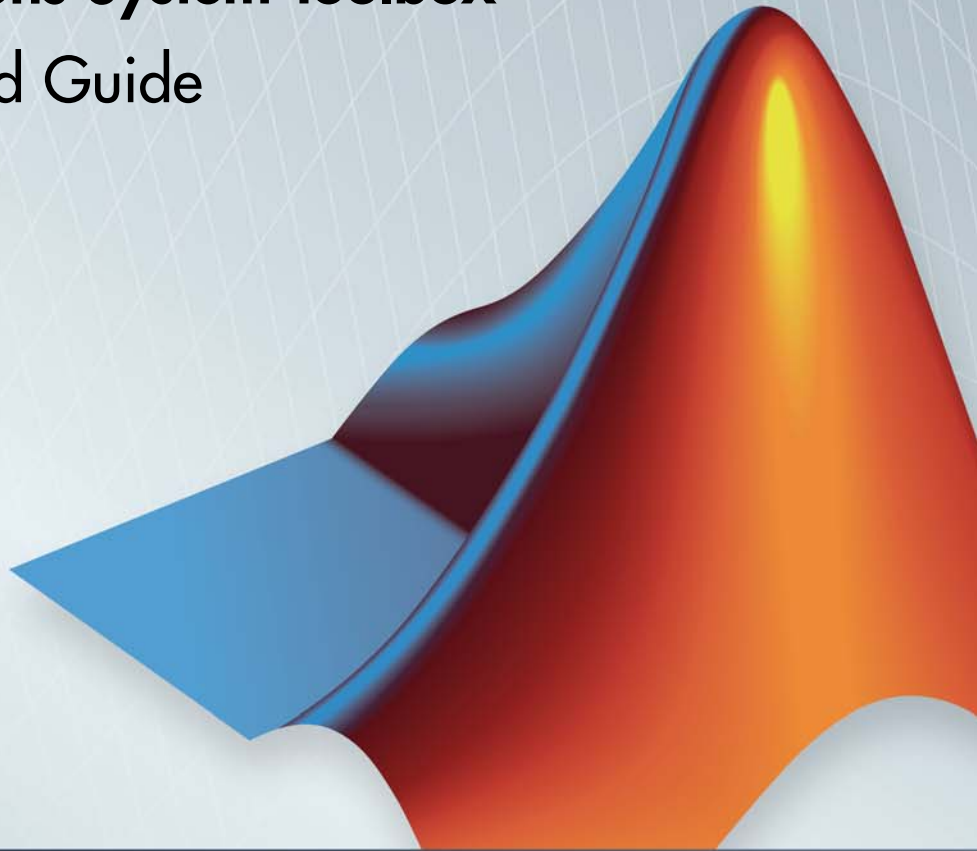


Communications System Toolbox™

Getting Started Guide

R2014a



MATLAB® & SIMULINK®



How to Contact MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Communications System Toolbox™ Getting Started Guide

© COPYRIGHT 2011–2014 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
September 2011 Online only
March 2012 Online only
September 2012 Online only
March 2013 Online only
September 2013 Online only
March 2014 Online only

New for Version 5.0 (Release 2011a)
Revised for Version 5.1 (Release 2011b)
Revised for Version 5.2 (Release 2012a)
Revised for Version 5.3 (Release 2012b)
Revised for Version 5.4 (Release 2013a)
Revised for Version 5.5 (Release 2013b)
Revised for Version 5.6 (Release 2014a)

Introduction

1

Communications System Toolbox Product

Description	1-2
Key Features	1-2

System Setup

Required Products	1-3
Expected Background	1-3
Configure the Simulink Environment for Communications Models	1-3

Access the Block Libraries

1-5

System Simulation

2

256-QAM with Simulink Blocks

2-2

Section Overview	2-2
Opening the Model	2-2
Overview of the Model	2-3
Quadrature Amplitude Modulation	2-4
Run a Simulation	2-5
Display the Error Rate	2-6
Set Block Parameters	2-7
Display a Phase Noise Plot	2-9

