Communications Toolbox™
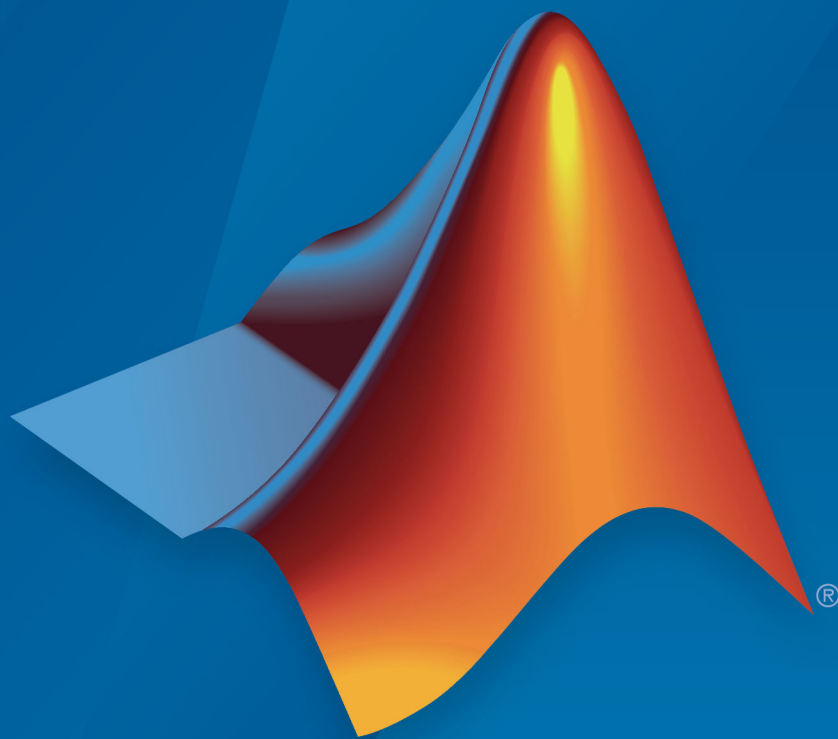
Getting Started Guide

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.
1 Apple Hill Drive
Natick, MA 01760-2098

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)

Contents

Visualization and Measurements

3

Introduction

- "Communications Toolbox Product Description" on page 1-2
- "Configure the Simulink Environment for Communications Models" on page 1-4
- "Access the Block Libraries" on page 1-7
- "Communications Toolbox Supported Hardware" on page 1-8

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

About the 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 "Create a Model" (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.
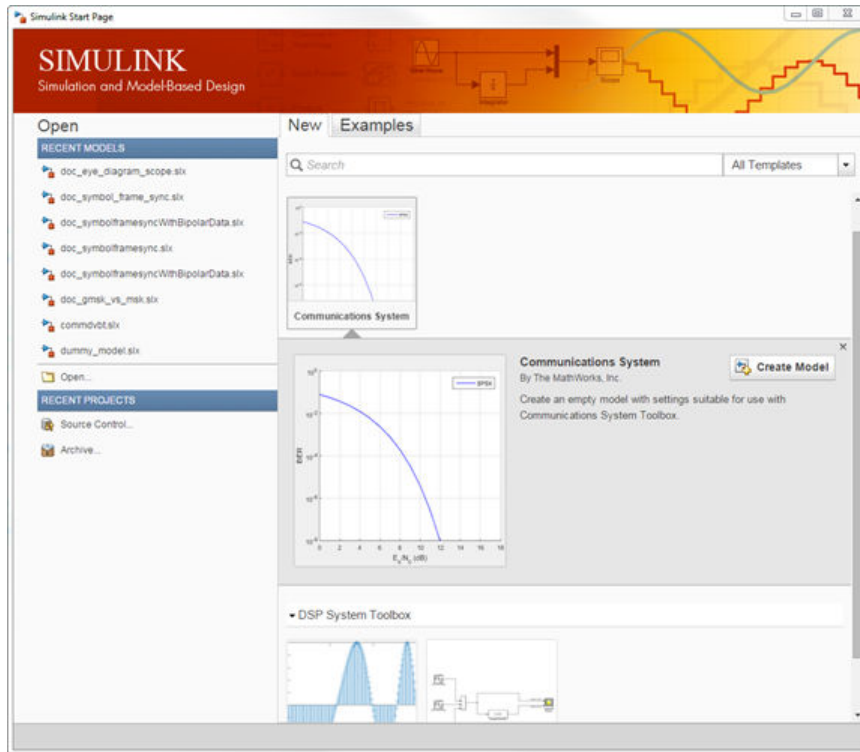
The new model using the template settings and contents appears in the Simulink Editor. The model is only in memory until you save it.

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'

1-5

Configuration Parameter	Setting
`'MaxStep'`	`'auto'`
`'StartTime'`	`'0.0'`
`'StopTime'`	`'inf'`
`'FixedStep'`	`'auto'`
`'SaveTime'`	`'off'`
`'SaveOutput'`	`'off'`
`'AlgebraicLoopMsg'`	`'error'`
`'RTWInlineParameters'`	`'on'`
`'BooleanDataType'`	`'off'`
`'UnnecessaryDatatypeConvMsg'`	`'none'`
`'LocalBlockOutputs'`	`'off'`

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.
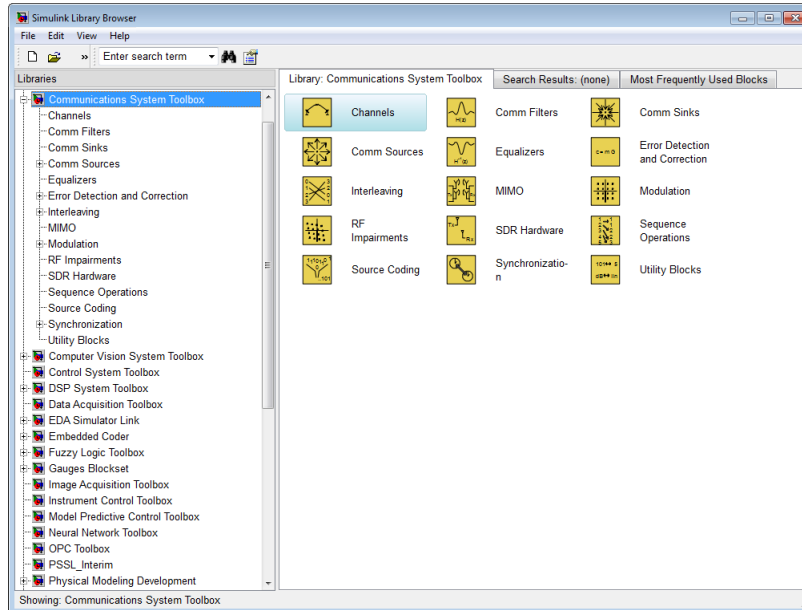
Simulink Library Browser

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.

Communications Toolbox Supported Hardware

![Get Support Package Now]

As of this release, Communications Toolbox supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
ADALM-Pluto Radio	Analog Devices®	R2017a	Current
RTL-SDR Radio	NooElec™	R2013b	Current
USRP® Embedded Series Radio	Ettus Research®	R2016b	Current
USRP® Radio	Ettus Research	R2011b	Current
Xilinx Zynq-Based Radio	Xilinx®	R2014b	Current

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

2

System Simulation

256-QAM with Simulink Blocks

In this section...

Section Overview

This section describes a Simulink model of a communications system and is intended to familiarize you with the basic functionality of Simulink models.. The model displays a scatter plot of a signal with added noise and performs error rate computations.

Opening the Model

To open the model, first start MATLAB. In the MATLAB Command Window, enter `doc_commphasenoise` at the prompt. This opens the model shown here in a new window.
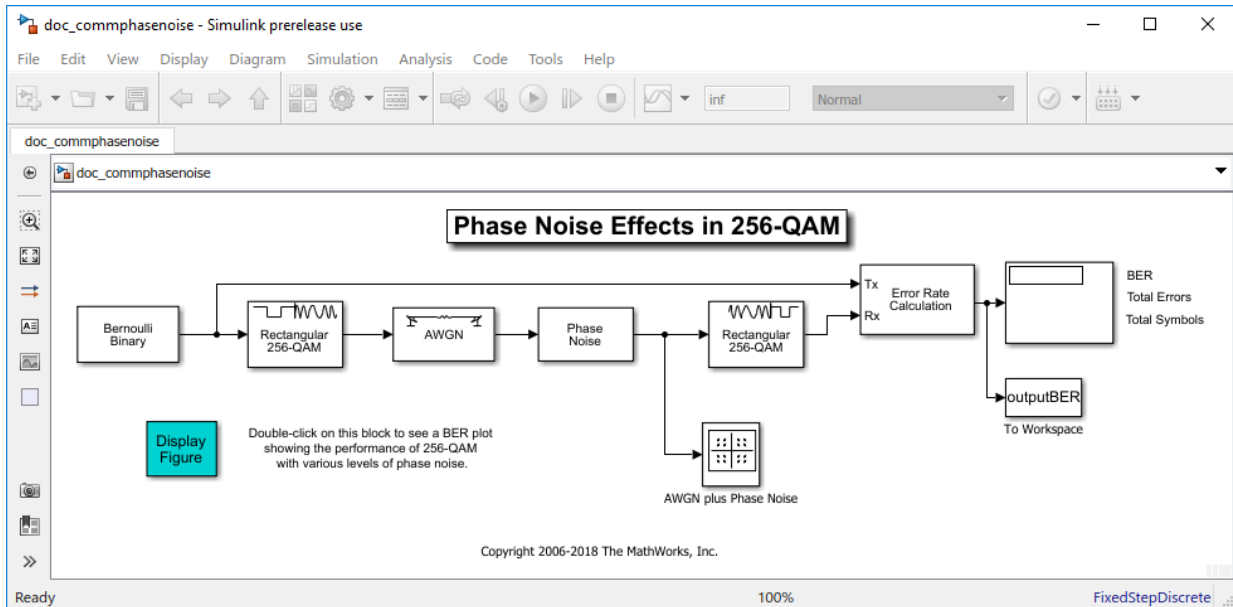
Overview of the Model

The `doc_commphasenoise` model, simulates the effect of phase noise on quadrature amplitude modulation (QAM) of a signal. The Simulink model is a graphical representation of a mathematical model of a communication system that generates a random signal, modulates it using QAM, and adds noise to simulate a channel. The model also contains components for displaying the symbol error rate and a scatter plot of the modulated signal.

The blocks and lines in the Simulink model describe mathematical relationships among signals and states.

- The Bernoulli Binary Generator block, labeled *Bernoulli Binary*, generates a signal consisting of a sequence of 8-bit binary values between zero and 255

- The Rectangular QAM Modulator Baseband block, to the right of the Bernoulli Binary Generator 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, to the right of the Phase Noise block, demodulates the signal.

Additional blocks in the model help you interpret the simulation.

- The Constellation Diagram block, labeled AWGN plus Phase Noise, displays a scatter plot of the signal with added noise.
- The Error Rate Calculation block counts symbols that differ between the received signal and the transmitted signal.
- The To Workspace block, labeled *outputBER* outputs the results to the workspace to use when plotting results.
- The Display Figure block, at the far right of the model window, displays the bit error rate (BER), the total number of errors, and the total number of symbols processed during the simulation.

All these blocks are included in Communications Toolbox. You can find more detailed information about these blocks by right-clicking the block and selecting **Help** from the context menu.

Quadrature Amplitude Modulation

This model simulates quadrature amplitude modulation (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, called the *constellation* of the signal. The constellation for baseband 256-ary QAM with is shown in the following figure.
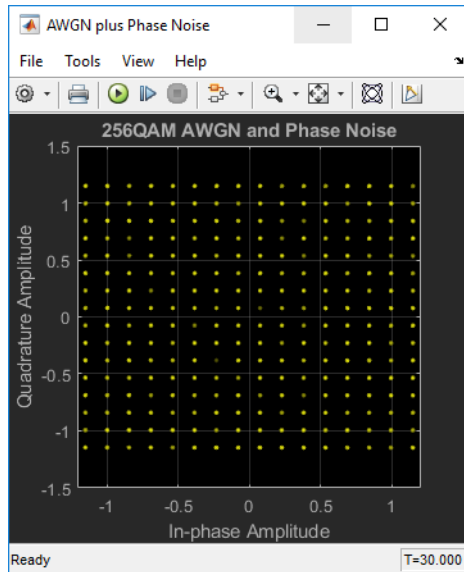
Constellation for 256-ary QAM

Run a Simulation

To run a simulation, click on the **Run** button at the top of the model window. The run duration is set to `inf`, which is specified by **Stop time** in the **Configuration Parameters** dialog box. The simulation is configured in the Error Rate Calculation block to run until 100 bit errors occur. You can stop the simulation at any time by selecting **Stop** from the **Simulation** menu at the top of the model window or by clicking the **Stop** button on the toolstrip.

When you run the model, a new window appears, displaying a scatter plot of the modulated signal with added noise. For a high EbN0 setting, the phase noise impairment is apparent as shown in this figure.
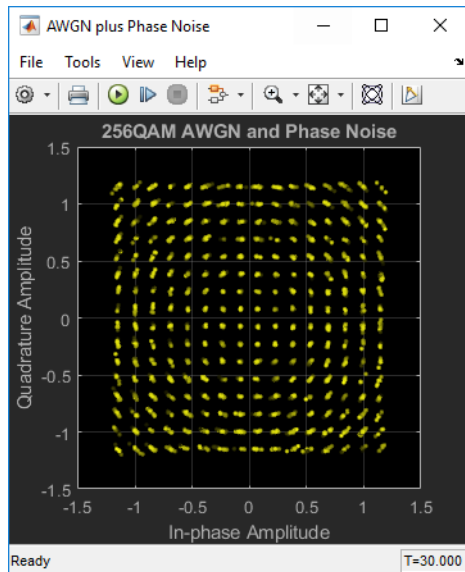
Scatter Plot of Signal Plus Noise

The points in the scatter plot do not lie exactly on the constellation shown in the figure because of the added noise. The radial displacement of points is due to the addition of phase noise, which alters the angle of the complex modulated signal.

Display the Error Rate

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

Note The image below is a representative example and may not exactly match results you see when running in Simulink.
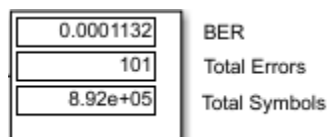
The block displays the output as follows:

- The first entry is the bit error rate (BER).
- The second entry is the total number of errors.
- The third entry is the total number of comparisons made. The notation `8.92e+005` is shorthand for 8.92×10^5.

Set Block Parameters

You can control the way a Simulink block functions by setting its parameters. To view or change a block's parameters, double-click the block. This opens a dialog box, sometimes called the block's *mask*. For example, the dialog box for the Phase Noise block is shown in the following figure.
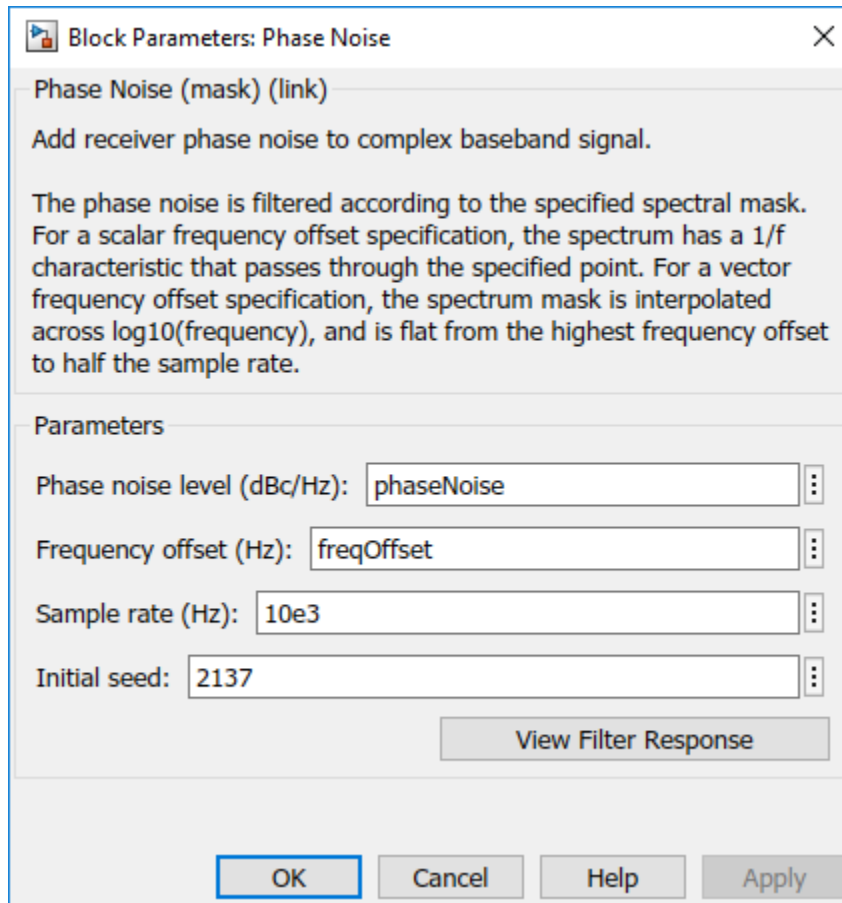
Dialog for the Phase Noise Block

To change the amount of phase noise, click in the **Phase noise level (dBc/Hz)** field and enter a new value. Then click **OK**.

Alternatively, you can enter a variable name, such as `phasenoise`, in the field. You can then set a value for that variable in the MATLAB Command Window, for example by entering `phasenoise = -60`. Setting parameters in the Command Window is convenient if you need to run multiple simulations with different parameter values.

You can also change the amount of noise in the AWGN Channel block. Double-click the block to open its dialog box, and change the value in the **Eb/No** parameter field. This changes the signal to noise ratio, in dB. Decreasing the value of **Eb/No** increases the noise level.

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 under **File**, open **Model Properties**, then select the **Callbacks** tab.

