3. Click the **Monte Carlo** tab.
4. Enter `0:9` for the **EbNo range**.
5. Enter `1000` for the **Number of errors**.
6. Enter `1e7` for the **Number of bits**.
7. Click the **Browse** button.
8. Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m.m` and click **Run**.

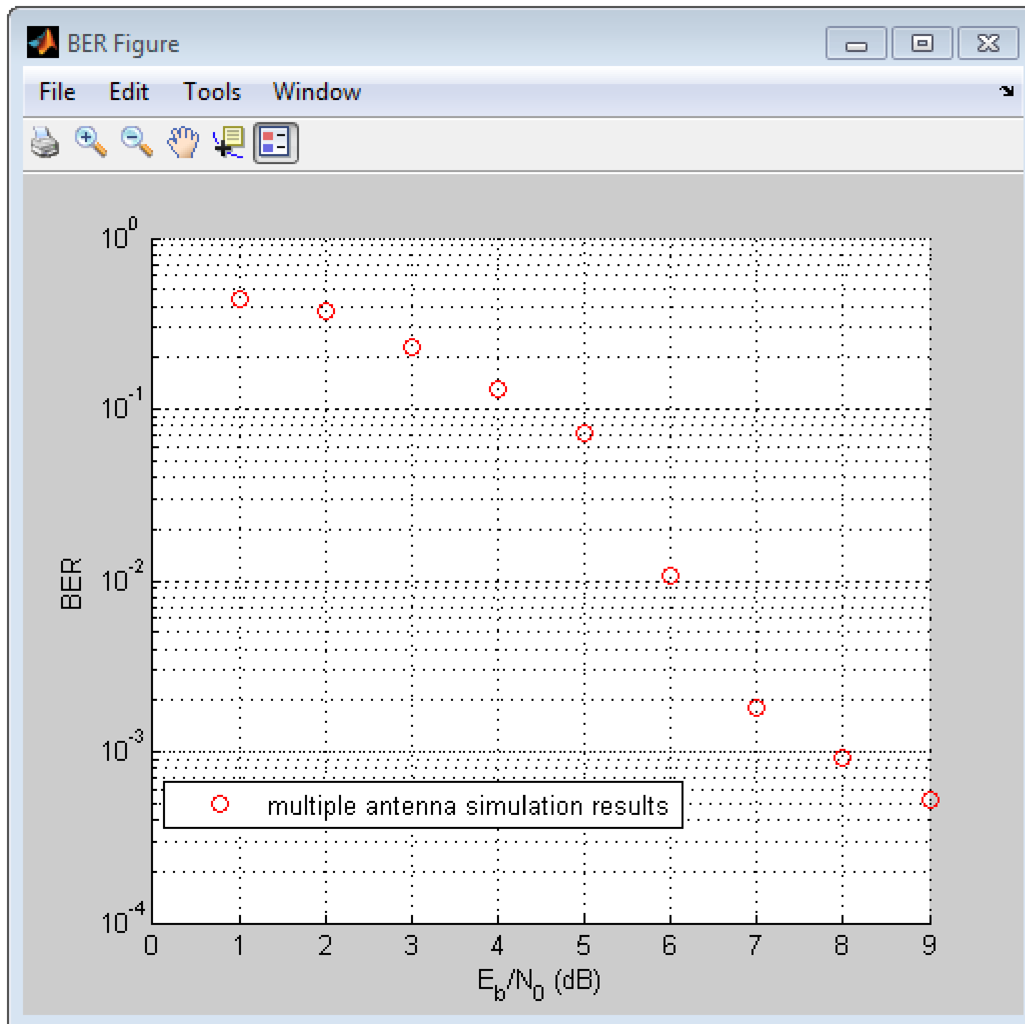
   BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



In Simulink

1. Access the BERTool application.
2. Click the **Monte Carlo** tab.

3 Enter `0:9` for the **EbNo range**.
4 Enter `700` for the **Number of errors**.
5 Enter `1e7` for the **Number of bits**.
6 Click the **Browse** button, select **All Files** for the **Files of type** field.
7 Navigate to `matlab/help/toolbox/comm/examples`, select `doc_design_iteration_viterbi_rayleigh_OFDM_MIMO.slx` and click **Run**.



Accelerate the simulation using MATLAB Coder

All of the functions and System objects that this design iteration workflow uses support C code generation. If you have a MATLAB Coder™ license, you can accelerate simulation speed by generating a .mex file using the `codegen` command.

In MATLAB

1 Copy the doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m.m file to a folder that is not on the MATLAB path. For example, C:\Temp.
2 Change your working directory to the folder you just created.

3. Execute the following commands to set a numerical value for each of the input arguments in the doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m function. For example:

```
EbNo=1;
MaxNumErrs=200;
MaxNumBits=1e7;
```