16-QAM with MATLAB Functions

2-11

Introduction	2-11
Modulate a Random Signal	2-11
Plot Signal Constellations	2-18
Pulse Shaping Using a Raised Cosine Filter	2-23
Error Correction using a Convolutional Code	2-30

Iterative Design Workflow for Communication	
Systems	2-35
Simulate a basic communications system	2-36
Introduce convolutional coding and hard-decision Viterbi decoding	2-41
Improve results using soft-decision decoding	2-46
Use turbo coding to improve BER performance	2-51
Apply a Rayleigh channel model	2-54
Use OFDM-based equalization to correct multipath fading	2-59
Use multiple antennas to further improve system performance	2-62
Accelerate the simulation using MATLAB Coder	2-66
QPSK and OFDM with MATLAB System Objects	2-70
Simulate a basic communications system	2-71
Introduce convolutional coding and hard-decision Viterbi decoding	2-74
Improve results using soft-decision decoding	2-76
Use turbo coding to improve BER performance	2-79
Apply a Rayleigh channel model	2-80
Use OFDM-based equalization to correct multipath fading	2-83
Use multiple antennas to further improve system performance	2-84
Accelerate the simulation using MATLAB Coder	2-86
Accelerating BER Simulations Using the Parallel Computing Toolbox	2-89

Visualization and Measurements

3

Scatter Plot and Eye Diagram with MATLAB	3-2
Scatter Plot and Eye Diagram with MATLAB	3-8
EVM and MER Measurements with Simulink	3-14

ACPR and CCDF Measurements with MATLAB	3-22
ACPR Measurements	3-22
CCDF Measurements	3-26

System Objects

4

What Is a System Toolbox?	4-2
What Are System Objects?	4-3
When to Use System Objects Instead of MATLAB	
Functions	4-5
System Objects vs. MATLAB Functions	4-5
Process Audio Data Using Only MATLAB Functions	
Code	4-5
Process Audio Data Using System Objects	4-6
System Design and Simulation in MATLAB	4-8
System Design and Simulation in Simulink	4-9
System Objects in MATLAB Code Generation	4-10
System Objects in Generated Code	4-10
System Objects in codegen	4-16
System Objects in the MATLAB Function Block	4-16
System Objects in the MATLAB System Block	4-16
System Objects and MATLAB Compiler Software	4-16
System Objects in Simulink	4-17
System Objects in the MATLAB Function Block	4-17
System Objects in the MATLAB System Block	4-17
System Object Methods	4-18
What Are System Object Methods?	4-18
The Step Method	4-18
Common Methods	4-19

System Design in MATLAB Using System Objects	4-21
Create Components for Your System	4-21
Configure Components for Your System	4-22
Assemble Components to Create Your System	4-23
Run Your System	4-25
Reconfigure Your System During Runtime	4-25
System Design in Simulink Using System Objects	4-28
Define New Kinds of System Objects for Use in Simulink ..	4-28
Test New System Objects in MATLAB	4-34
Add System Objects to Your Simulink Model	4-35

Introduction

- “Communications System Toolbox Product Description” on page 1-2
- “System Setup” on page 1-3
- “Access the Block Libraries” on page 1-5

Communications System Toolbox Product Description

Design and simulate the physical layer of communication systems

Communications System Toolbox™ provides algorithms for designing, simulating, and analyzing communications systems. These capabilities are provided as MATLAB® functions, MATLAB System objects, and Simulink® blocks. The system toolbox enables source coding, channel coding, interleaving, modulation, equalization, synchronization, and channel modeling. You can also analyze bit error rates, generate eye and constellation diagrams, and visualize channel characteristics. Using adaptive algorithms, you can model dynamic communications systems that use OFDM, OFDMA, and MIMO techniques. Algorithms support fixed-point data arithmetic and C or HDL code generation.