Model Properties: doc_commphasenoise   ✕

| Main | Callbacks | History | Description | Data |

Model callbacks

PreLoadFcn*
PostLoadFcn
InitFcn
StartFcn
PauseFcn
ContinueFcn
StopFcn
PreSaveFcn
PostSaveFcn
CloseFcn*

Model pre-load function:

```
Nbps = 8; % for 256 QAM
EbN0 = 35;
phaseNoise = -76;
freqOffset = 1000;
tgtMaxErr = 100;
```

OK   Cancel   Help   Apply
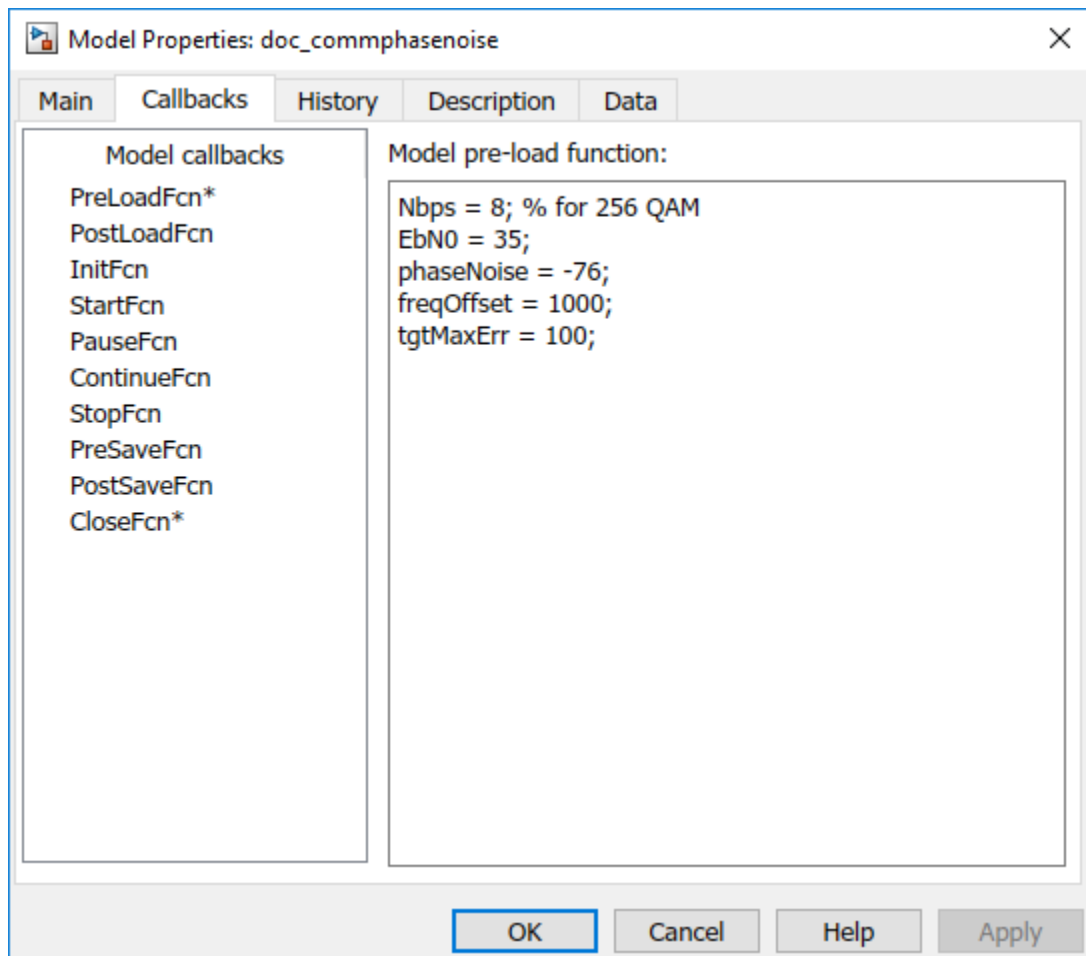
For more information on **Model Properties** and callback functions, see "Model Callbacks" (Simulink).

You can experiment with the model by changing these or other parameters and then running a simulation. For example,

- Change **Phase noise level (dBc/Hz)** to `-150` in the dialog box for the Phase Noise block.
- Change **Eb/No** to `100` in the dialog for the AWGN Channel block.

The default setting for simulation duration is `inf` with the Error Rate Calculation is set to stop when 100 errors have occurred. By reducing the phase noise and increasing the **Eb/No**, nearly all noise is removed from the model. This will mean the simulation may need to run for a long time before 100 errors occur. So, before running the model with nearly no noise, you may want to change the run duration from *inf* to a smaller value, such as `10`. When you now run a simulation, the scatter plot appears as in the figure "Constellation for 256-ary QAM" on page 2-5.

Display a Phase Noise Plot

To display a figure that plots simulation results of BER vs EbNo curves for a range of phase noise settings, double-click the Display Figureblock at the bottom left of the model window.

Performance of 256-QAM for Frequency Offset of 1000 Hz
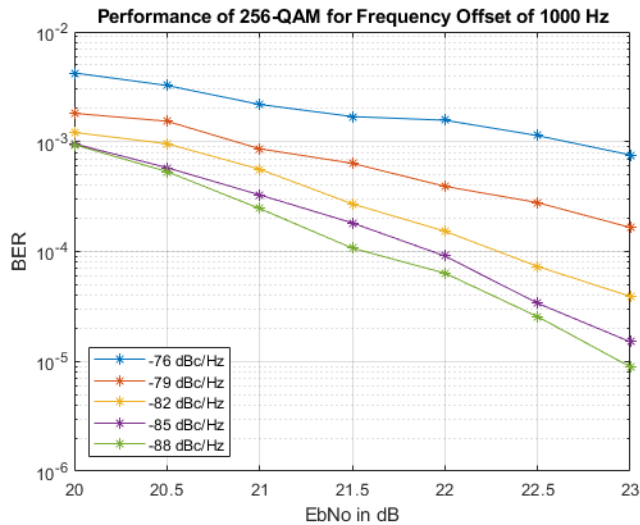
BER Plot at Different Noise Levels

Each curve is a plot of bit error rate as a function of signal to noise ratio for a fixed amount of phase noise.

You can create plots like this by running multiple simulations with different values for the **Phase noise level (dBc/Hz)** and **Eb/No** parameters. To generate the plot shown in this example, run this code. Since random numbers are generated when running the simulation, your results may not exactly match the results shown here.

```
%% Plot 256QAM BER vs EbN0dB
% Helper script to generate EbN0 vs. BER curves plotting phase
% noise performance of 256 QAM for a frequency offset of 1000 Hz.
% This script generates the 'Display Figure' for the doc_commphasenoise model.
clear;
open_system('doc_commphasenoise');
%%
% Set the frequency offset to 1000 Hz in the Phase Noise block and
% number of bits per symbol to 8 thoughout the model.
freqOffset = 1000;
Nbps = 8;
% Define EbNo and phase noise ranges for simulation and plotting.
EbNoRng=(20:0.5:23);
PhNoiseRng=-88:3:-76;
% Specify the target number of bit errors.
tgtMaxErr = 200;
%%
% Sweep phase noise and EbNo ranges to generate results, then plot.
ber = zeros(length(PhNoiseRng),length(EbNoRng));
for ip=1:length(PhNoiseRng)
```

```
        phaseNoise = PhNoiseRng(ip);
        for ies=1:length(EbNoRng)
            EbN0 = EbNoRng(ies);
            sim('doc_commphasenoise')
            ber(ip,ies) = outputBER(1);
        end
end

semilogy(EbNoRng,ber,'*-');
axis([EbNoRng(1) EbNoRng(end) 1e-6 1e-2])
xlabel('EbNo in dB')
ylabel('BER')
legend([num2str(PhNoiseRng(:)) repmat(' dBc/Hz',[length(PhNoiseRng),1])],'Location','southwest')
title(['Performance of 256-QAM for Frequency Offset of ' num2str(freqOffset) ' Hz'])
hold off
grid on
```

16-QAM with MATLAB Functions

In this section...
"Introduction" on page 2-13
"Modulate a Random Signal" on page 2-13
"Plot Signal Constellations" on page 2-19
"Pulse Shaping Using a Raised Cosine Filter" on page 2-22
"Error Correction using a Convolutional Code" on page 2-28

Introduction

This section builds an example step-by-step to give you a first look at the Communications Toolbox software. This section also shows how Communications Toolbox functionalities build upon the computational and visualization tools in the underlying MATLAB environment.

Modulate a Random Signal

This example shows how to process a binary data stream using a communication system that consists of a baseband modulator, channel, and demodulator. The system's bit error rate (BER) is computed and the transmitted and received signals are displayed in a constellation diagram.

The following table summarizes the basic operations used, along with relevant Communications Toolbox and MATLAB functions. The example uses baseband 16-QAM (quadrature amplitude modulation) as the modulation scheme and AWGN (additive white Gaussian noise) as the channel model.

Task	Function
Generate a Random Binary Data Stream	`randi`
Convert the Binary Signal to an Integer-Valued Signal	`bi2de`
Modulate using 16-QAM	`qammod`
Add White Gaussian Noise	`awgn`
Create a Constellation Diagram	`scatterplot`

Task	Function
Demodulate using 16-QAM	`qamdemod`
Convert the Integer-Valued Signal to a Binary Signal	`de2bi`
Compute the System BER	`biterr`

Generate a Random Binary Data Stream

The conventional format for representing a signal in MATLAB is a vector or matrix. This example uses the `randi` function to create a column vector that contains 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.

The code below also creates a stem plot of a portion of the data stream, showing the binary values. Notice the use of the colon (`:`) operator in MATLAB to select a portion of the vector.

Define parameters.

```
M = 16;                   % Size of signal constellation
k = log2(M);              % Number of bits per symbol
n = 30000;                % Number of bits to process
numSamplesPerSymbol = 1;  % Oversampling factor
```

Create a binary data stream as a column vector.

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

Plot the first 40 bits in a stem plot.

```
stem(dataIn(1:40),'filled');
title('Random Bits');
xlabel('Bit Index');
ylabel('Binary Value');
```
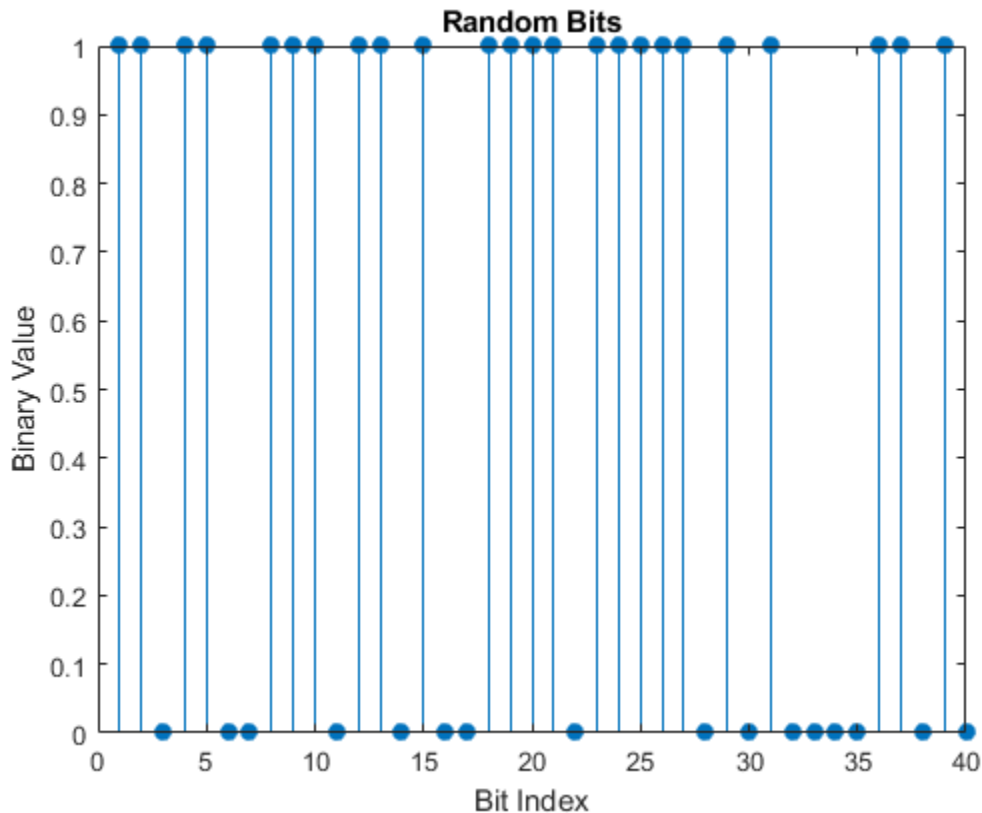
Convert the Binary Signal to an Integer-Valued Signal

The `qammod` function implements a rectangular, M-ary QAM modulator, M being 16 in this example. The default configuration is such that the object receives integers between 0 and 15 rather than 4-tuples of bits. In this example, we preprocess the binary data stream `dataIn` before using the `qammod` function. In particular, the `bi2de` function is used to convert each 4-tuple to a corresponding integer.

Perform a bit-to-symbol mapping.

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

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');
```
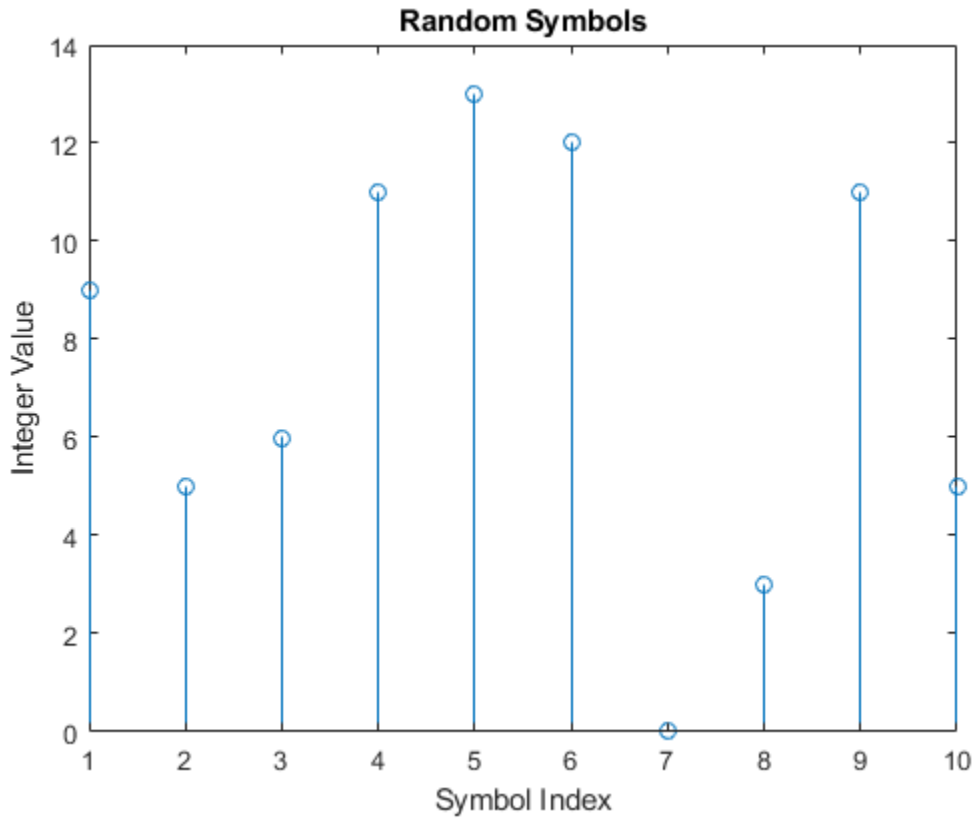

Modulate using 16-QAM

Having generated the `dataSymbolsIn` column vector, use the `qammod` function to apply 16-QAM modulation for both binary and Gray coded bit-to-symbol mappings. Recall that `M` is 16, the alphabet size.

Apply modulation.

```matlab
dataMod = qammod(dataSymbolsIn,M,'bin');          % Binary coding, phase offset = 0
dataModG = qammod(dataSymbolsIn,M); % Gray coding, phase offset = 0
```

The results are complex column vectors whose values are elements of the 16-QAM signal constellation. A later step in this example will plot the constellation diagram.

To learn more about modulation functions, see "Digital Modulation". Also, note that the `qammod` function does not apply pulse shaping. To extend this example to use pulse shaping, see "Pulse Shaping Using a Raised Cosine Filter" on page 2-22. For an example that uses Gray coding with PSK modulation, see Gray Coded 8-PSK.

Add White Gaussian Noise

The ratio of bit energy to noise power spectral density, E_b/N_0, is arbitrarily set to 10 dB. From that value, the signal-to-noise ratio (SNR) can be determined. Given the SNR, the modulated signal, `dataMod`, is passed through the channel by using the `awgn` function.

Note The `numSamplesPerSymbol` variable is not significant in this example but will make it easier to extend the example later to use pulse shaping.

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

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

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

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

Create a Constellation Diagram

The `scatterplot` function is used to display the in-phase and quadrature components of the modulated signal, `dataMod`, and its received, noisy version, `receivedSignal`. By looking at the resultant diagram, the effects of AWGN are readily observable.

Use the `scatterplot` function to show the constellation diagram.

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

Scatter plot
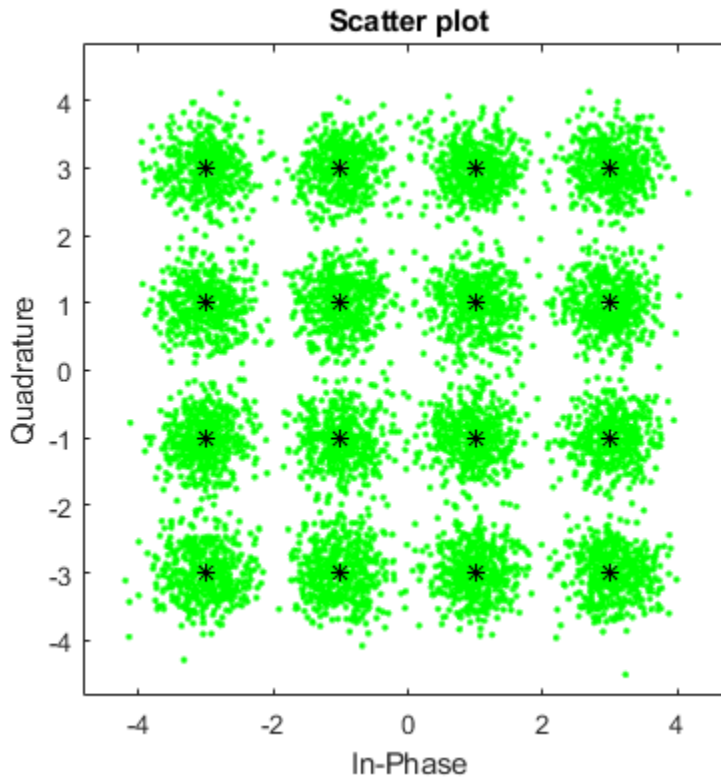
Demodulate 16-QAM

The qamdemod function is used to demodulate the received data and output integer-valued data symbols.

Demodulate the received signals using the qamdemod function.

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

Convert the Integer-Valued Signal to a Binary Signal

The de2bi function is used to convert the data symbols from the QAM demodulator, dataSymbolsOut, into a binary matrix, dataOutMatrix with dimensions of N_{sym}-by-

$N_{bits/sym}$, where N_{sym} is the total number of QAM symbols and $N_{bits/sym}$ is the number of bits per symbol, four in this case. The matrix is then converted into a column vector whose length is equal to the number of input bits, 30,000. The process is repeated for the Gray coded data symbols, `dataSymbolsOutG`.

Reverse the bit-to-symbol mapping performed earlier.

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

Compute the System BER

The function `biterr` is used to calculate the bit error statistics from the original binary data stream, `dataIn`, and the received data streams, `dataOut` and `dataOutG`.

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

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

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

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

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

Observe that Gray coding significantly reduces the bit error rate.

Plot Signal Constellations

The example in the previous section, "Modulate a Random Signal" on page 2-13, created a scatter plot from the modulated signal. Although the plot showed the points in the QAM constellation, the plot did not indicate which integers of the modulator are mapped to a given constellation point. This section illustrates two possible mappings: 1) binary coding, and 2) Gray coding. It was previously demonstrated that Gray coding provides superior bit error rate performance.

Binary Symbol Mapping for 16-QAM Constellation

Apply 16-QAM modulation to all possible input values using the default symbol mapping, binary.