4. Execute the codegen command to generate the executable MATLAB file.

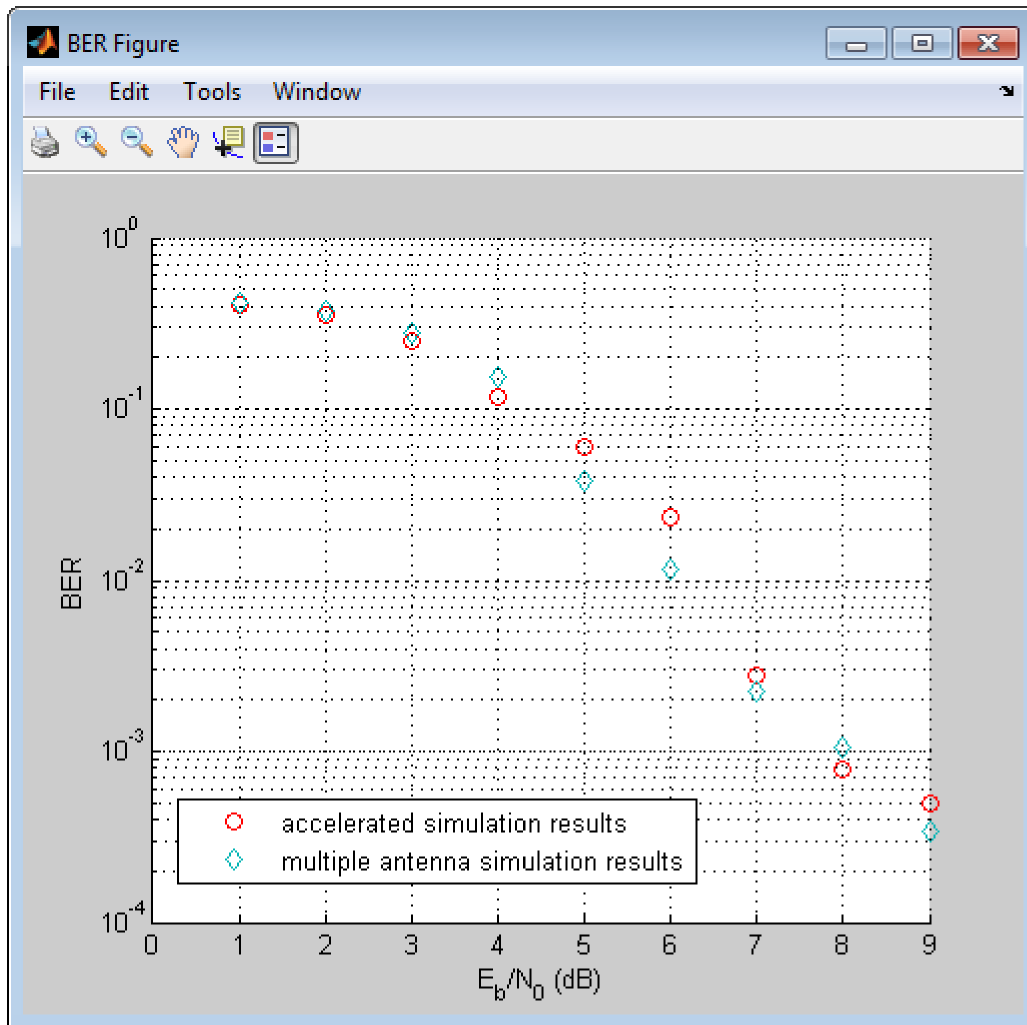
```
codegen -args {EbNo,MaxNumErrs,MaxNumBits}
doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m
```

5. The file extension of the MATLAB executable file that gets generated depends upon your operating system. For example, on 64–bit Windows® the file extension will be .mexw64, and the full file name will be doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m_mex.mexw64.

   If you run the mex file you just generated in BERTool, you will obtain the simulation results more quickly.
6. Access the BERTool application.
7. Click the **Monte Carlo** tab.
8. Enter `0:9` for the **EbNo range**.
9. Enter `700` for the **Number of errors**.
10. Enter `1e7` for the **Number of bits**.
11. Click the **Browse** button, and select `All Files`.

    Navigate to folder you created in step 1 and click **Run**.

    BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the previous curve. Any variation in the BER curve of the mex file and the MATLAB file from which it was generated is related to the seed of the random number generator and is statistically insignificant. In this example, BERTool generates the curve much more quickly when you use MATLAB Coder to generate C code. Notice that BERTool generates similar BER results in about 1/4 of the time that it took for the original simulation took to complete.


BER Figure
File
Edit
Tools
Window
BER
E_b/N_0 (dB)
accelerated simulation results
multiple antenna simulation results
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
1
2
3
4
5
6
7
8
9

What Is Different About Using Communications Toolbox Online?

Visualization and Measurements

- "Scatter Plot and Eye Diagram with MATLAB Functions" on page 3-2
- "EVM and MER Measurements with Simulink" on page 3-6
- "ACPR and CCDF Measurements with MATLAB System Objects" on page 3-13

Scatter Plot and Eye Diagram with MATLAB Functions

This example shows how to visualize signal behavior through the use of eye diagrams and scatter plots. The example uses a QPSK signal which is passed through a square-root raised cosine (RRC) filter.

Scatter Plot

Set the RRC filter, modulation scheme, and plotting parameters.

```
span = 10;          % Filter span
rolloff = 0.2;      % Rolloff factor
sps = 8;            % Samples per symbol
M = 4;              % Modulation alphabet size
k = log2(M);        % Bits/symbol
phOffset = pi/4;    % Phase offset (radians)
n = 1;              % Plot every nth value of the signal
offset = 0;         % Plot every nth value of the signal, starting from offset+1
```

Create the filter coefficients using the `rcosdesign` function.

```
filtCoeff = rcosdesign(rolloff,span,sps);
```

Generate random symbols for an alphabet size of `M`.

```
rng default
data = randi([0 M-1],5000,1);
```

Apply QPSK modulation.

```
dataMod = pskmod(data,M,phOffset);
```

Filter the modulated data.

```
txSig = upfirdn(dataMod,filtCoeff,sps);
```

Calculate the SNR for an oversampled QPSK signal.

```
EbNo = 20;
snr = EbNo + 10*log10(k) - 10*log10(sps);
```

Add AWGN to the transmitted signal.

```
rxSig = awgn(txSig,snr,'measured');
```

Apply the RRC receive filter.

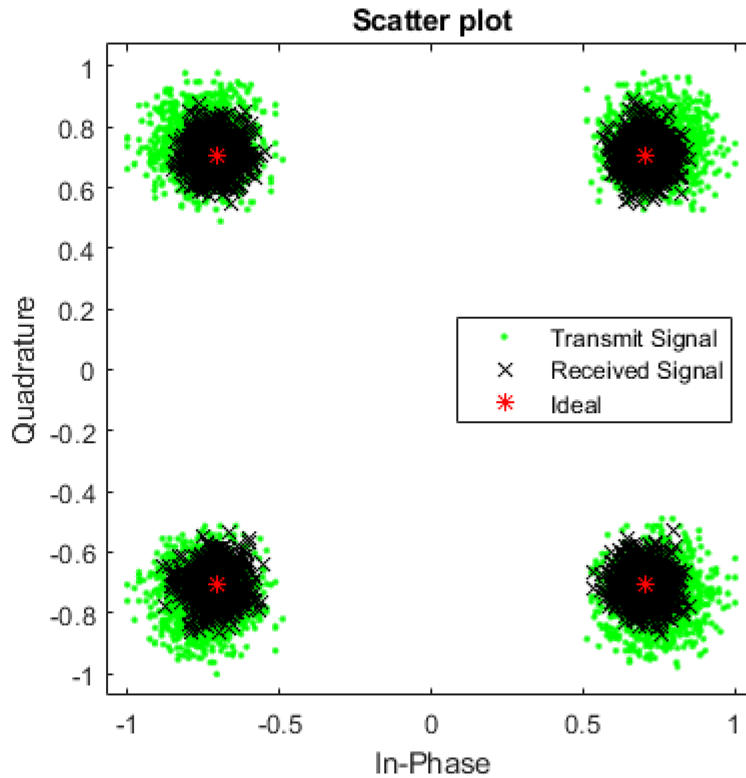
```
rxSigFilt = upfirdn(rxSig, filtCoeff,1,sps);
```