Key Features

- Algorithms for designing the physical layer of communications systems, including source coding, channel coding, interleaving, modulation, channel models, MIMO, equalization, and synchronization
- GPU-enabled System objects for computationally intensive algorithms such as Turbo, LDPC, and Viterbi decoders
- Eye Diagram Scope app and visualization functions for constellations and channel scattering
- Bit Error Rate app for comparing the simulated bit error rate of a system with analytical results
- Channel models, including AWGN, Multipath Rayleigh Fading, Rician Fading, MIMO Multipath Fading, and LTE MIMO Multipath Fading
- Basic RF impairments, including nonlinearity, phase noise, thermal noise, and phase and frequency offsets
- Algorithms available as MATLAB functions, MATLAB System objects, and Simulink blocks
- Support for fixed-point modeling and C and HDL code generation

System Setup

In this section...

“Required Products” on page 1-3

“Expected Background” on page 1-3

“Configure the Simulink Environment for Communications Models” on page 1-3

Required Products

The Communications System Toolbox product is part of a family of MathWorks® products. You need to install several products to use this product. For more information about the required products, see the MathWorks website, at <http://www.mathworks.com/products/communications/requirements.html>.

Expected Background

This documentation assumes that you already have background knowledge in the subject of digital communications. If you do not yet have this background, then you can acquire it using a standard communications text or the books listed in the Selected Bibliography subsections that accompany many topics.

The discussion and examples in this section are aimed at new users. Continue reading and try the examples. Then, read the subsequent content that pertains to your specific areas of interest. As you learn which System object™, block, or function you want to use, refer to the online reference pages for more information.

Configure the Simulink Environment for Communications Models

Using commstartup.m

The Communications System Toolbox product provides a file, `commstartup.m`. This file changes the default Simulink model settings to values more appropriate for the simulation of communication systems. The changes apply

to new models that you create later in the MATLAB session, but not to previously created models.


Note The DSP System Toolbox™ application includes a similar `dspstartup` script, which assigns different model settings. For modeling communication systems, you should use `commstartup` alone.

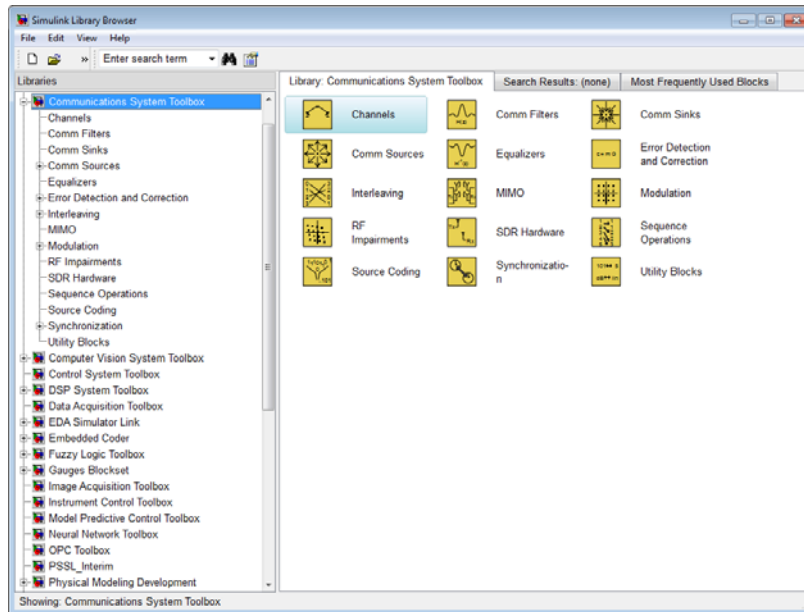
To install the communications-related model settings each time you start MATLAB, invoke `commstartup` from your `startup.m` file. The settings in `commstartup` cause models to:

- Use the variable-step discrete solver in single-tasking mode
- Use starting and ending times of 0 and `Inf`, respectively
- Avoid producing a warning or error message for inherited sample times in source blocks
- Set the Simulink Boolean logic signals parameter to `Off`
- Avoid saving output or time information to the workspace
- Produce an error upon detecting an algebraic loop
- Inline parameters if you use the Model Reference feature of Simulink

If your communications model does not work well with these default settings, you can change each of the individual settings as the model requires.