```
M = 16;                     % Modulation order
x = (0:15)';                % Integer input
y1 = qammod(x,16,'bin');    % 16-QAM output
```

Use the `scatterplot` function to plot the constellation diagram and annotate it with binary representations of the constellation points.

```
scatterplot(y1)
text(real(y1)+0.1, imag(y1), dec2bin(x))
title('16-QAM, Binary Symbol Mapping')
axis([-4 4 -4 4])
```
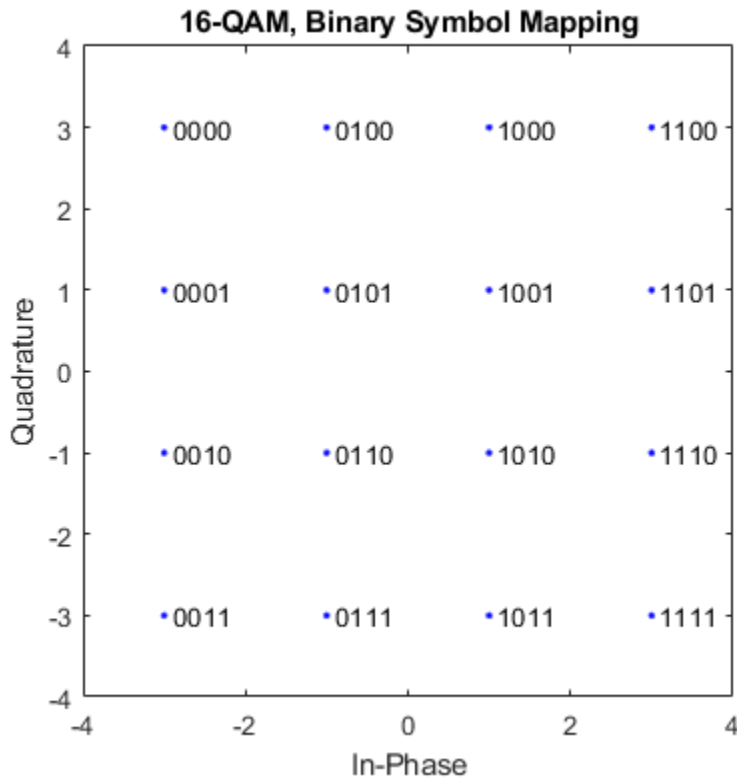
Gray-coded Symbol Mapping for 16-QAM Constellation

Apply 16-QAM modulation to all possible input values using Gray-coded symbol mapping.

```
y2 = qammod(x,16,'gray');  % 16-QAM output, Gray-coded
```

Use the `scatterplot` function to plot the constellation diagram and annotate it with binary representations of the constellation points.

```
scatterplot(y2)
text(real(y2)+0.1, imag(y2), dec2bin(x))
title('16-QAM, Gray-coded Symbol Mapping')
axis([-4 4 -4 4])
```
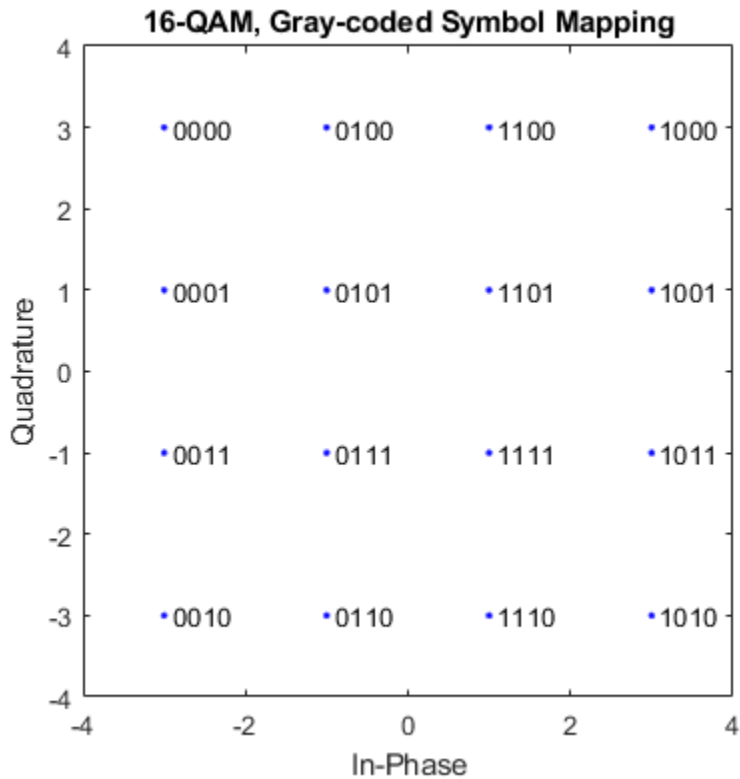
Examine the Plots

In the binary mapping plot, notice that symbols 1 (0 0 0 1) and 2 (0 0 1 0) correspond to adjacent constellation points on the left side of the diagram. The binary representations of these integers differ by two bits unlike the Gray-coded signal constellation in which each point differs by only one bit from its direct neighbors, which leads to better BER performance.

Pulse Shaping Using a Raised Cosine Filter

The "Modulate a Random Signal" on page 2-13 example was modified to employ a pair of square-root raised cosine (RRC) filters to perform pulse shaping and matched filtering.

The filters are created by the `rcosdesign` function. In "Error Correction using a Convolutional Code" on page 2-28, this example is extended by introducing forward error correction (FEC) to improve BER performance.

To create a BER simulation, a modulator, demodulator, communication channel, and error counter functions must be used and certain key parameters must be specified. In this case, 16-QAM modulation is used in an AWGN channel.

Establish Simulation Framework

Set the simulation parameters.

```
M = 16;                    % Size of signal constellation
k = log2(M);               % Number of bits per symbol
numBits = 3e5;             % Number of bits to process
numSamplesPerSymbol = 4;   % Oversampling factor
```

Create Raised Cosine Filter

Set the square-root, raised cosine filter parameters.

```
span = 10;        % Filter span in symbols
rolloff = 0.25;   % Roloff factor of filter
```

Create a square-root, raised cosine filter using the `rcosdesign` function.

```
rrcFilter = rcosdesign(rolloff, span, numSamplesPerSymbol);
```

Display the RRC filter impulse response using the `fvtool` function.

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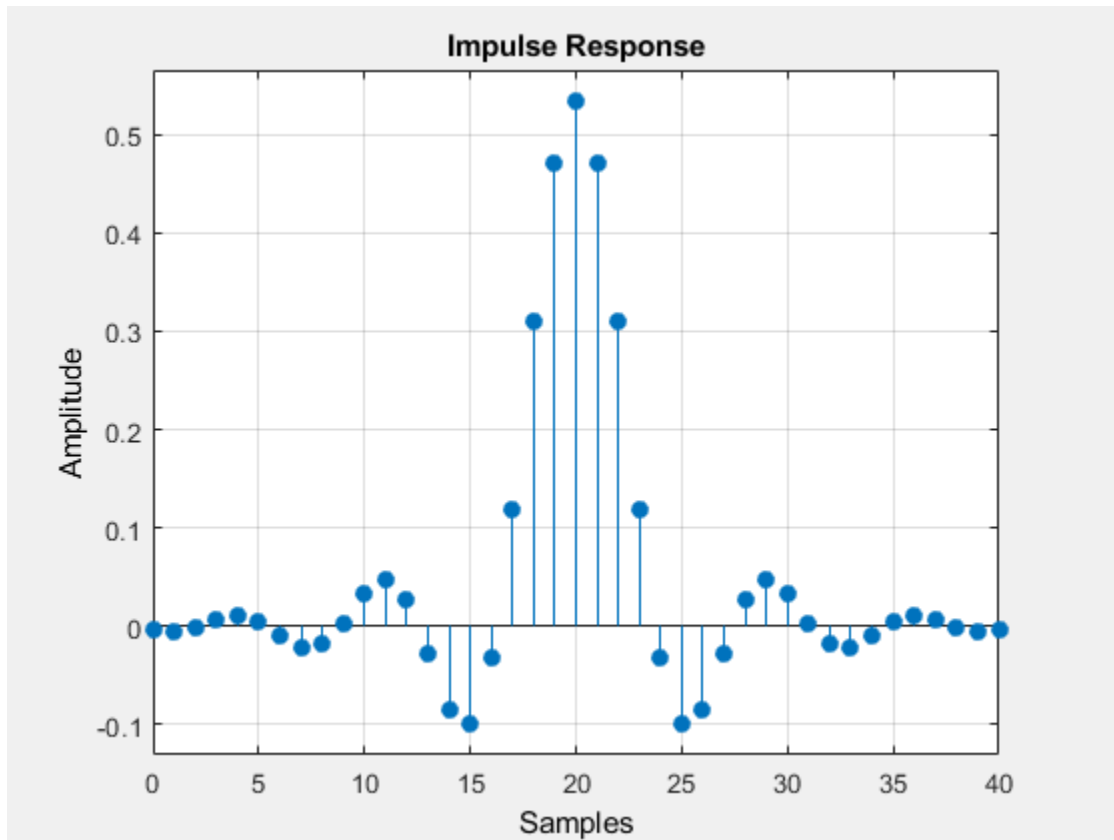
BER Simulation

Use the `randi` function to generate random binary data. The `rng` function should be set to its default state 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, which is then converted into integer symbols.

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

Apply 16-QAM modulation using `qammod`.

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

Using the `upfirdn` function, upsample and apply the square-root, raised cosine filter.

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

The `upfirdn` function upsamples the modulated signal, `dataMod`, by a factor of `numSamplesPerSymbol`, pads the upsampled signal with zeros at the end to flush the filter and then applies the filter.

Set the E_b/N_0 to 10 dB and convert the SNR given the number of bits per symbol, k, and the number of samples per symbol.

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

Pass the filtered signal through an AWGN channel.

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

Filter the received signal using the square-root, raised cosine filter and remove a portion of the signal to account for the filter delay in order to make a meaningful BER comparison.

```
rxFiltSignal = upfirdn(rxSignal,rrcFilter,1,numSamplesPerSymbol);   % Downsample and fi
rxFiltSignal = rxFiltSignal(span+1:end-span);                       % Account for delay
```

These functions apply the same square-root raised cosine filter that the transmitter used earlier, and then downsample the result by a factor of `nSamplesPerSymbol`. The last command removes the first `Span` symbols and the last `Span` symbols in the decimated signal because they represent the cumulative delay of the two filtering operations. Now `rxFiltSignal`, which is the input to the demodulator, and `dataSymbolsOut`, which is the output from the modulator, have the same vector size. In the part of the example that computes the bit error rate, it is required to compare vectors that have the same size.

Apply 16-QAM demodulation to the received, filtered signal.

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

Using the `de2bi` function, convert the incoming integer symbols into binary data.

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

Apply the `biterr` function to determine the number of errors and the associated BER.

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

The bit error rate = 1.83e-03, based on 550 errors

Visualization of Filter Effects

Create an eye diagram for a portion of the filtered signal.

```
eyediagram(txSignal(1:2000),numSamplesPerSymbol*2);
```
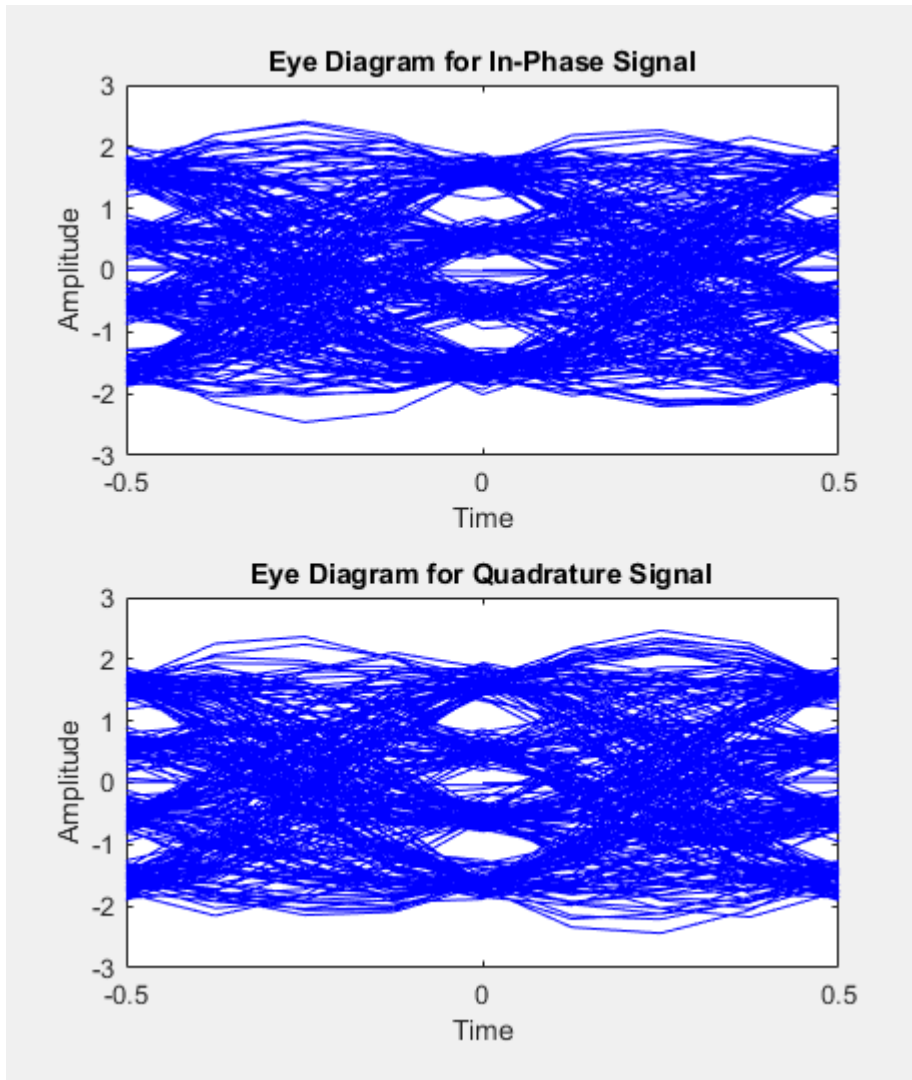
The `eyediagram` function creates an eye diagram for part of the filtered noiseless signal. This diagram illustrates the effect of the pulse shaping. Note that the signal shows significant intersymbol interference (ISI) because the filter is a square-root raised cosine filter, not a full raised cosine filter.

Created a scatter plot of the received signal before and after filtering.

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

Error Correction using a Convolutional Code

Building upon the "Pulse Shaping Using a Raised Cosine Filter" on page 2-22 example, this example shows how bit error rate performance improves with the addition of forward error correction, FEC, coding.

Establish Simulation Framework

To create the simulation, a modulator, demodulator, raised cosine filter pair, communication channel, and error counter functions are used and certain key parameters are specified. In this case, a 16-QAM modulation scheme with raised cosine filtering is used in an AWGN channel. With the exception of the number of bits, the specified parameters are identical to those used in the previous example.

Set the simulation variables. The number of bits is increased from the previous example so that the bit error rate may be estimated more accurately.

```
M = 16;                                    % Size of signal constellation
k = log2(M);                               % Number of bits per symbol
numBits = 100000;                          % Number of bits to process
numSamplesPerSymbol = 4;                   % Oversampling factor
```

Generate Random Data

Use the `randi` function to generate random, binary data once the `rng` function has been called. When set to its default value, the `rng` function ensures that the results from this example are repeatable.

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

Convolutional Encoding

The performance of the "Pulse Shaping Using a Raised Cosine Filter" on page 2-22 example can be significantly improved upon by employing forward error correction. In this example, convolutional coding is applied to the transmitted bit stream in order to

correct errors arising from the noisy channel. Because it is often implemented in real systems, the Viterbi algorithm is used to decode the received data. A hard decision algorithm is used, which means that the decoder interprets each input as either a "0" or a "1".

Define a convolutional coding trellis for a rate 2/3 code. The `poly2trellis` function defines the trellis that represents the convolutional code that `convenc` uses for encoding the binary vector, `dataIn`. The two input arguments of the `poly2trellis` function indicate the code's constraint length and generator polynomials, respectively.

```
tPoly = poly2trellis([5 4],[23 35 0; 0 5 13]);
codeRate = 2/3;
```

Encode the input data using the previously created trellis.

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

Modulate Data

The encoded binary data is converted into an integer format so that 16-QAM modulation can be applied.

Reshape the input vector into a matrix of 4-bit binary data, which is then converted into integer symbols.

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

Apply 16-QAM modulation.

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

Raised Cosine Filtering

As in the "Pulse Shaping Using a Raised Cosine Filter" on page 2-22 example, RRC filtering is applied to the modulated signal before transmission. The example makes use of the `rcosdesign` function to create the filter and the `upfirdn` function to filter the data.

Specify the filter span and rolloff factor for the square-root, raised cosine filter.