Demodulate the filtered signal.

```
dataOut = pskdemod(rxSigFilt,M,phOffset,'gray');
```

Use the `scatterplot` function to show scatter plots of the signal before and after filtering. You can see that the receive filter improves performance as the constellation more closely matches the ideal values. The first `span` symbols and the last `span` symbols represent the cumulative delay of the two filtering operations and are removed from the two filtered signals before generating the scatter plots.

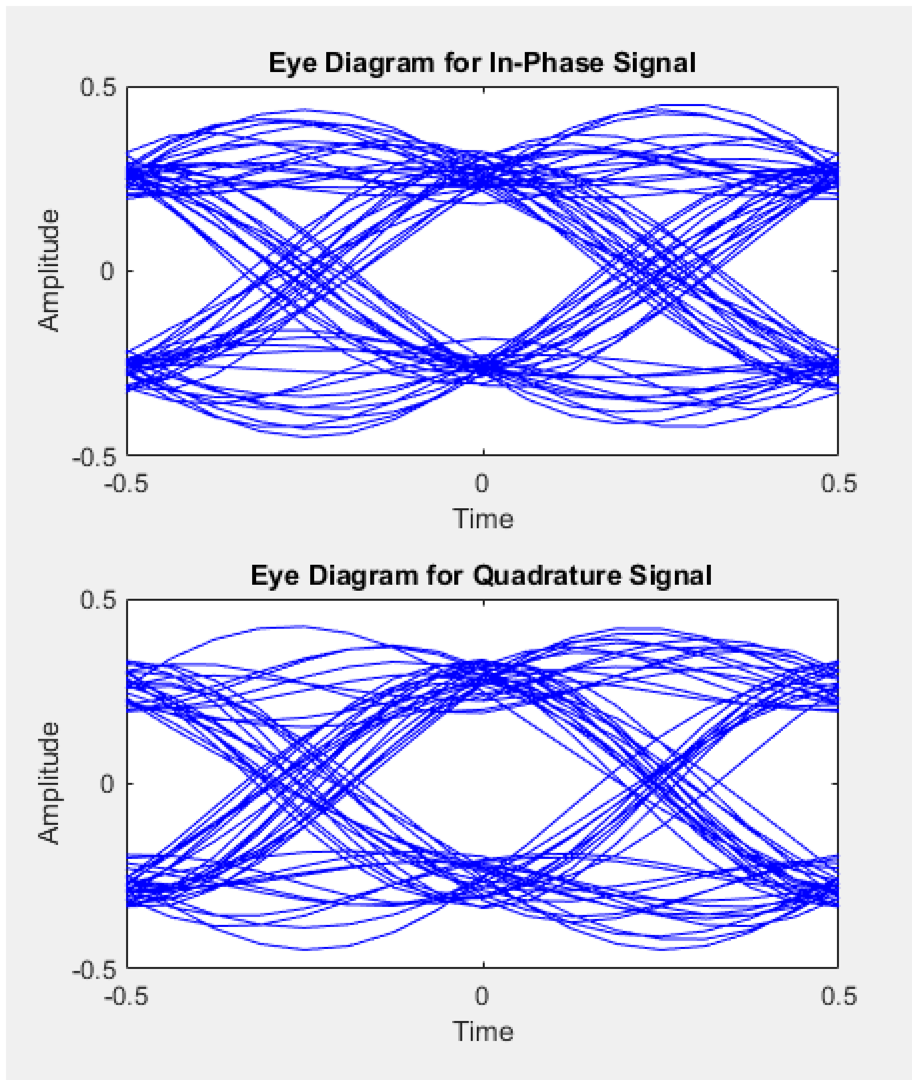
```
h = scatterplot(sqrt(sps)*txSig(sps*span+1:end-sps*span),sps,offset,'g.');
hold on
scatterplot(rxSigFilt(span+1:end-span),n,offset,'kx',h)
scatterplot(dataMod,n,offset,'r*',h)
legend('Transmit Signal','Received Signal','Ideal','location','best')
```