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 System Toolbox, DSP System Toolbox, and Simulink libraries.

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

System Simulation

- “256-QAM with Simulink Blocks” on page 2-2
- “16-QAM with MATLAB Functions” on page 2-11
- “Iterative Design Workflow for Communication Systems” on page 2-35
- “QPSK and OFDM with MATLAB System Objects” on page 2-70
- “Accelerating BER Simulations Using the Parallel Computing Toolbox” on page 2-89

256-QAM with Simulink Blocks

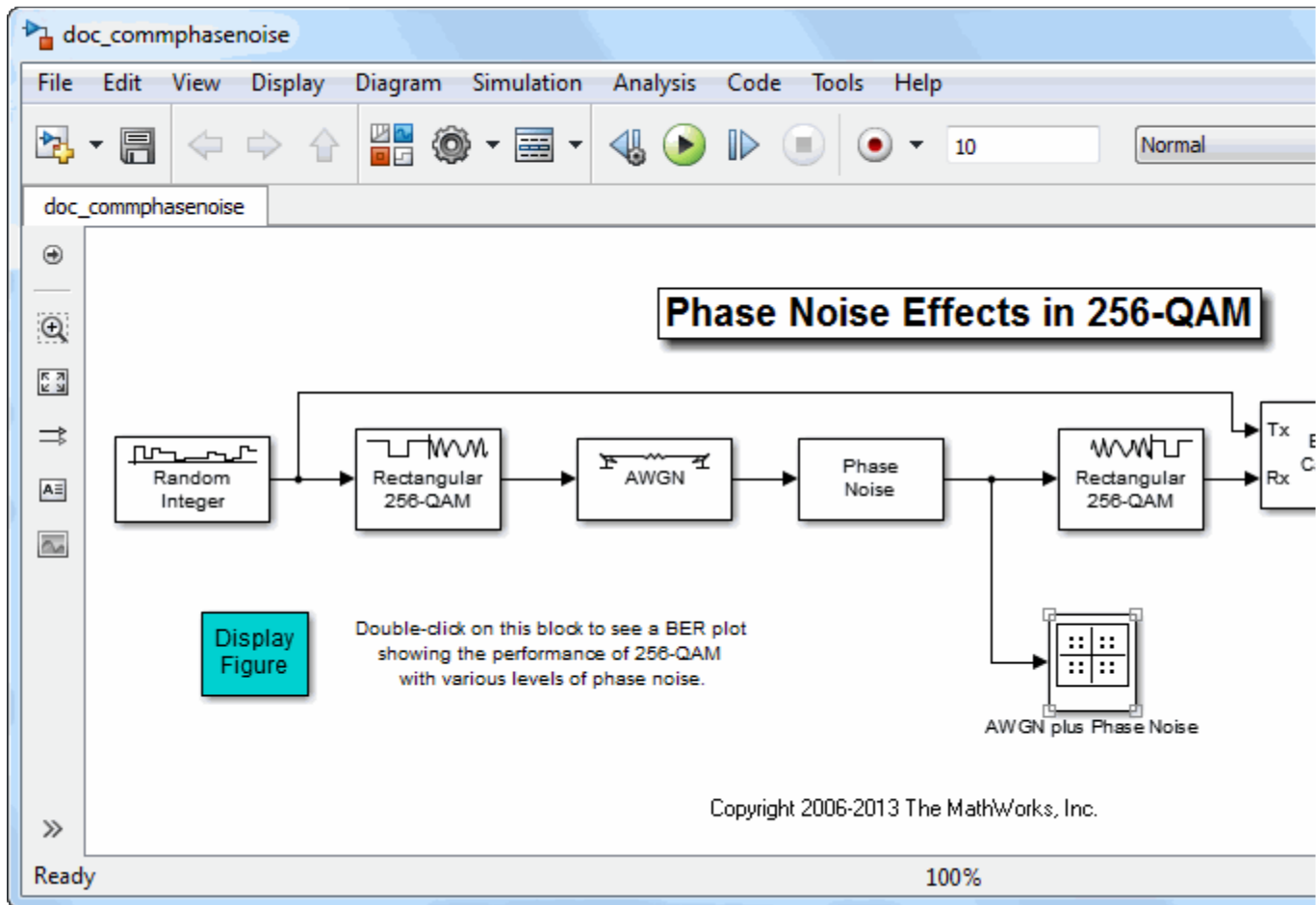
In this section...
“Section Overview” on page 2-2
“Opening the Model” on page 2-2
“Overview of the Model” on page 2-3
“Quadrature Amplitude Modulation” on page 2-4
“Run a Simulation” on page 2-5
“Display the Error Rate” on page 2-6
“Set Block Parameters” on page 2-7
“Display a Phase Noise Plot” on page 2-9