```
span = 10;        % Filter span in symbols
rolloff = 0.25;   % Roloff factor of filter
```

Create the filter using the `rcosdesign` function.

```
rrcFilter = rcosdesign(rolloff, span, numSamplesPerSymbol);
```

Using the `upfirdn` function, upsample and apply the square-root, raised cosine filter.

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

AWGN Channel

Calculate the signal-to-noise ratio, SNR, based on the input E_b/N_0, the number of samples per symbol, and the code rate. Converting from E_b/N_0 to SNR requires one to account for the number of information bits per symbol. In the previous example, each symbol corresponded to k bits. Now, each symbol corresponds to $k*$`codeRate` information bits. More concretely, three symbols correspond to 12 coded bits in 16-QAM, which correspond to 8 uncoded (information) bits.

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

Pass the filtered signal through an AWGN channel.

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

Receive and Demodulate Signal

Filter the received signal using the RRC filter and remove a portion of the signal to account for the filter delay in order to make a meaningful BER comparison.

```
rxFiltSignal = upfirdn(rxSignal,rrcFilter,1,numSamplesPerSymbol);   % Downsample and fi
rxFiltSignal = rxFiltSignal(span+1:end-span);                       % Account for delay
```

Demodulate the received, filtered signal using the `qamdemod` function.

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

Viterbi Decoding

Use the `de2bi` function to convert the incoming integer symbols into bits.

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

Decode the convolutionally encoded data with a Viterbi decoder. The syntax for the `vitdec` function instructs it to use hard decisions. The 'cont' argument instructs it to use

a mode designed for maintaining continuity when the function is repeatedly invoked (as in a loop). Although this example does not use a loop, the 'cont' mode is used for the purpose of illustrating how to compensate for the delay in this decoding operation.

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

BER Calculation

Using the `biterr` function, compare `dataIn` and `dataOut` to obtain the number of errors and the bit error rate while taking the decoding delay into account. The continuous operation mode of the Viterbi decoder incurs a delay whose duration in bits equals the traceback length, `traceBack`, times the number of input streams at the encoder. For this rate 2/3 code, 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 bit error rate, the first 2×`traceBack` bits in `dataOut` and the last 2×`traceBack` bits in the original vector, `dataIn`, are discarded. Without delay compensation, the BER computation is meaningless.

```matlab
decDelay = 2*traceBack;                                  % Decoder delay, in bits
[numErrors, ber] = ...
   biterr(dataIn(1:end-decDelay),dataOut(decDelay+1:end));

fprintf('\nThe bit error rate = %5.2e, based on %d errors\n', ...
    ber, numErrors)