Eye Diagram

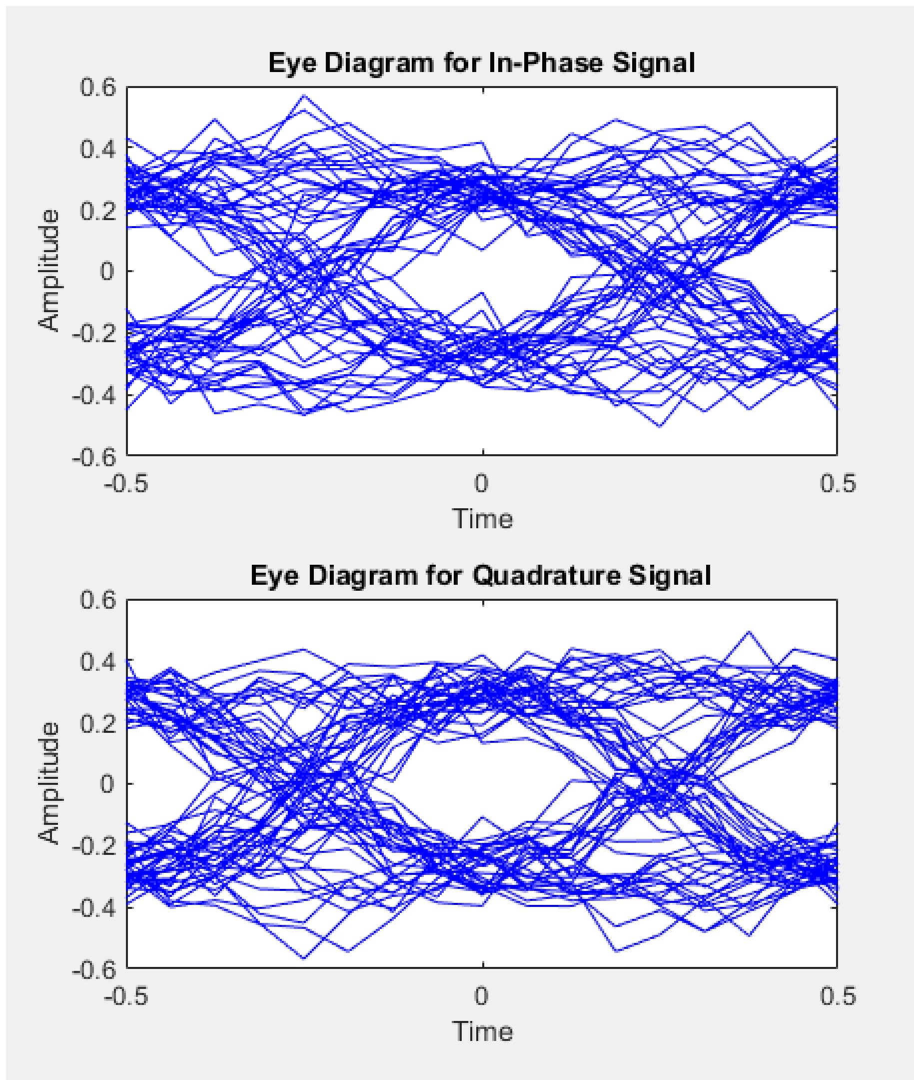
Display 1000 points of the transmitted signal eye diagram over two symbol periods.

```
eyediagram(txSig(sps*span+1:sps*span+1000),2*sps)
```



Display 1000 points of the received signal eye diagram.

```
eyediagram(rxSig(sps*span+1:sps*span+1000),2*sps)
```



Observe that the received eye diagram begins to close due to the presence of AWGN. Moreover, the filter has finite length which also contributes to the non-ideal behavior.

EVM and MER Measurements with Simulink

This model shows how error vector magnitude (EVM) and modulation error rate (MER) measurements are made using Simulink blocks.

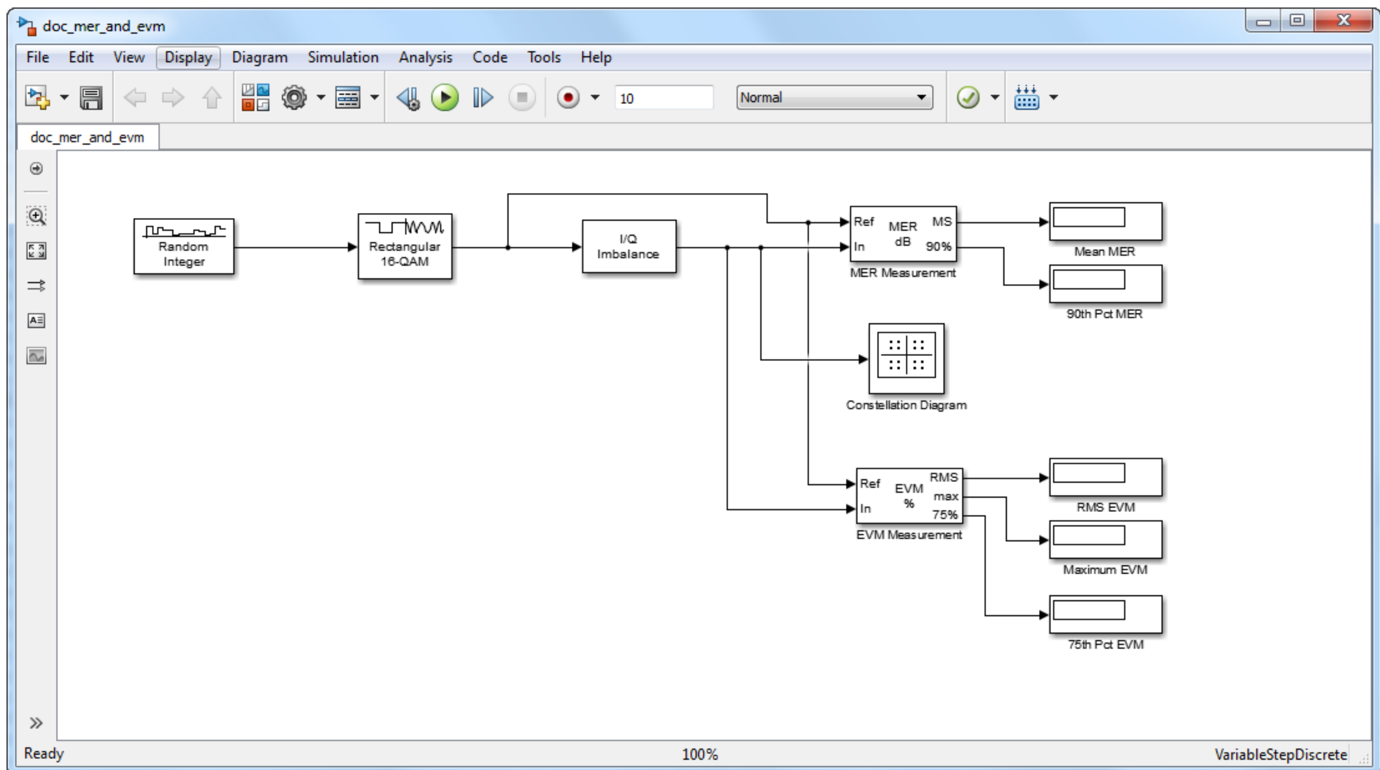
Load the model doc_mer_and_evm from the MATLAB command prompt.

```
doc_mer_and_evm
```