Section Overview

This section describes an example model of a communications system. The model displays a scatter plot of a signal with added noise. The purpose of this section is to familiarize you with the basics of Simulink models and how they function.

Opening the Model

To open the model, first start MATLAB. In the MATLAB Command Window, enter `doc_comphasenoise` at the prompt. This opens the model in a new window, as shown in the following figure.



Overview of the Model

The model shown in the preceding section, “Opening the Model” on page 2-2, 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 Random Integer Generator block, labeled Random Integer, generates a signal consisting of a sequence of random integers between zero and 255
- The Rectangular QAM Modulator Baseband block, to the right of the Random Integer 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.

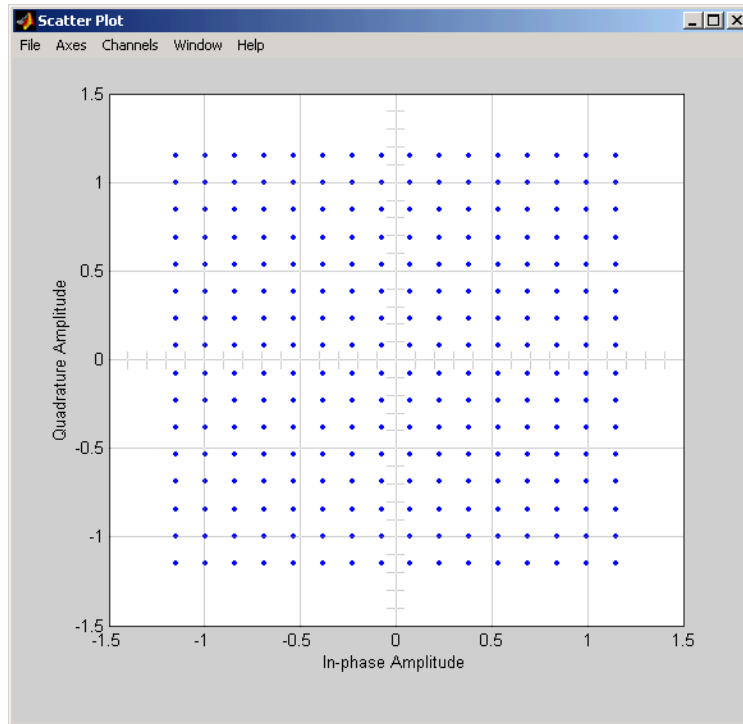
In addition, the following blocks in the model help you interpret the simulation:

- 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 Display block, at the far right of the model window, displays the symbol error rate (SER), the total number of errors, and the total number of symbols processed during the simulation.

All these blocks are included in Communications System 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 is shown in the following figure.