The bit error rate = 6.00e-05, based on 6 errors
```

It can be seen that for the same E_b/N_0 of 10 dB, the number of errors when using FEC is reduced as the BER is improves from 2.0×10^{-3} to 6.9×10^{-4}.

More About Delays

The decoding operation in this example incurs a delay, which means that the output of the decoder lags the input. Timing information does not appear explicitly in the example, and the duration of the delay depends on the specific operations being performed. Delays occur in various communications-related operations, including convolutional decoding, convolutional interleaving/deinterleaving, equalization, and filtering. To find out the duration of the delay caused by specific functions or operations, refer to the specific documentation for those functions or operations. For example:

- The `vitdec` reference page
- "Delays of Convolutional Interleavers"
- "Fading Channels"

Iterative Design Workflow for Communication Systems

In this section...
"Simulate a basic communications system" on page 2-34
"Introduce convolutional coding and hard-decision Viterbi decoding" on page 2-38
"Improve results using soft-decision decoding" on page 2-42
"Use turbo coding to improve BER performance" on page 2-47
"Apply a Rayleigh channel model" on page 2-50
"Use OFDM-based equalization to correct multipath fading" on page 2-54
"Use multiple antennas to further improve system performance" on page 2-57
"Accelerate the simulation using MATLAB Coder" on page 2-60

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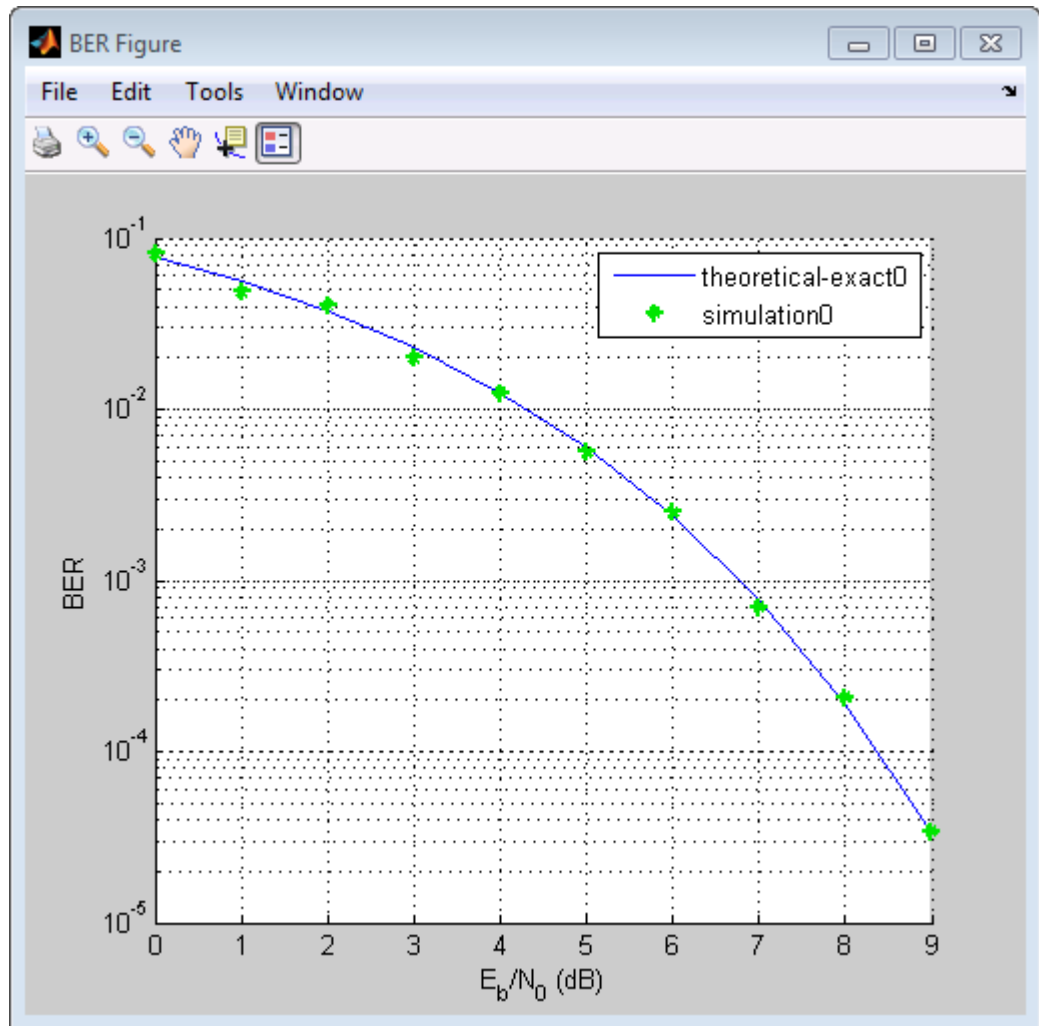
Every function with two output variables and three input variables can be called using BERTool. This is how you interpret the three input variables:

- 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.
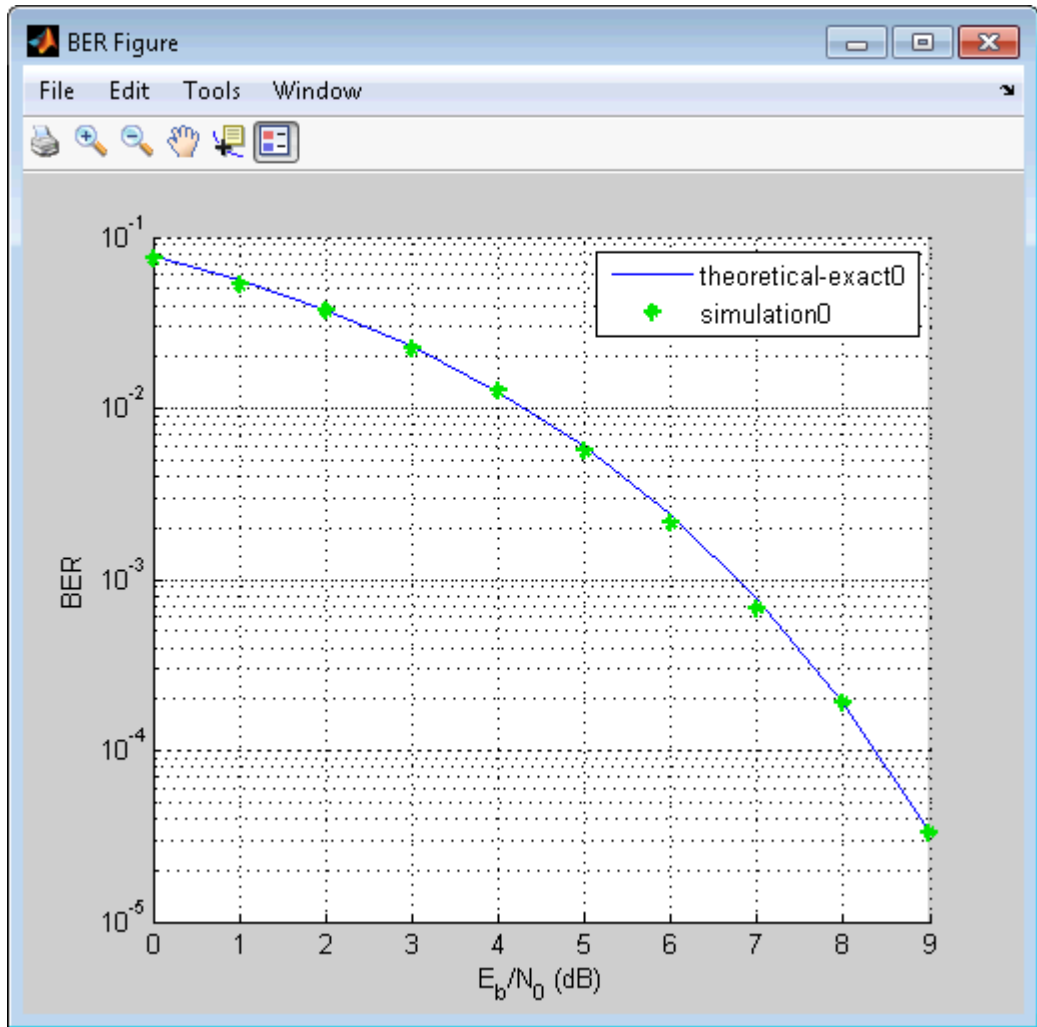
Introduce convolutional coding and hard-decision Viterbi decoding

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**.
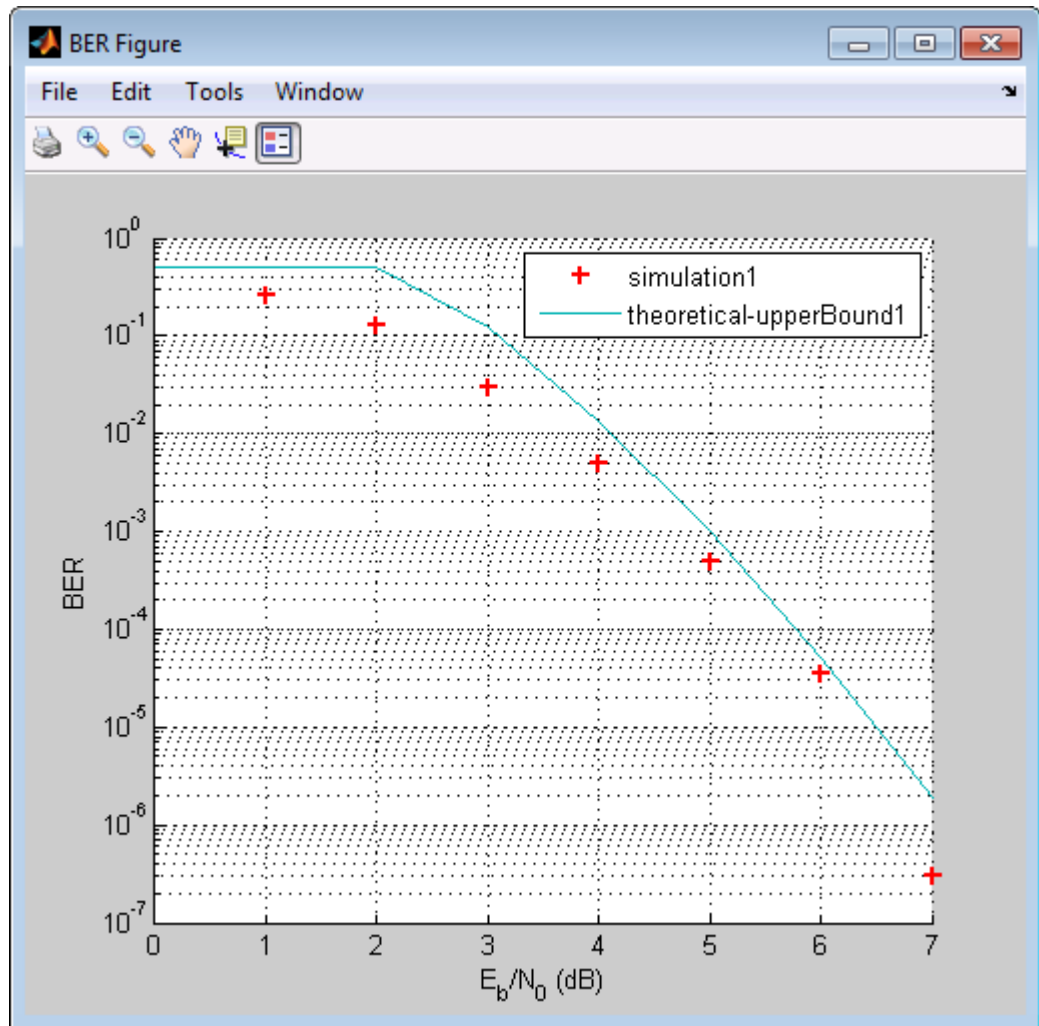**6**   Select **Hard** for the **Decision method**.