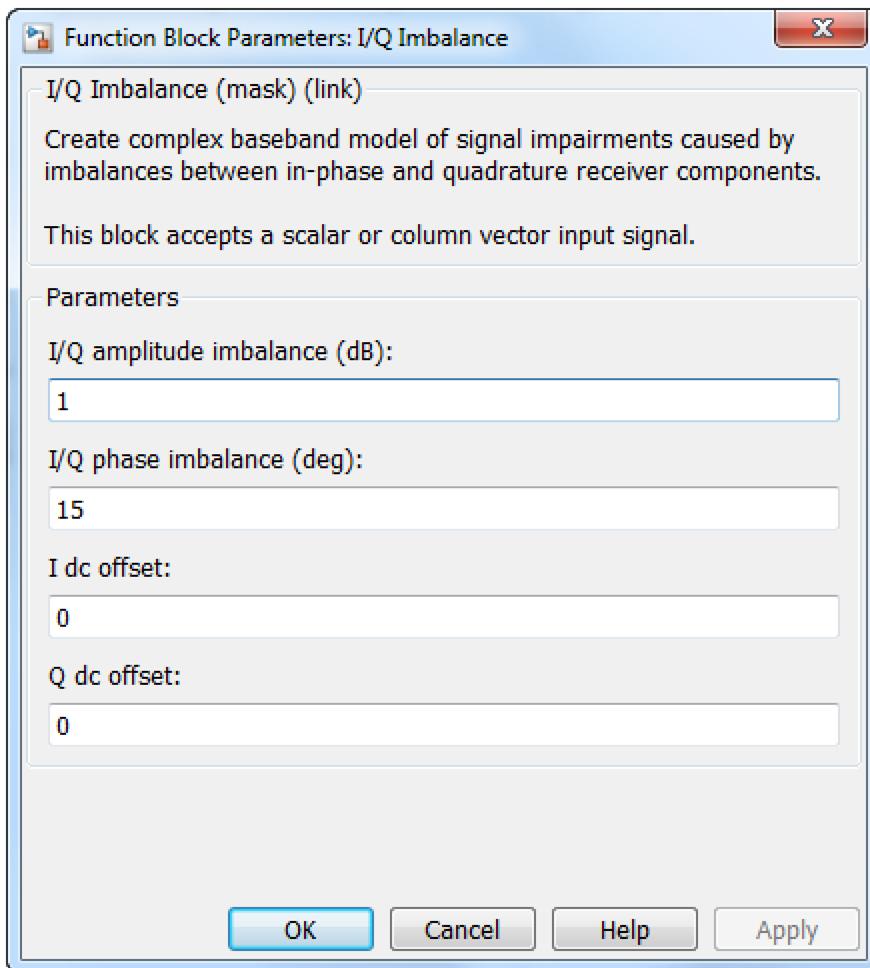
This example includes:

- A 16-QAM modulated signal
- An I/Q imbalance
- A constellation diagram block
- EVM Measurement and MER Measurement blocks

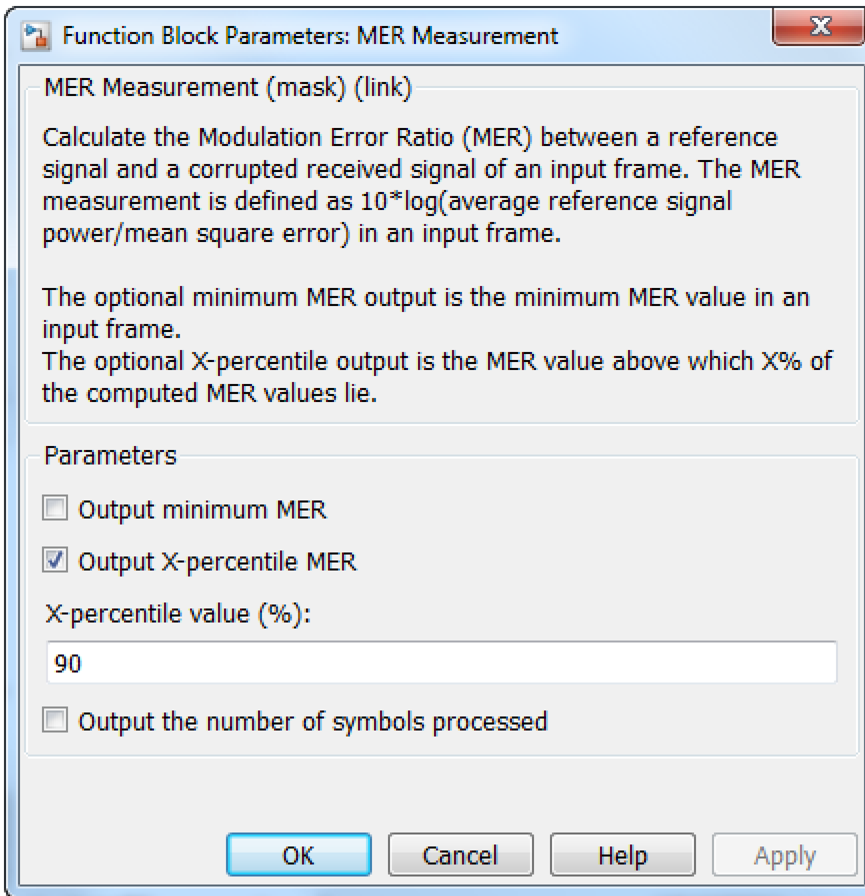
The model applies an I/Q imbalance to a QAM-modulated signal at which point MER and EVM measurements are made. The constellation diagram provides a visual representation of the effects the imbalance has on the modulation performance indicators.