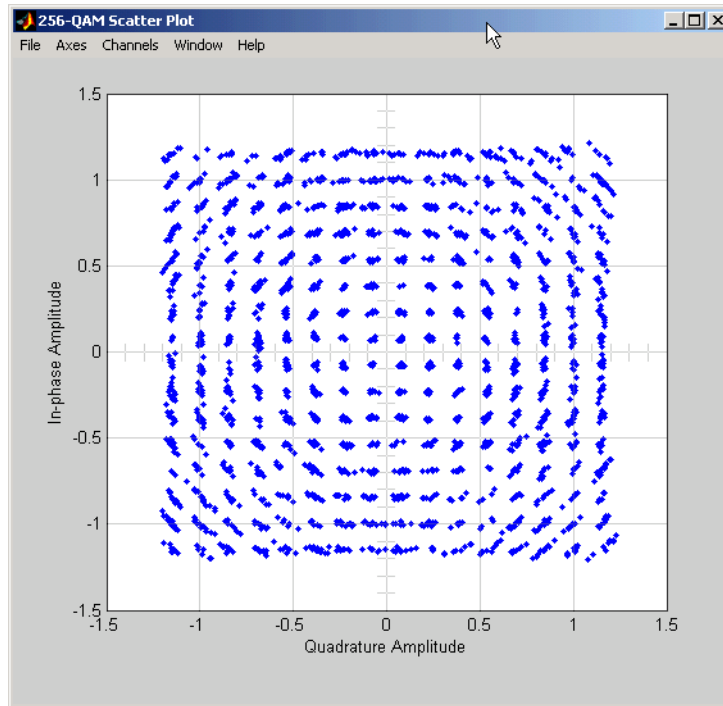


Constellation for 256-ary QAM

Run a Simulation

To run a simulation, click on the **Run** button at the top of the model window. The simulation stops automatically at the **Stop time**, which is specified in the **Configuration Parameters** dialog box. You can stop the simulation at any time by selecting **Stop** from the **Simulation** menu at the top of the model window (or, on Microsoft Windows, by clicking the **Stop** button on the toolbar).

When you run the model, a new window appears, displaying a scatter plot of the modulated signal with added noise, as shown in the following 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 pattern 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.