     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.
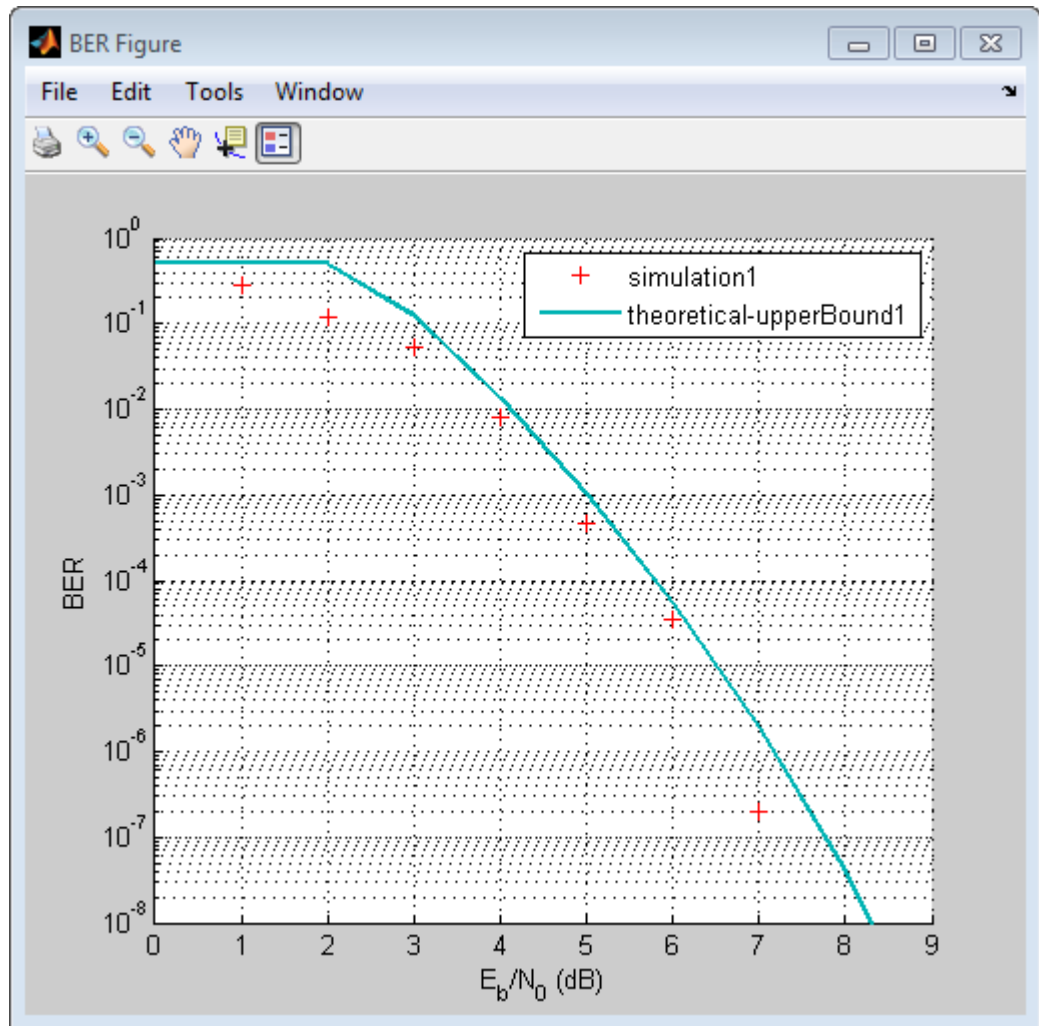
Improve results using soft-decision decoding

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**.
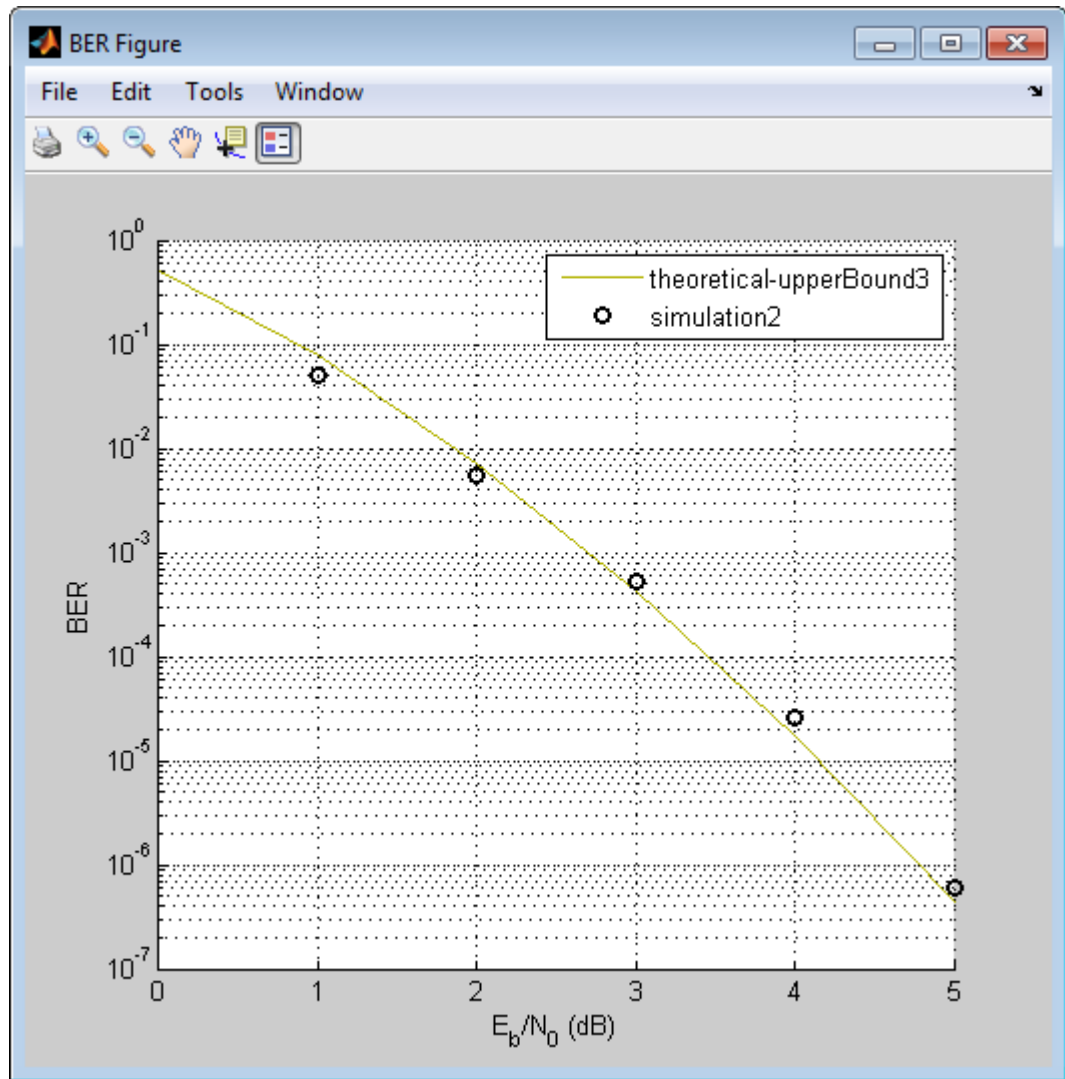**5** Select **Soft** for the **Decision method**.

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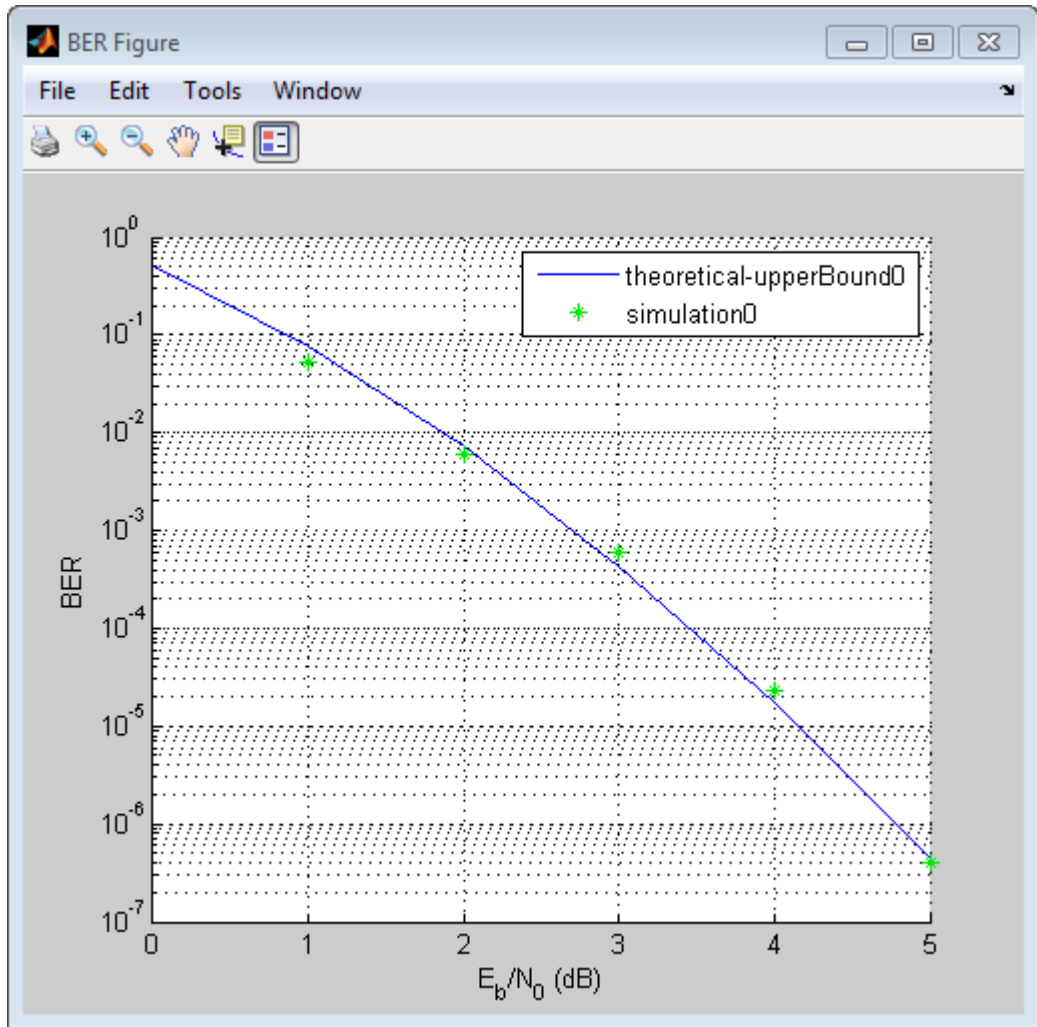
When you plot the soft-decision theoretical curve, you will observe BER curve improvements of about 2 dB relative to the hard-decision decoding. Notice that the simulation results also reflects a similar BER improvement.

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.
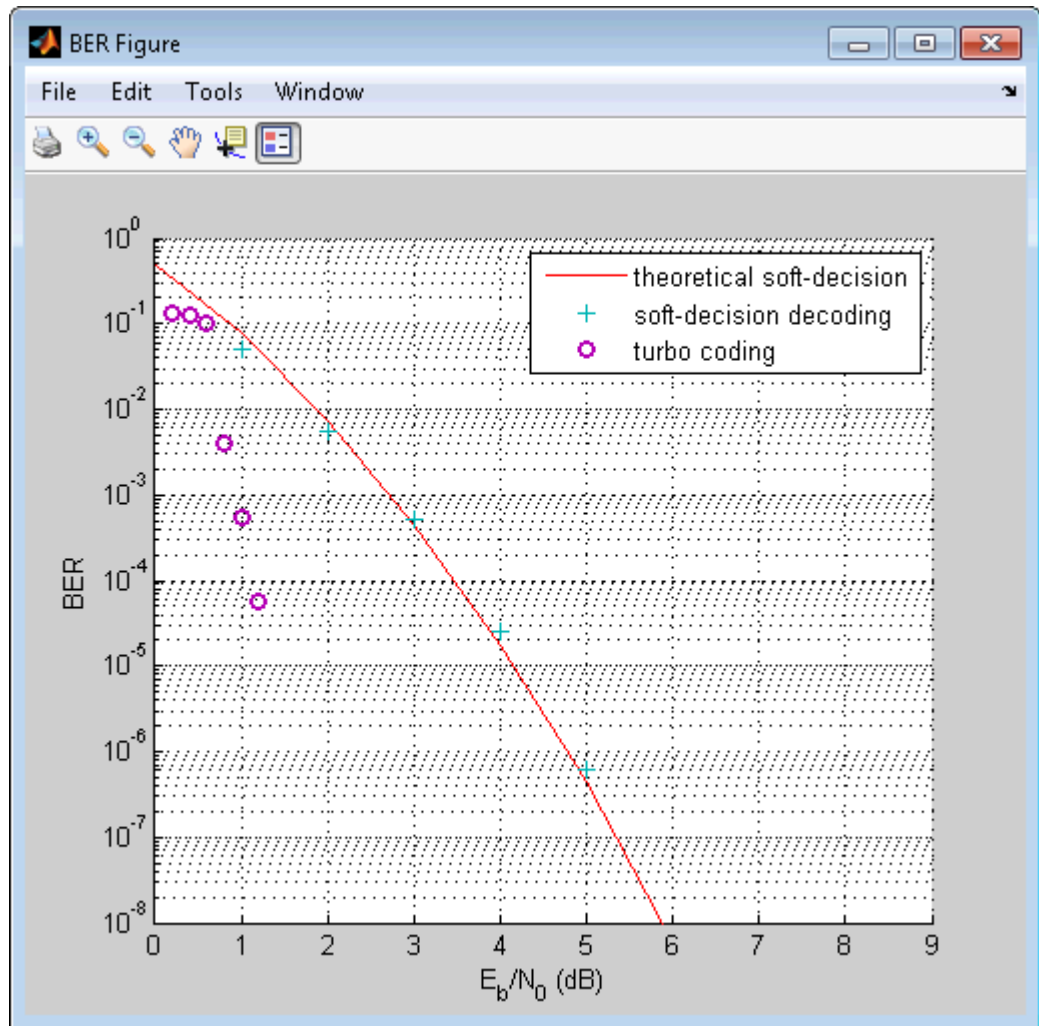
In Simulink

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.
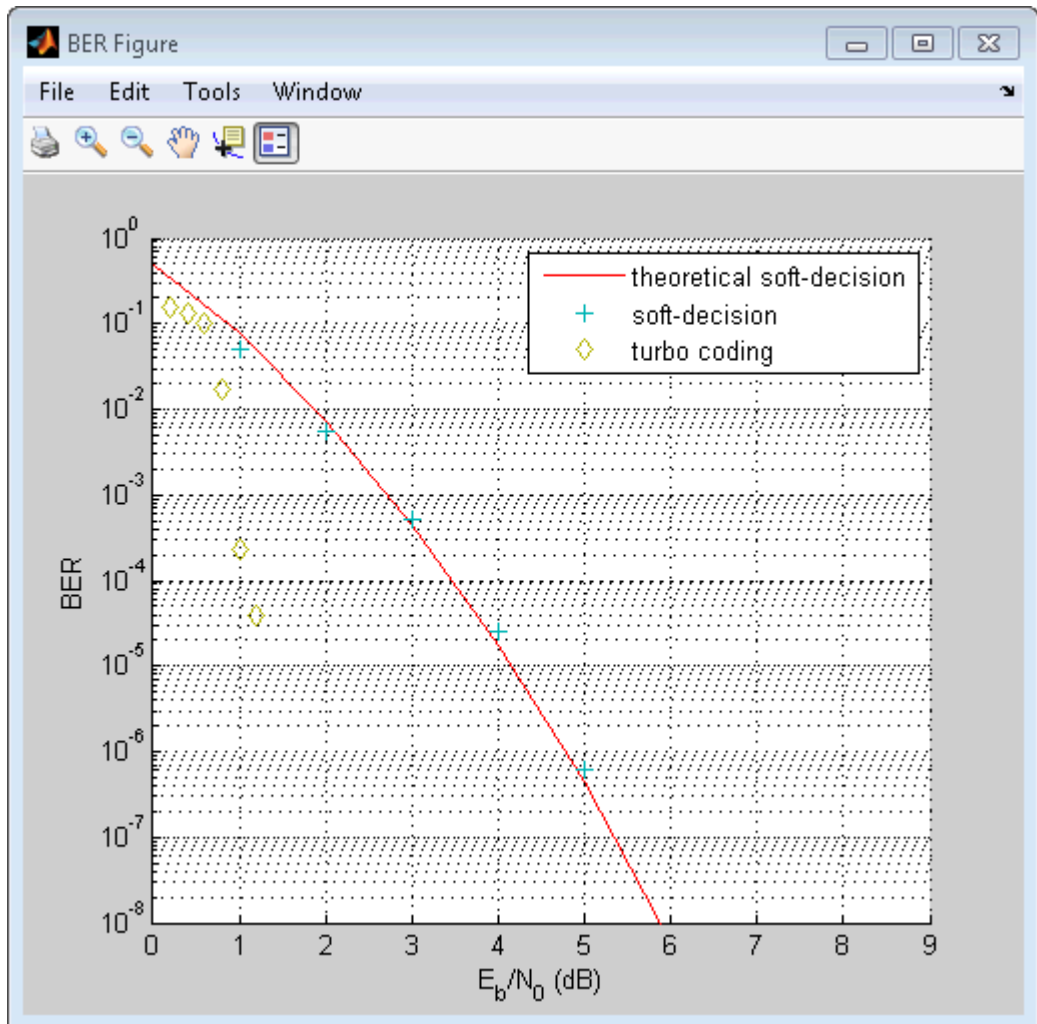
Apply a Rayleigh channel model

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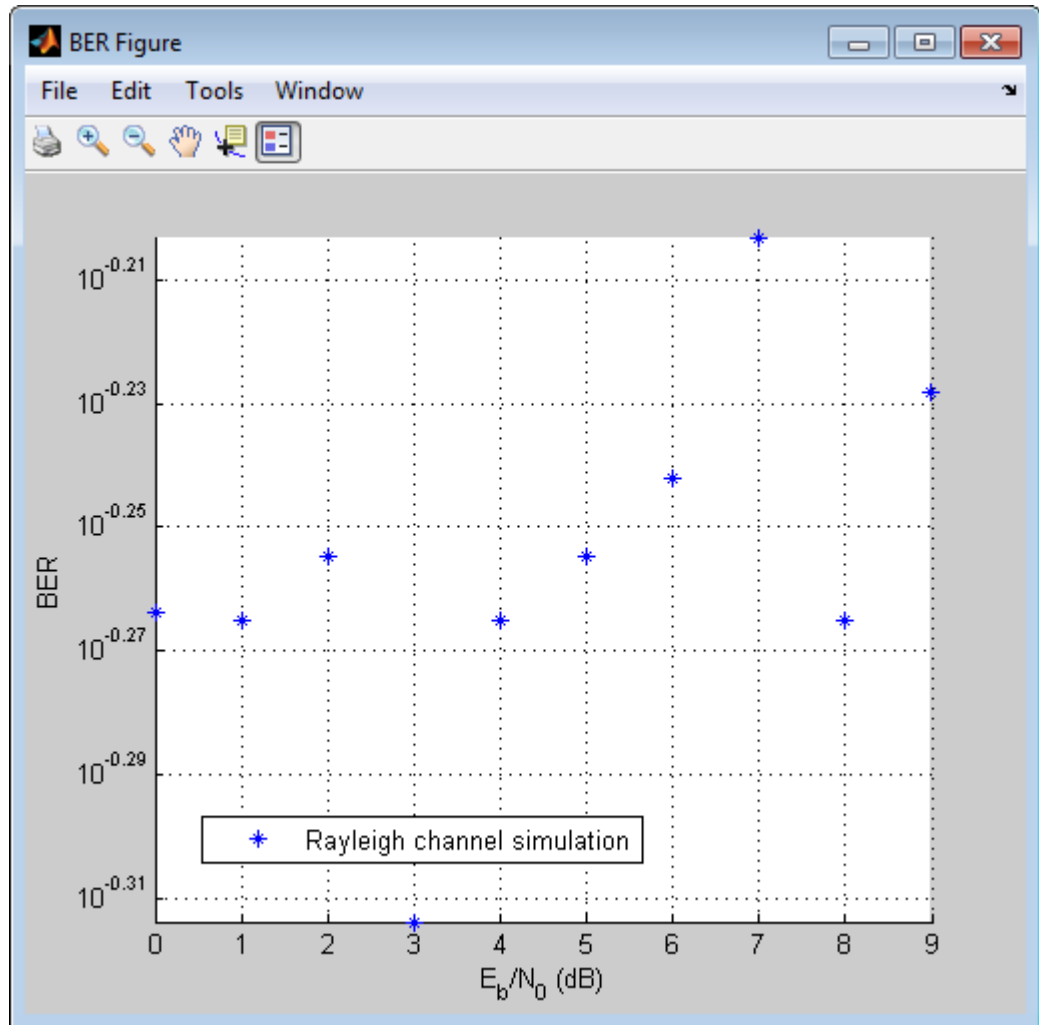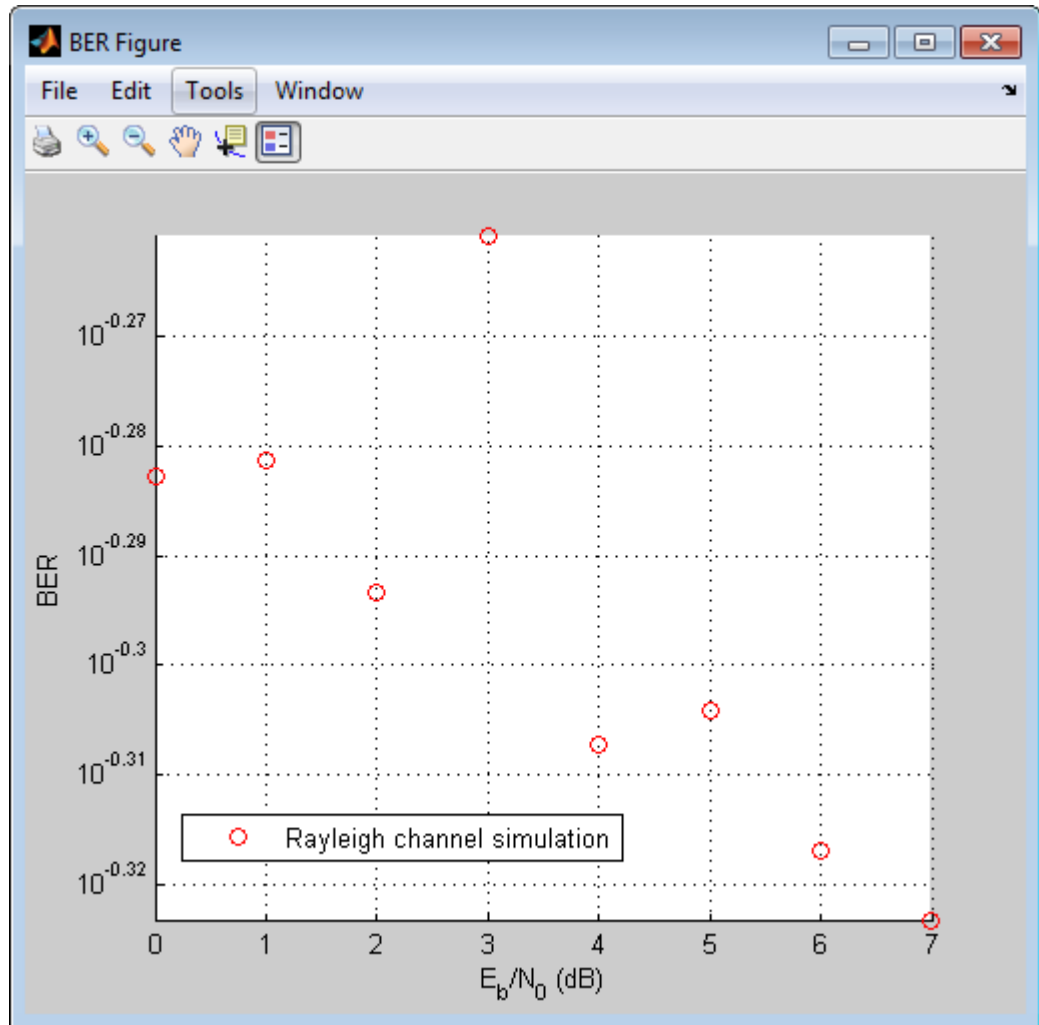
In the presence of multipath fading, the BER performance reduces to that of a binary channel with a consistent value of one-half. To correct the effect of multipath fading, you must add equalization to the communications system.

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.
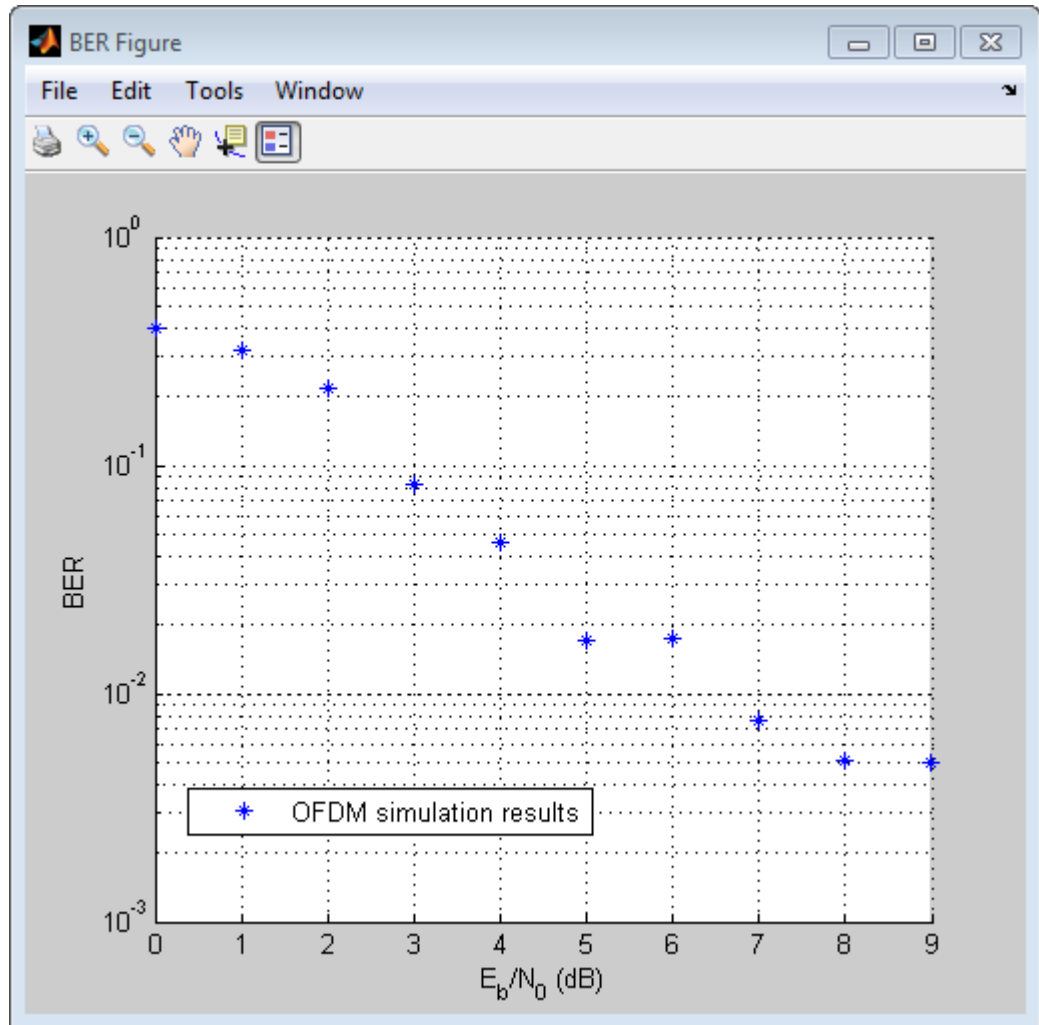
In Simulink

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.
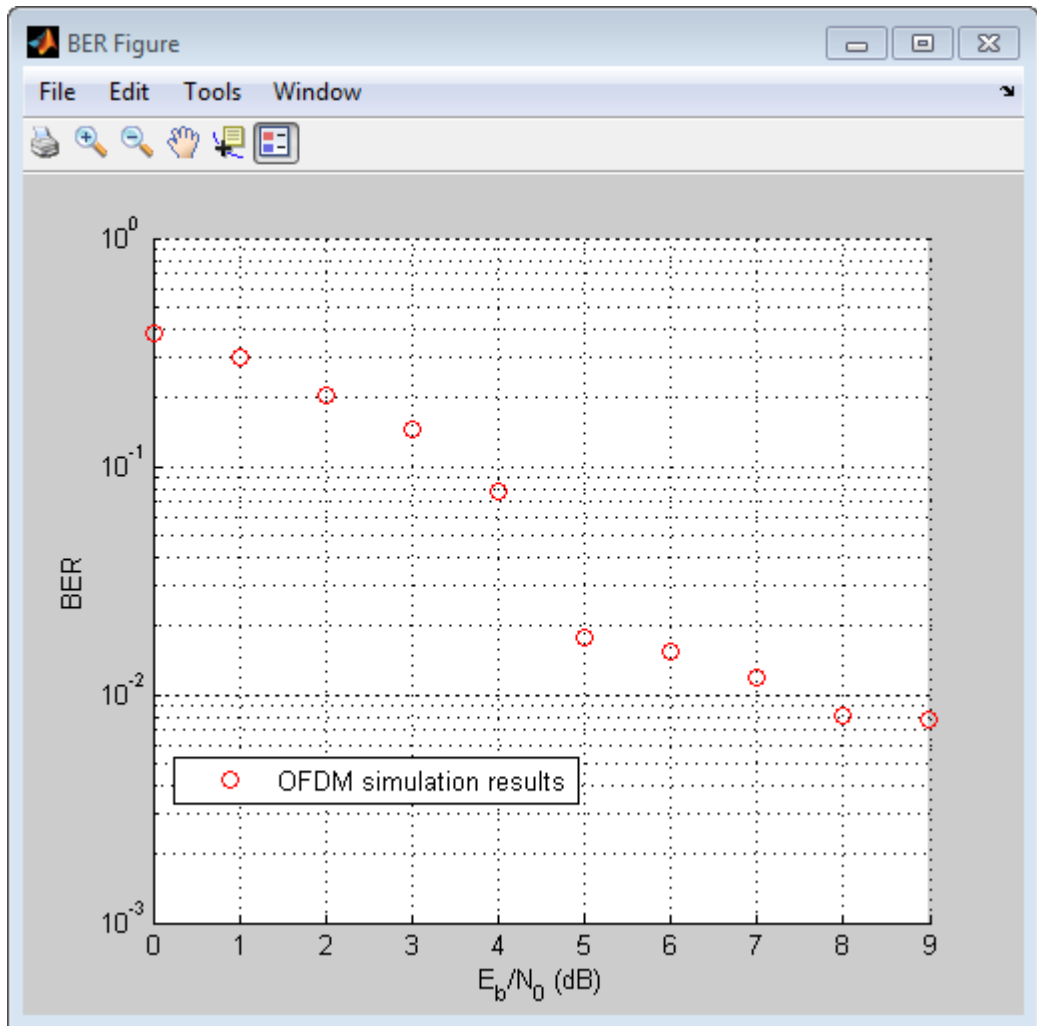
Use multiple antennas to further improve system performance

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.
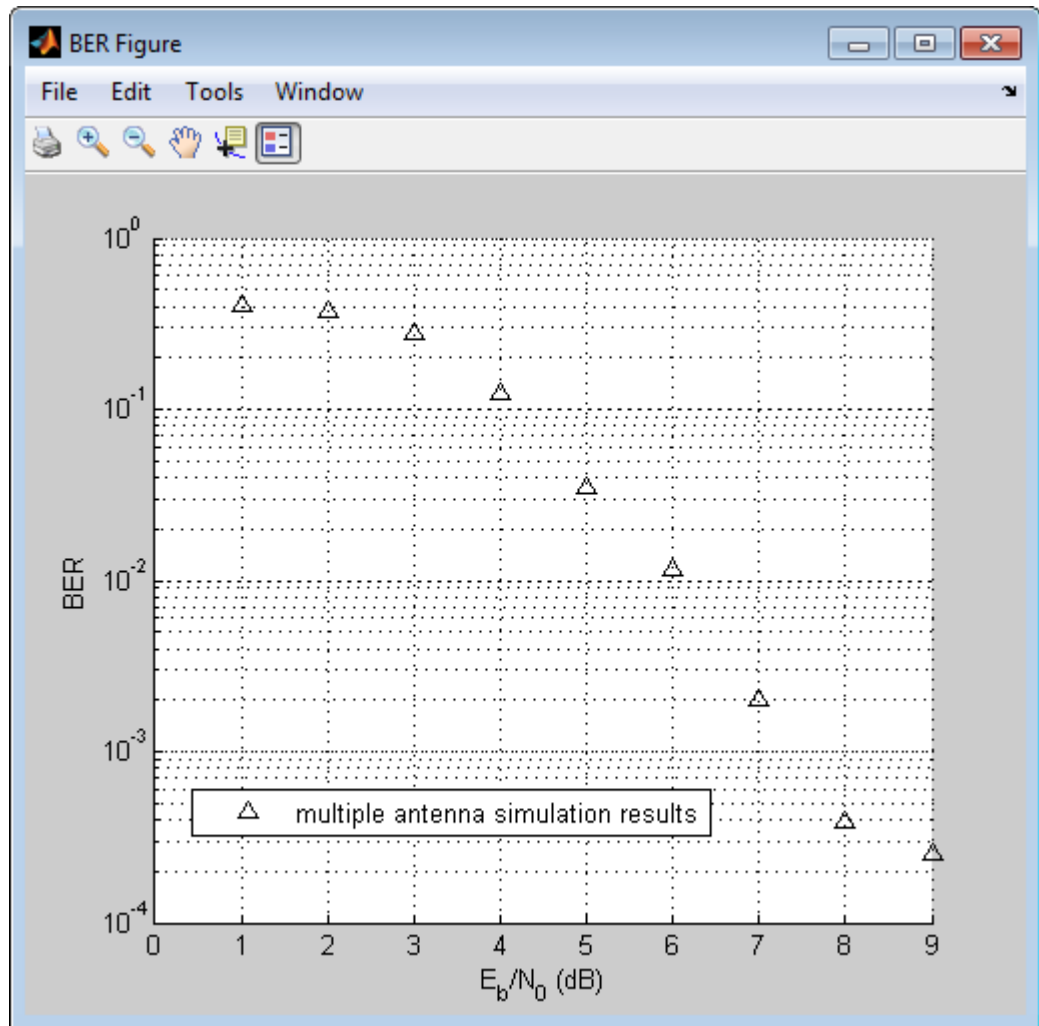
In Simulink

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**.
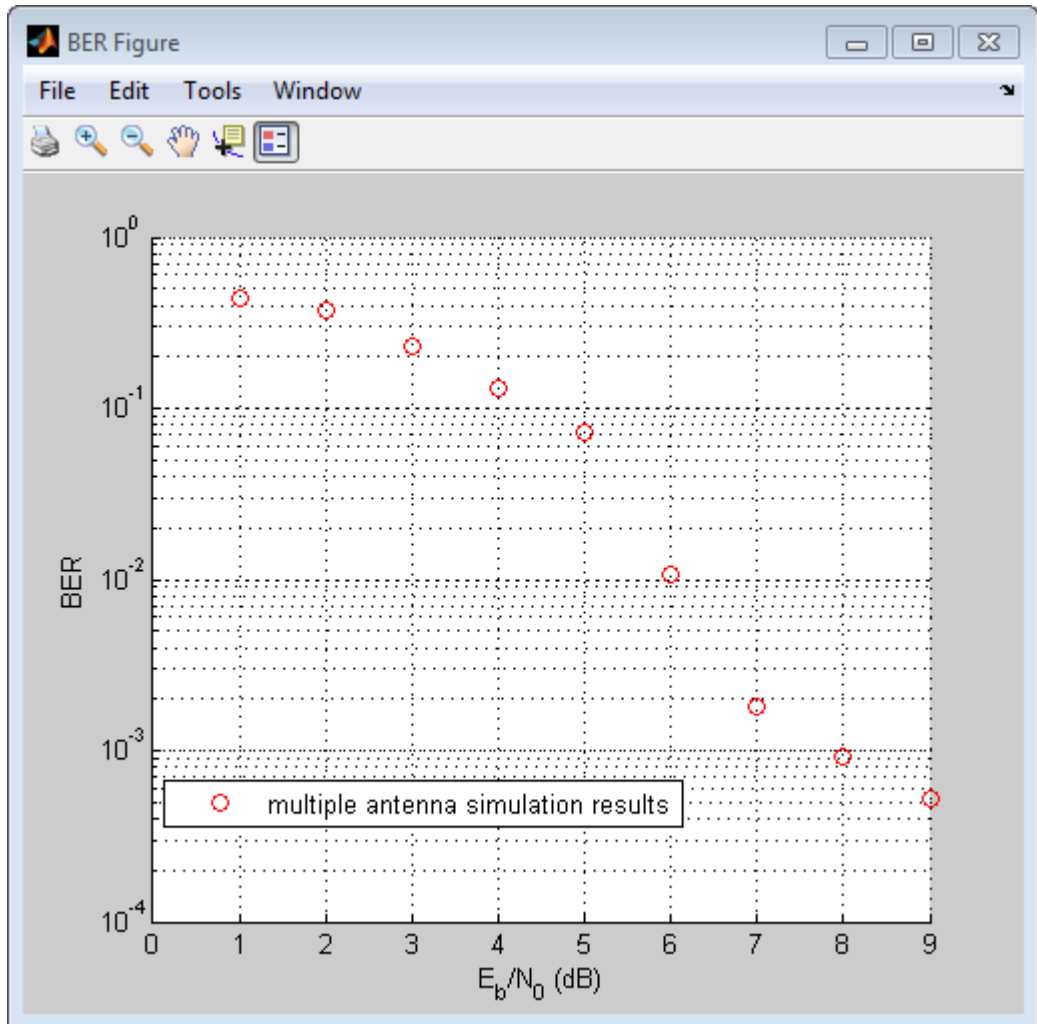
Accelerate the simulation using MATLAB Coder

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.
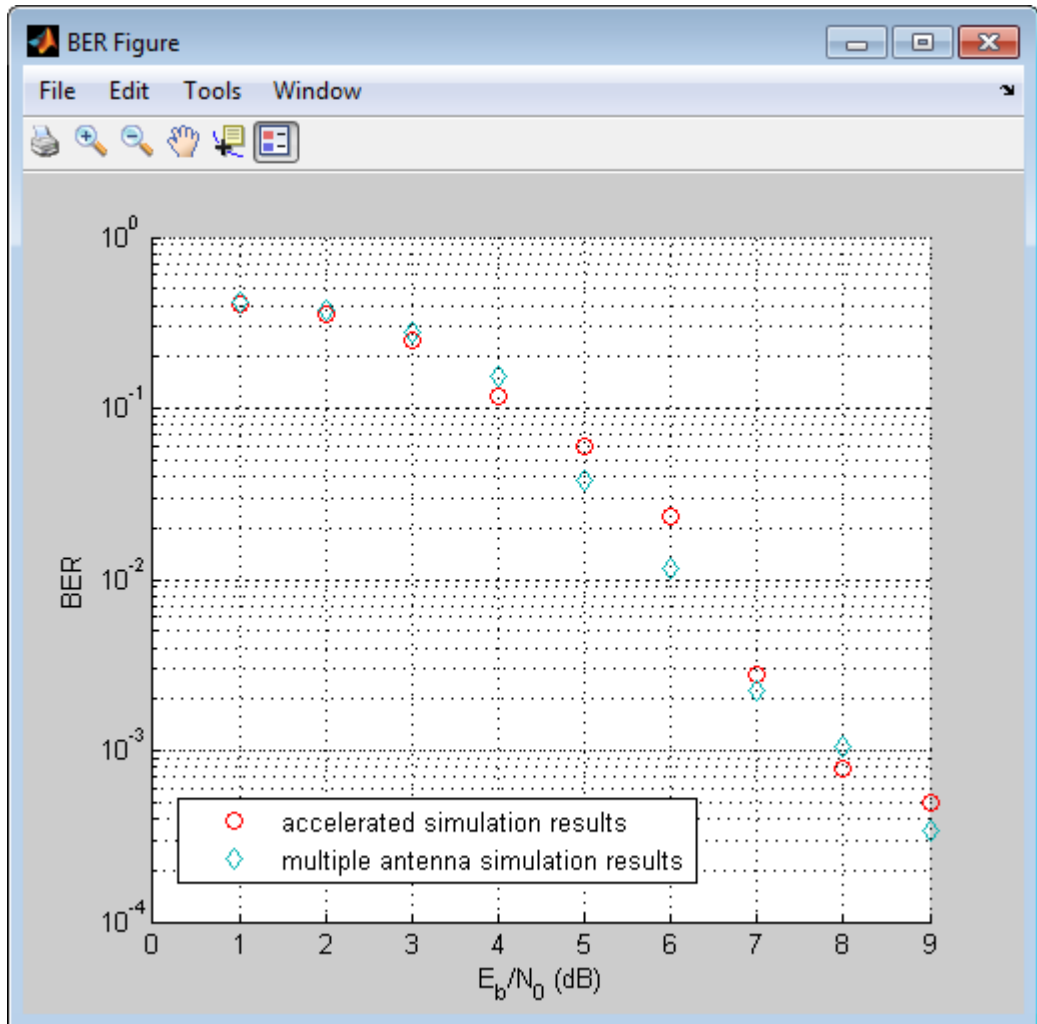
QPSK and OFDM with MATLAB System Objects

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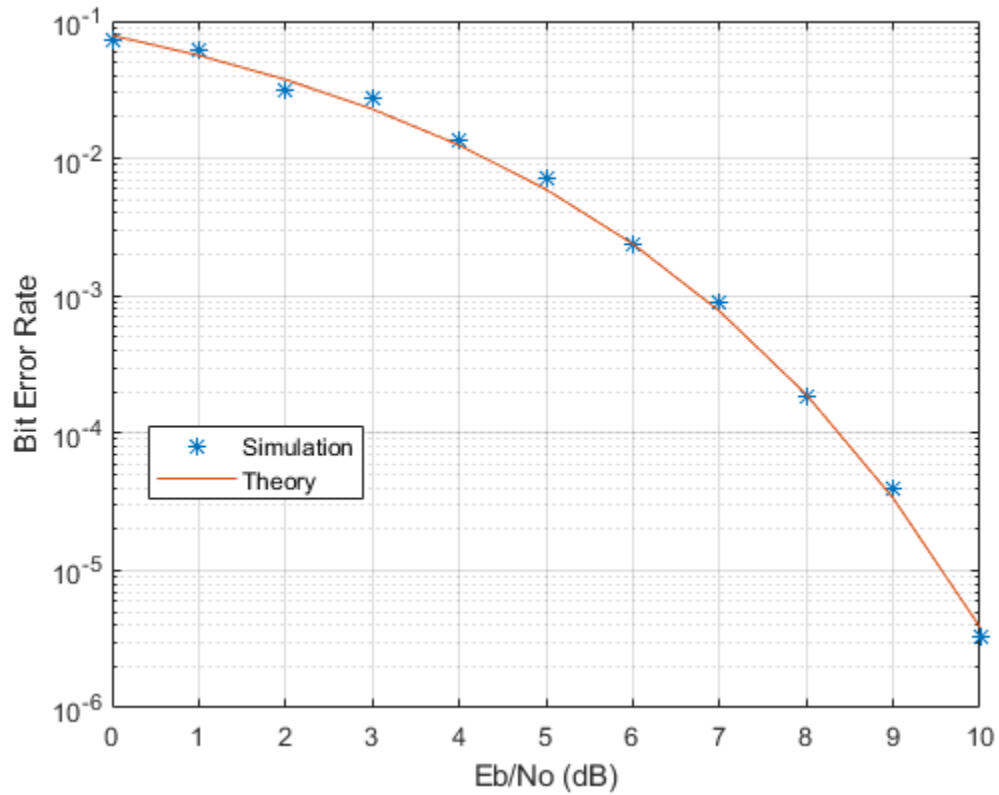
Use the `berawgn` function to determine the theoretical BER for a QPSK system.