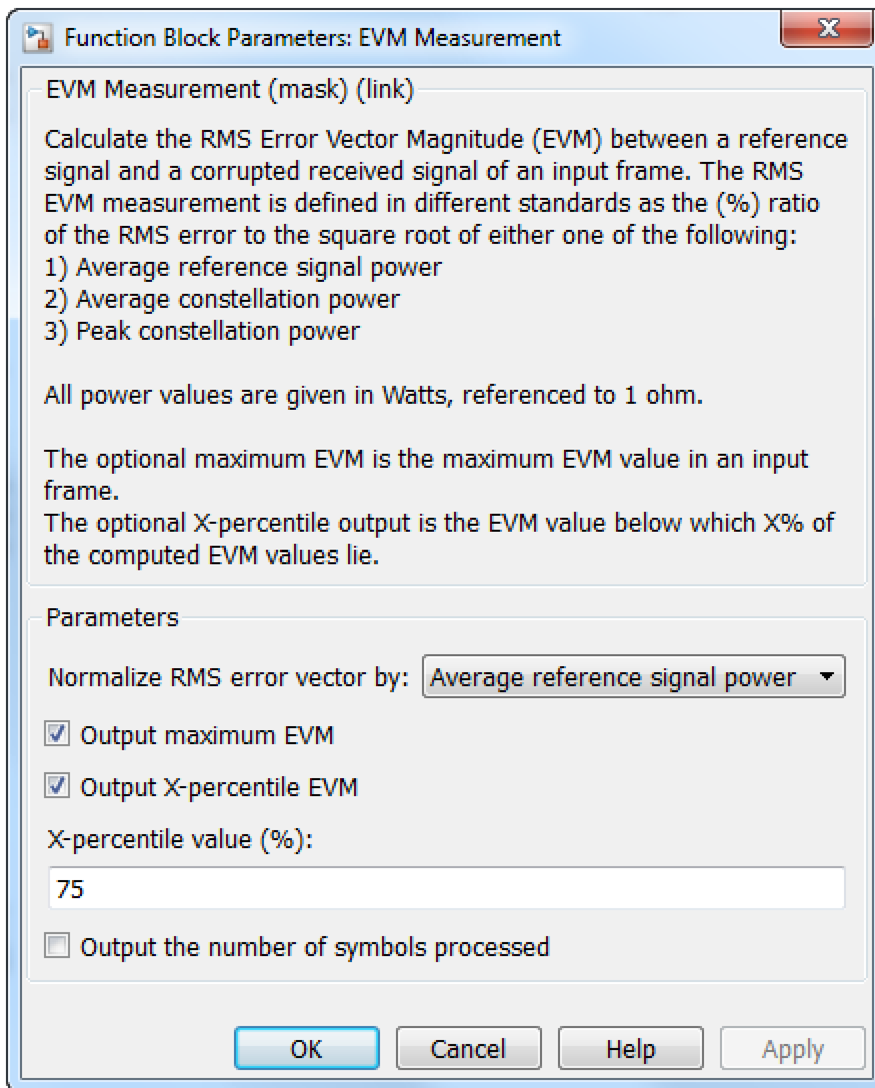
The I/Q Imbalance block is set to that the **I/Q amplitude imbalance (dB)** is set to 1 and the **I/Q phase imbalance (deg)** is set to 15.



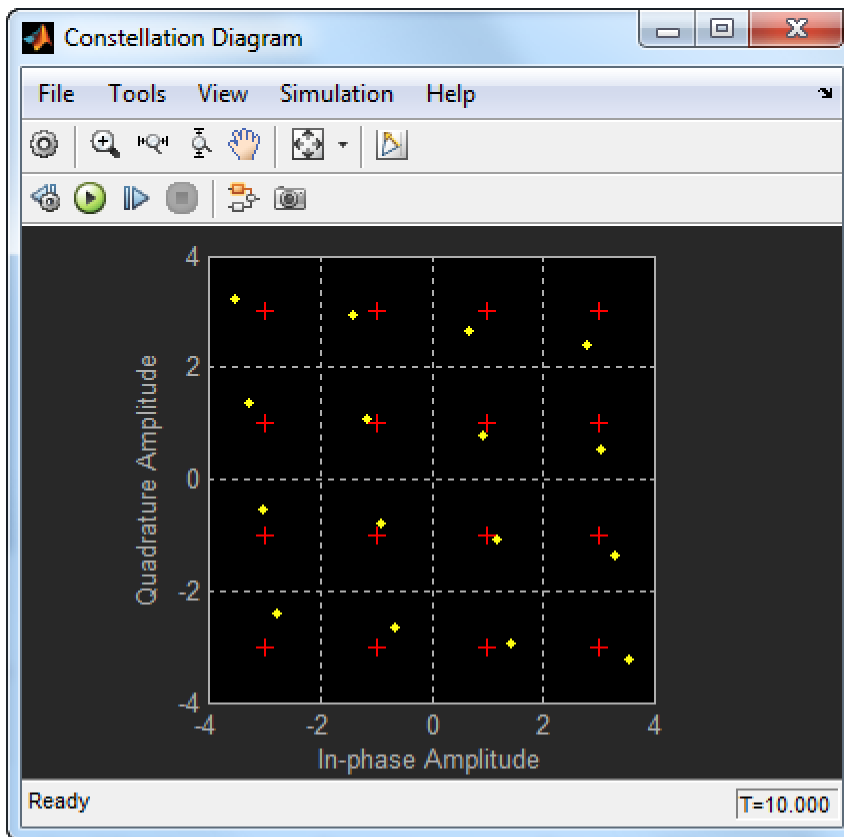
The MER Measurement block is set so that it outputs the X-percentile MER value which is set to 90%.



The EVM Measurement block is set to output the maximum and 75th percentile EVM values.

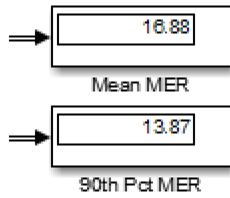


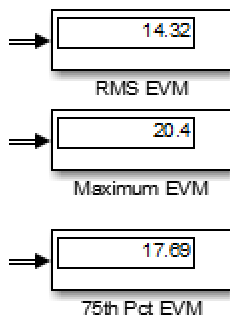
Run the model.



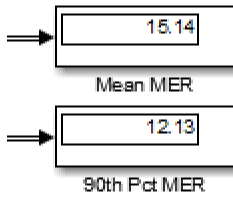
You can see that I/Q amplitude and phase imbalance has shifted the constellation diagram so that each symbol is not exactly equal to its reference symbol (shown in red). Change the I/Q Imbalance block to see the effects of differing imbalances on the constellation diagram.

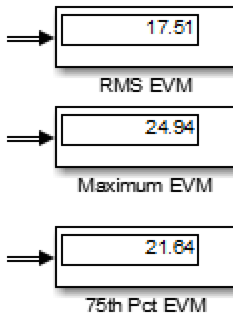
Observe the EVM and MER values. For the default configuration of the model, the mean MER is approximately 16.9 dB and the 90th percentile MER is 13.9 dB. The RMS EVM is, approximately, 14.3%, the maximum EVM is 20.4%, and the 75th percentile EVM is 17.7%.