0.007701	SER
774	Total Errors
1.005e+005	Total Symbols

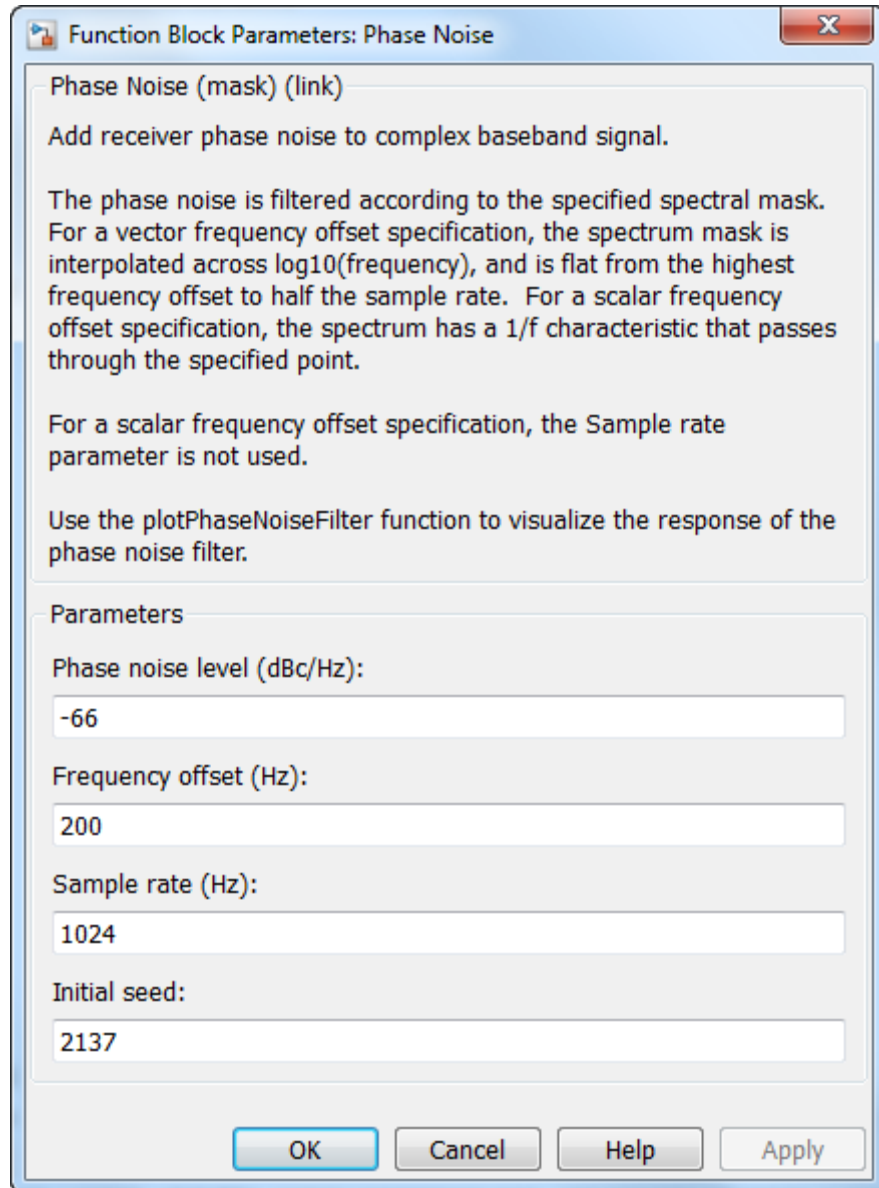
Error Rate Display

The block displays the output as follows:

- The first entry is the symbol error rate (SER).
- The second entry is the total number of errors.
- The third entry is the total number of comparisons made. The notation $1e+004$ is shorthand for 10^4 .

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 **Es/No** parameter field. This changes the signal to noise ratio, in dB. Decreasing the value of **Es/No** increases the noise level.

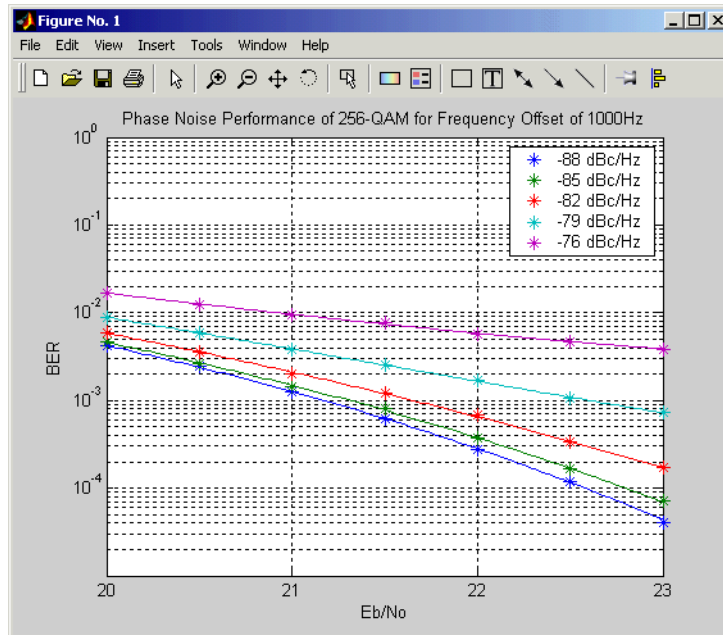
You can experiment with the model by changing these or other parameters and then running a simulation. For example,

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

This removes nearly all noise from the model. 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

Double-click the block labeled “Display Figure” at the bottom left of the model window. This displays a plot showing the results of multiple simulations.



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 **E_s/N_0** parameters.

16-QAM with MATLAB Functions

In this section...