```
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 shows how to use the 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 the Parallel Computing Toolbox is available. Create a pool of workers.

```
pool = gcp;
```

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')
```

2-67

```
xlabel('Eb/No (dB)')
ylabel('BER')
grid
```
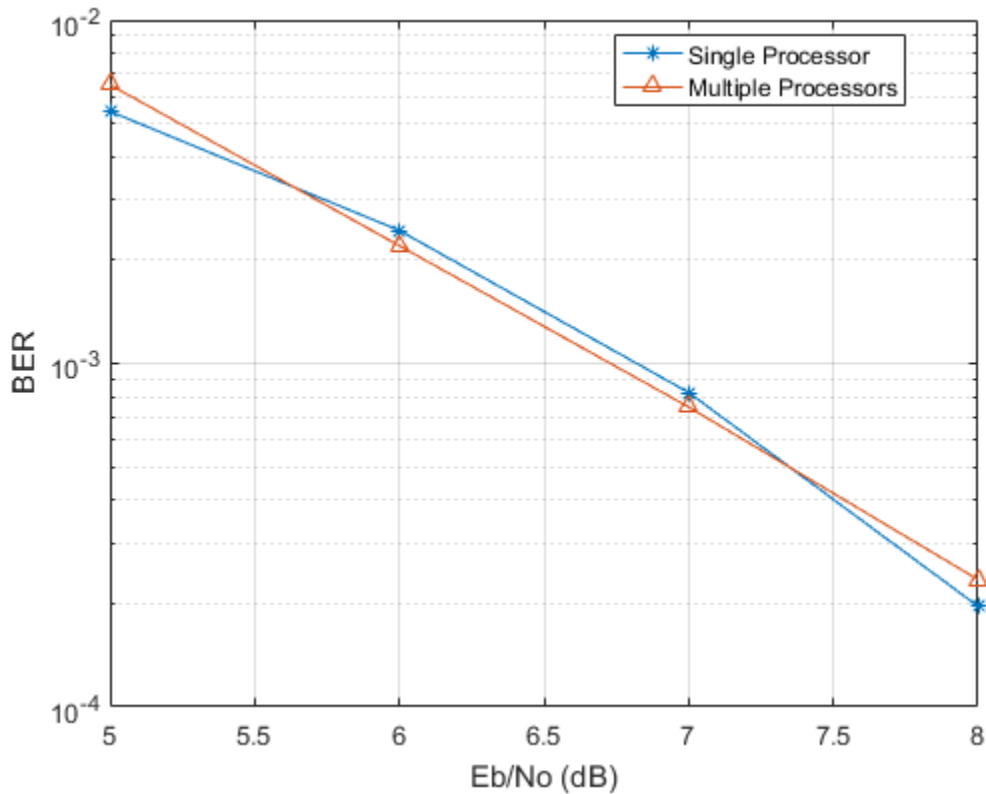

You can see that the BER curves are essentially the same with any variance being due to differing random number seeds.

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 = 101.0 sec for one worker
Simulation time = 32.8 sec for multiple workers
```

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

What Is Different About Using Communications Toolbox Online?

3

Visualization and Measurements

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')
```
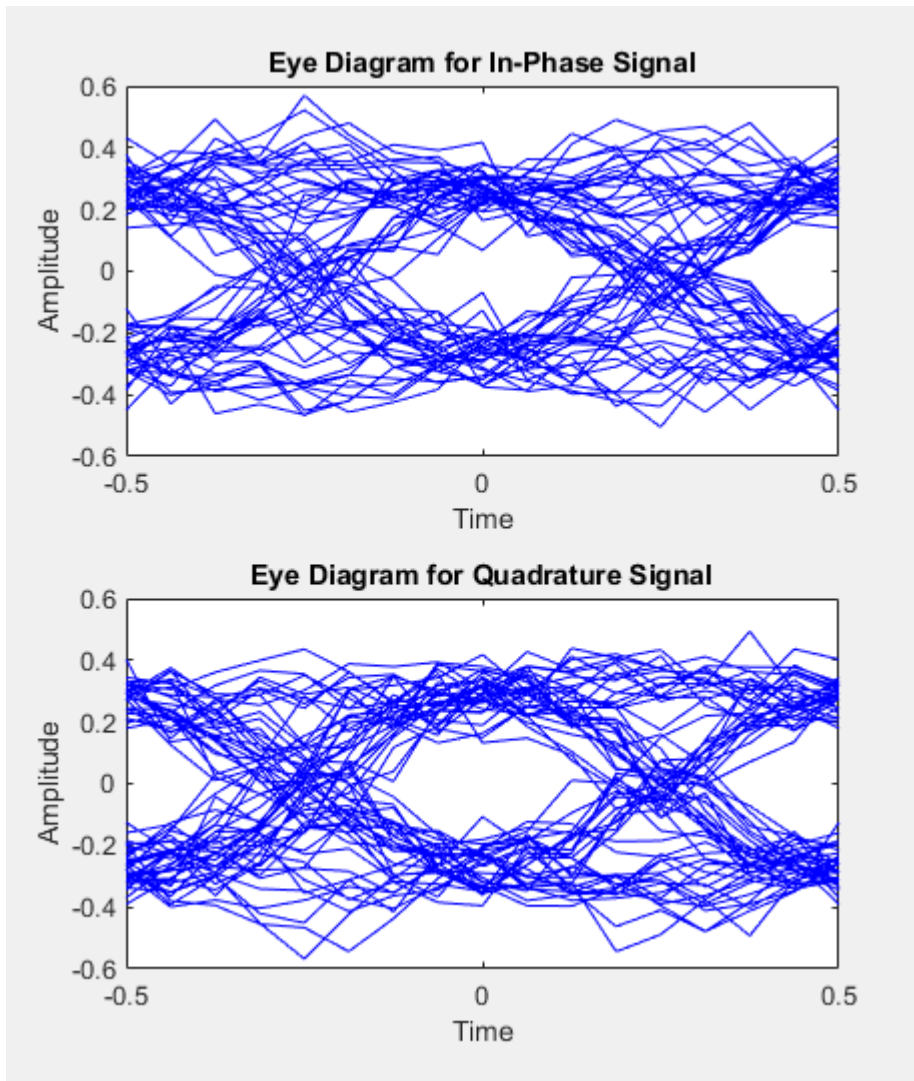
Scatter plot
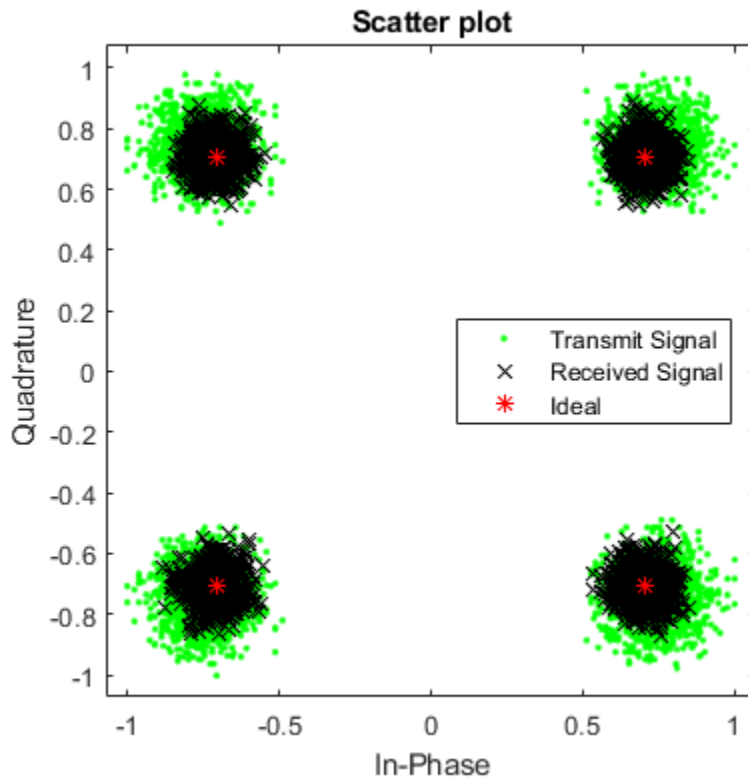
Eye Diagram

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

```
eyediagram(txSig(sps*span+1:sps*span+1000),2*sps)
```
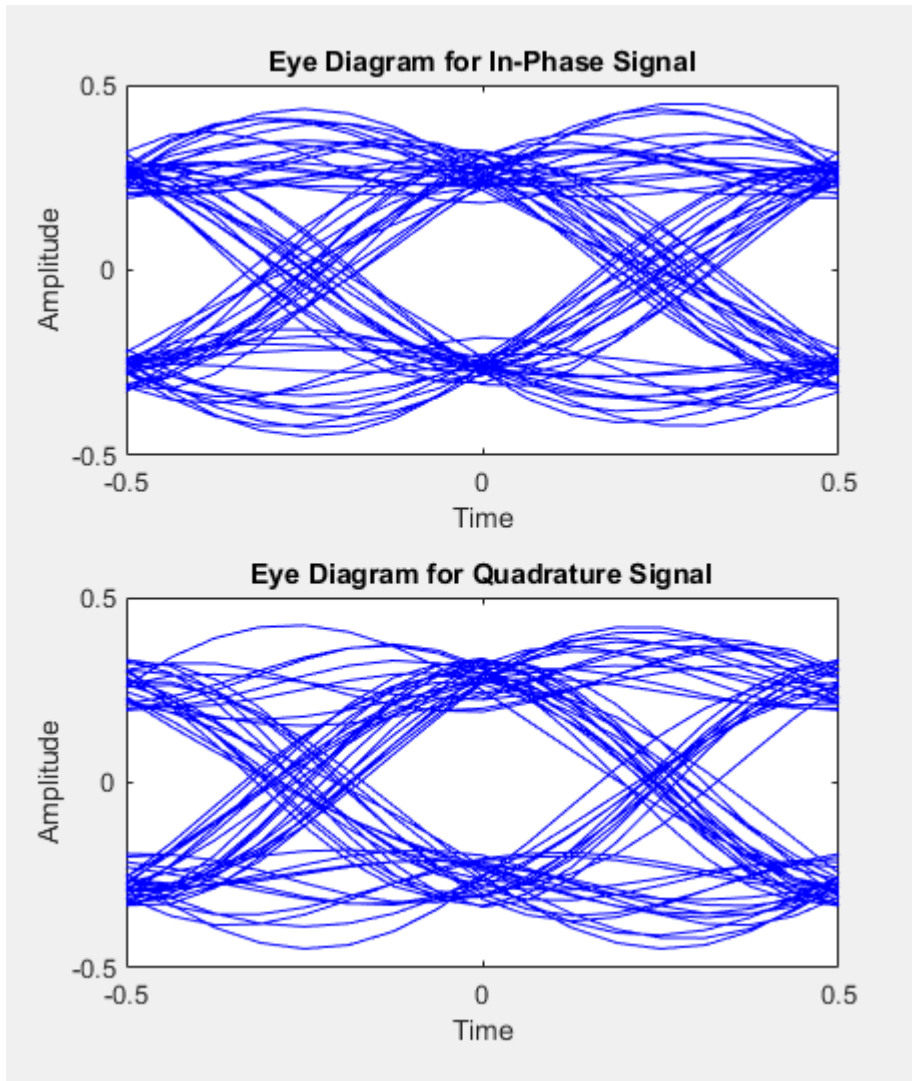
Display 1000 points of the received signal eye diagram.