Change the **I/Q amplitude imbalance (dB)** value in the I/Q Imbalance block to 2 dB. You can see that the all the MER and EVM metrics degrade.


15.14
Mean MER
12.13
90th Pct MER


17.51
RMS EVM
24.94
Maximum EVM
21.64
75th Pct EVM

ACPR and CCDF Measurements with MATLAB System Objects

In this section...
“ACPR Measurements” on page 3-13
“CCDF Measurements” on page 3-15

ACPR Measurements

This example shows how to measure the adjacent channel power ratio (ACPR) from a baseband, 50 kbps QPSK signal. ACPR is the ratio of signal power measured in an adjacent frequency band to the power from the same signal measured in its main band. The number of samples per symbol is set to four.

Set the samples per symbol (`sps`) and channel bandwidth (`bw`) parameters.

```
sps = 4;
bw = 50e3;
```

Generate 10,000 4-ary symbols for QSPK modulation.

```
data = randi([0 3],10000,1);
```

Construct a QPSK modulator and then modulate the input data.

```
qpskMod = comm.QPSKModulator;
x = qpskMod(data);
```

Apply rectangular pulse shaping to the modulated signal. This type of pulse shaping is typically not done in practical system but is used here for illustrative purposes.

```
y = rectpulse(x,sps);
```

Construct an ACPR System object. The sample rate is the bandwidth multiplied by the number of samples per symbol. The main channel is assumed to be at 0 while the adjacent channel offset is set to 50 kHz (identical to the bandwidth of the main channel). Likewise, the measurement bandwidth of the adjacent channel is set to be the same as the main channel. Lately, enable the main and adjacent channel power output ports.

```
acpr = comm.ACPR('SampleRate',bw*sps,...
    'MainChannelFrequency',0,...
    'MainMeasurementBandwidth',bw,...
    'AdjacentChannelOffset',50e3,...
    'AdjacentMeasurementBandwidth',bw,...
    'MainChannelPowerOutputPort', true,...
    'AdjacentChannelPowerOutputPort',true);
```

Measure the ACPR, the main channel power, and the adjacent channel power of signal *y*.

```
[ACPRout,mainPower,adjPower] = acpr(y)
```

```
ACPRout = -9.3071

mainPower = 28.9389

adjPower = 19.6318
```

Change the frequency offset to 75 kHz and determine the ACPR. Since the AdjacentChannelOffset property is nontunable, you must first release `acpr`. Observe that the ACPR improves when the channel offset is increased.

```
release(acpr)
acpr.AdjacentChannelOffset = 75e3;
ACPRout = acpr(y)
```

```
ACPRout = -13.1702
```

Release `acpr` and specify a 50 kHz adjacent channel offset.

```
release(acpr)
acpr.AdjacentChannelOffset = 50e3;
```

Create a raised cosine filter and filter the modulated signal.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol', sps);
z = txfilter(x);
```

Measure the ACPR for the filtered signal, z . You can see that the ACPR improves from -9.5 dB to -17.7 dB when raised cosine pulses are used.

```
ACPRout = acpr(z)
```

```
ACPRout = -17.2245
```

Plot the adjacent channel power ratios for a range of adjacent channel offsets. Set the channel offsets to range from 30 kHz to 70 kHz in 10 kHz steps. Recall that you must first release `hACPR` to change the offset.

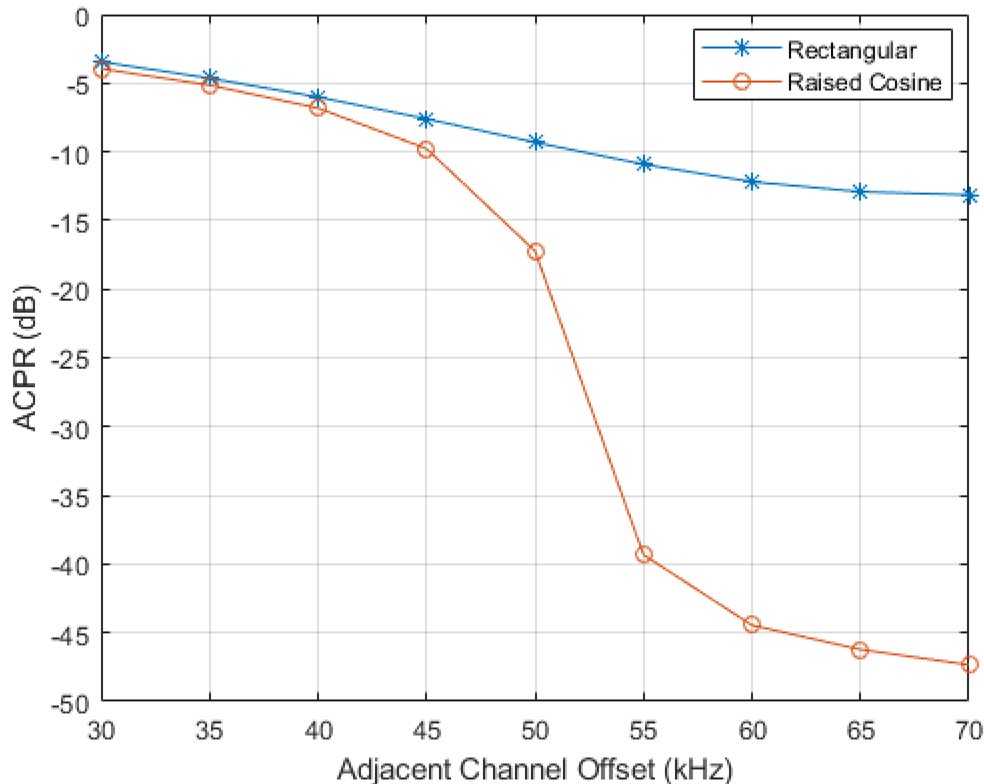
```
freqOffset = 1e3*(30:5:70);
release(acpr)
acpr.AdjacentChannelOffset = freqOffset;
```

Determine the ACPR values for the signals with rectangular and raised cosine pulse shapes.