```
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.
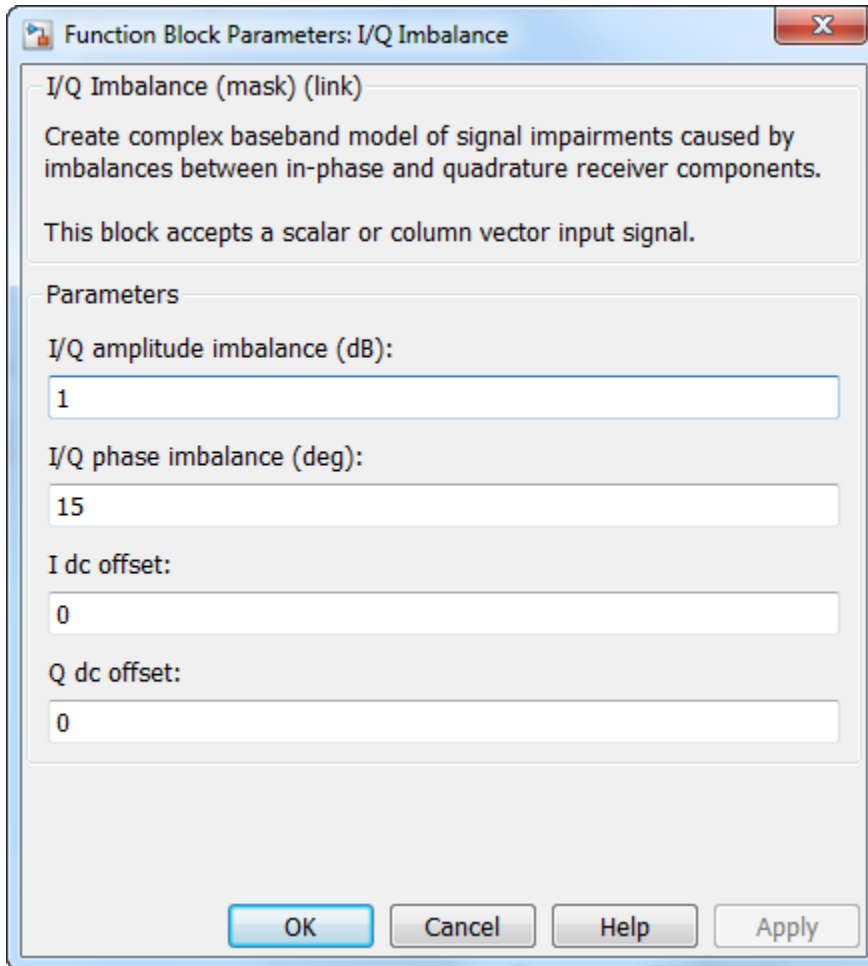
doc_mer_and_evm

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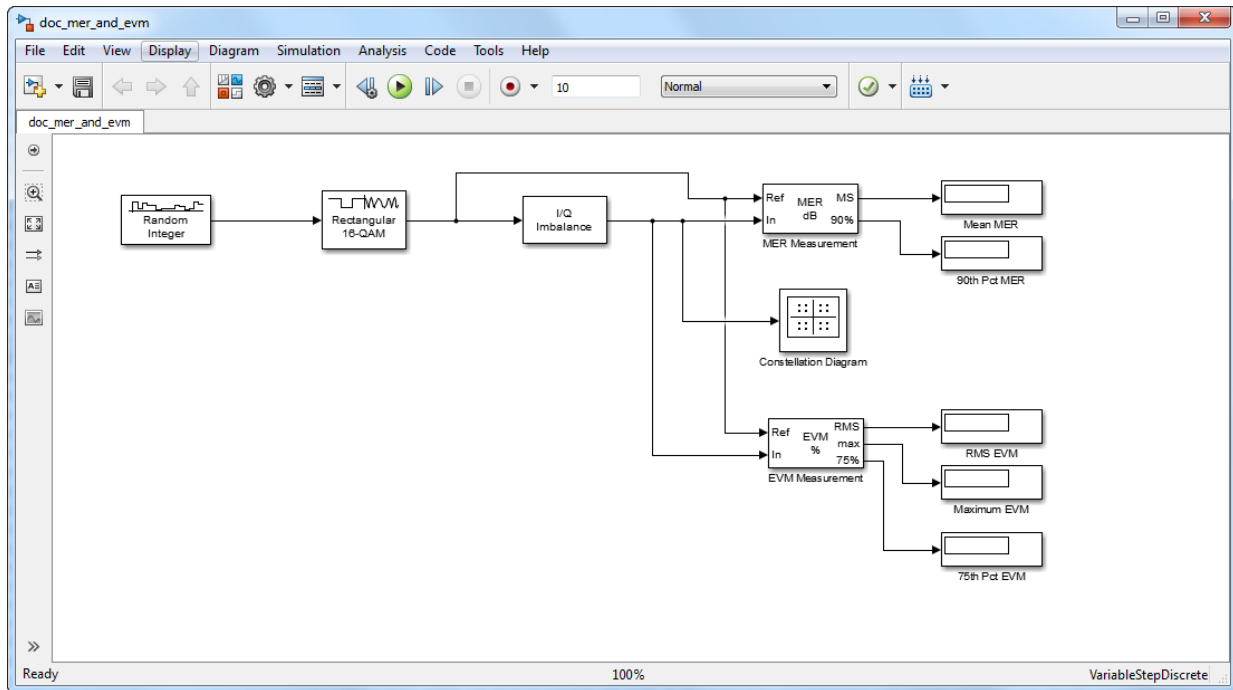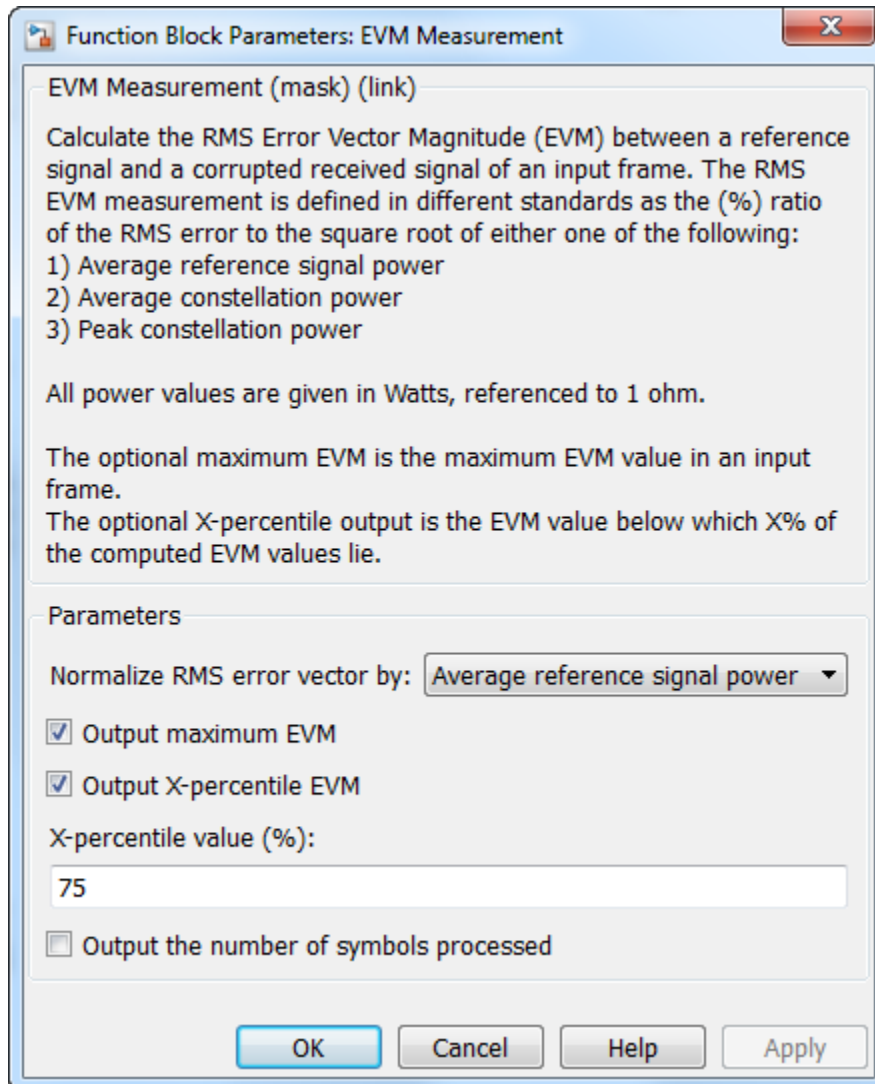
The I/Q Imbalance block is set to that the **I/Q amplitude imbalance (dB)** is set to 1 and the **I/Q phase imbalance (deg)** is set to 15.

Function Block Parameters: I/Q Imbalance

I/Q Imbalance (mask) (link)

Create complex baseband model of signal impairments caused by imbalances between in-phase and quadrature receiver components.

This block accepts a scalar or column vector input signal.

Parameters

I/Q amplitude imbalance (dB):

1

I/Q phase imbalance (deg):

15

I dc offset:

0
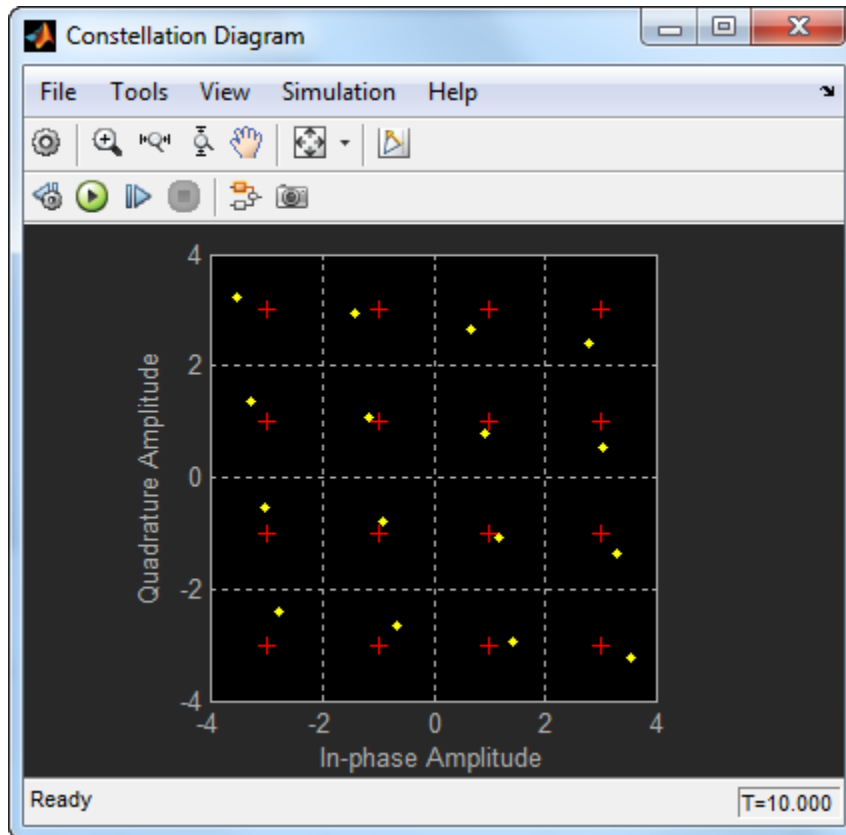
Q dc offset:

0

| OK | Cancel | Help | Apply |

The MER Measurement block is set so that it outputs the X-percentile MER value which is set to 90%.
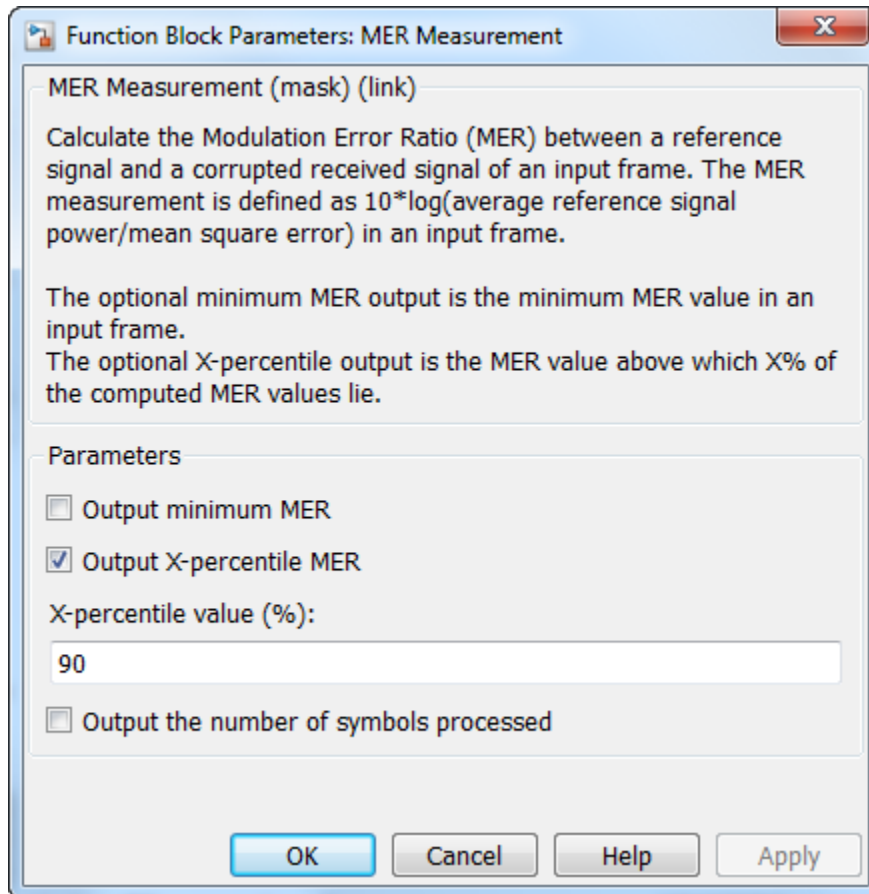
The EVM Measurement block is set to output the maximum and 75th percentile EVM values.

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%.

Mean MER

16.88

90th Pct MER

13.87

RMS EVM

14.32

Maximum EVM

20.4

75th Pct EVM

17.69
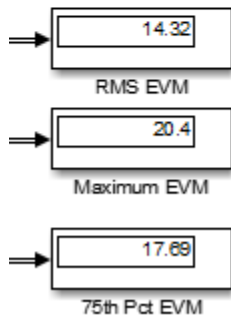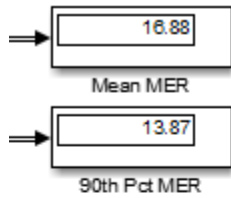
Change the **I/Q amplitude imbalance (dB)** value in the I/Q Imbalance block to 2 dB. You can see that the all the MER and EVM metrics degrade.

Mean MER

90th Pct MER

RMS EVM

Maximum EVM

75th Pct EVM
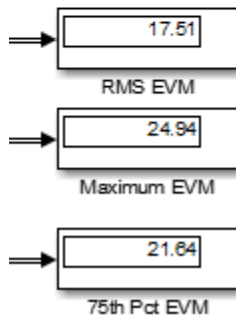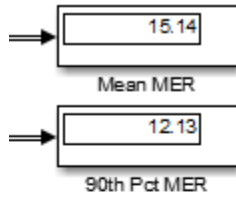
ACPR and CCDF Measurements with MATLAB System Objects

In this section...
"ACPR Measurements" on page 3-15
"CCDF Measurements" on page 3-18

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
```
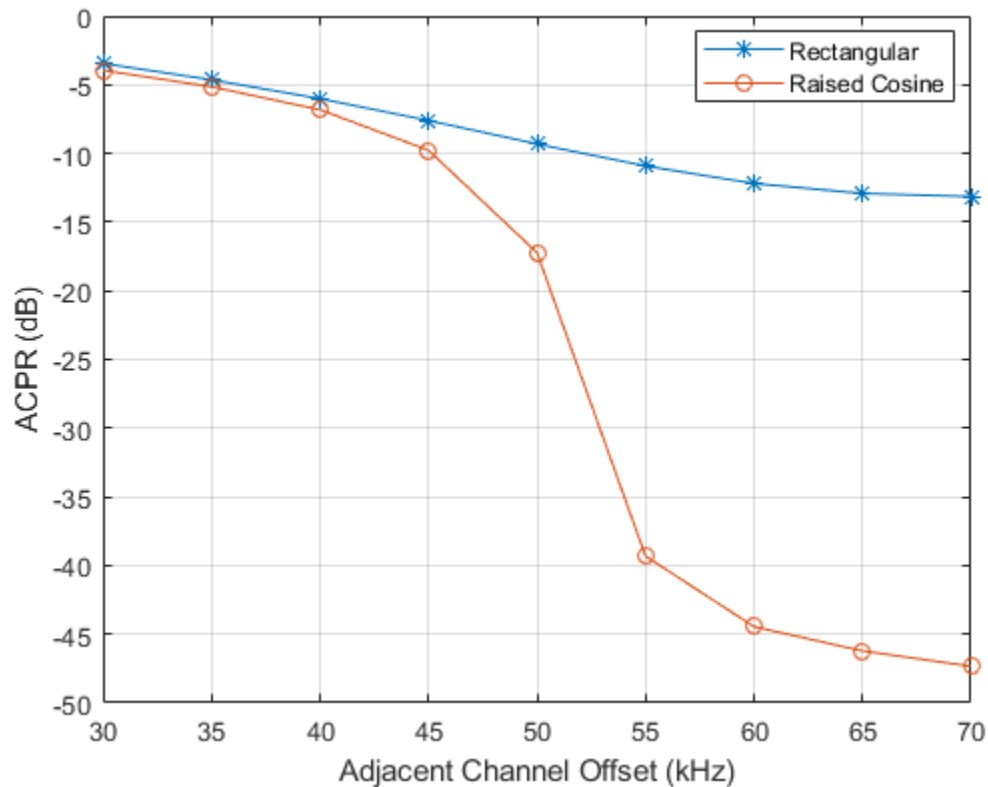
CCDF Measurements

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')
```
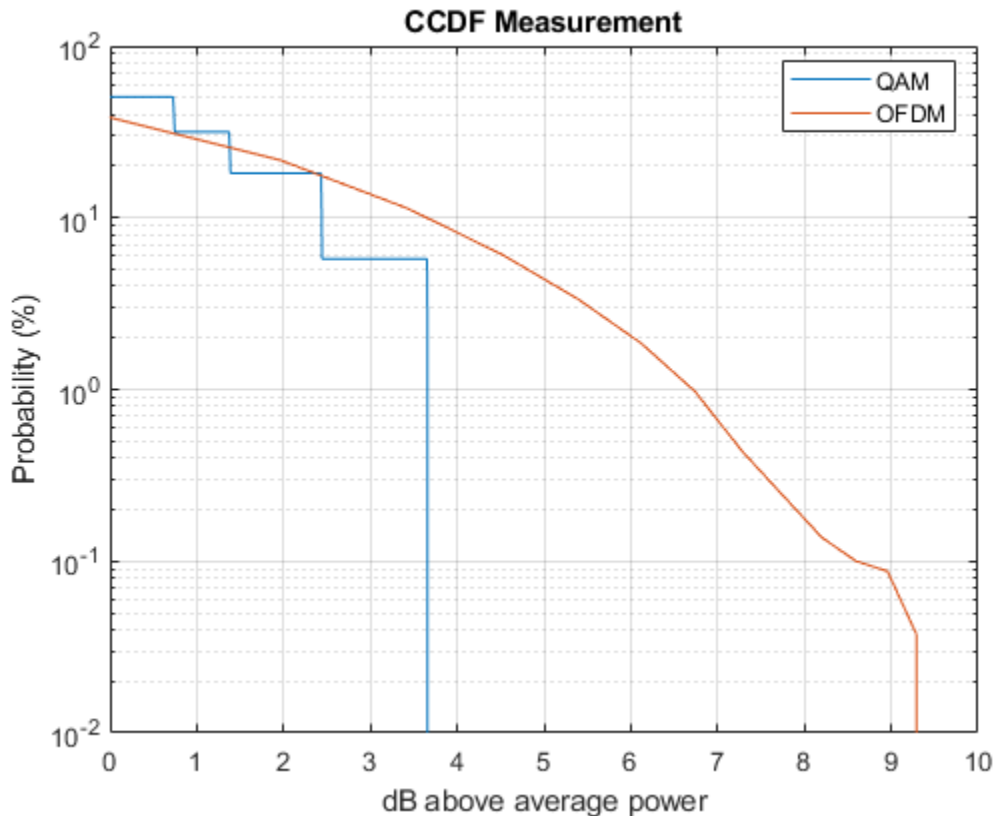

Compare the PAPR values for the QAM modulated and OFDM modulated signals.

```
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.