```
ACPR1 = acpr(y);
ACPR2 = acpr(z);
```

Plot the adjacent channel power ratios.

```
plot(freqOffset/1000,ACPR1,'*-',freqOffset/1000, ACPR2,'o-')
xlabel('Adjacent Channel Offset (kHz)')
ylabel('ACPR (dB)')
legend('Rectangular','Raised Cosine','location','best')
grid
```



CCDF Measurements

This example shows how to use the Complementary Cumulative Distribution Function (CCDF) System object to measure the probability of a signal's instantaneous power being greater than a specified level over its average power. Construct the `comm.CCDF` object, enable the PAPR output port, and set the maximum signal power limit to 50 dBm.

```
ccdf = comm.CCDF('PAPROutputPort',true,'MaximumPowerLimit', 50);
```

Create an OFDM modulator having an FFT length of 256 and a cyclic prefix length of 32.

```
ofdmMod = comm.OFDMModulator('FFTLength',256,'CyclicPrefixLength',32);
```

Determine the input and output sizes of the OFDM modulator object using the `info` function of the `comm.OFDMModulator` object.

```
ofdmDims = info(ofdmMod)
```

```
ofdmDims = struct with fields:
    DataInputSize: [245 1]
       OutputSize: [288 1]
```

```
ofdmInputSize = ofdmDims.DataInputSize;
ofdmOutputSize = ofdmDims.OutputSize;
```

Set the number of OFDM frames.

```
numFrames = 20;
```

Allocate memory for the signal arrays.

```
qamSig = repmat(zeros(ofdmInputSize),numFrames,1);
ofdmSig = repmat(zeros(ofdmOutputSize),numFrames,1);
```

Generate the 64-QAM and OFDM signals for evaluation.

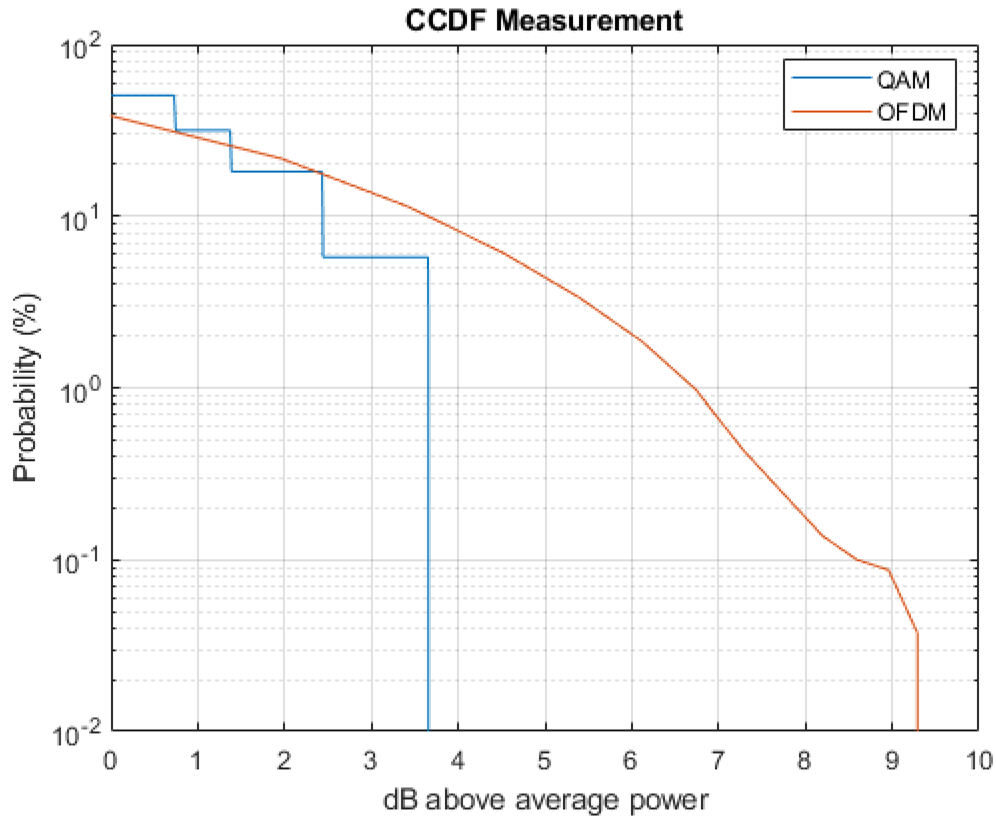
```
for k = 1:numFrames
    % Generate random data symbols
    data = randi([0 63],ofdmInputSize);
    % Apply 64-QAM modulation
    tmpQAM = qammod(data,64);
    % Apply OFDM modulation to the QAM-modulated signal
    tmpOFDM = ofdmMod(tmpQAM);
    % Save the signal data
    qamSig((1:ofdmInputSize)+(k-1)*ofdmInputSize(1)) = tmpQAM;
    ofdmSig((1:ofdmOutputSize)+(k-1)*ofdmOutputSize(1)) = tmpOFDM;
end
```

Determine the average signal power, the peak signal power, and the PAPR ratios for the two signals. The two signals being evaluated must be the same length so the first 4000 symbols are evaluated.

```
[Fy,Fx,PAPR] = ccdf([qamSig(1:4000),ofdmSig(1:4000)]);
```

Plot the CCDF data. Observe that the likelihood of the power of the OFDM modulated signal being more than 3 dB above its average power level is much higher than for the QAM modulated signal.

```
plot(ccdf)
legend('QAM','OFDM','location','best')
```



Compare the PAPR values for the QAM modulated and OFDM modulated signals.

```
fprintf('\nPAPR for 64-QAM = %5.2f dB\nPAPR for OFDM = %5.2f dB\n',...
    PAPR(1), PAPR(2))
```

```
PAPR for 64-QAM =  3.65 dB
PAPR for OFDM =  9.44 dB
```

You can see that by applying OFDM modulation to a 64-QAM modulated signal, the PAPR increases by 5.8 dB. This means that if 30 dBm transmit power is needed to close a 64-QAM link, the power amplifier needs to have a maximum power of 33.7 dBm to ensure linear operation. If the same signal were then OFDM modulated, a 39.5 dBm power amplifier is required.