“Introduction” on page 2-11

“Modulate a Random Signal” on page 2-11

“Plot Signal Constellations” on page 2-18

“Pulse Shaping Using a Raised Cosine Filter” on page 2-23

“Error Correction using a Convolutional Code” on page 2-30

Introduction

Communications System Toolbox software implements a variety of communications-related tasks. Many of the functions in the toolbox perform computations associated with a particular component of a communication system, such as a demodulator or equalizer. Other functions are designed for visualization or analysis. The toolbox includes both functions and System objects. Which to use is described in “When to Use System Objects Instead of MATLAB Functions” on page 4-5.

This section builds an example step-by-step to give you a first look at the Communications System Toolbox software. This section also shows how Communications System 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 System 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
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.

Note The sampling times associated with the bits do not appear explicitly, and MATLAB has no inherent notion of time. For the purpose of this example, knowing only the values in the data stream is enough to solve the problem.

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

Define parameters.

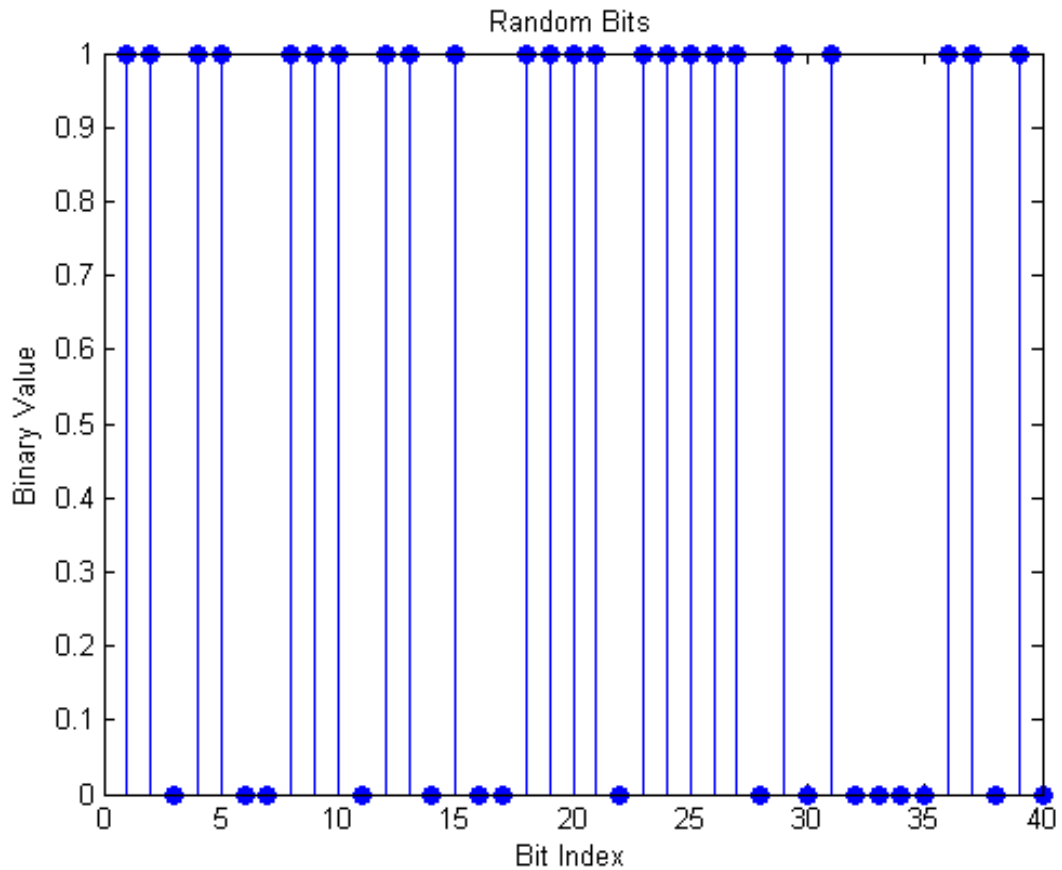
```
M = 16; % Size of signal constellation
k = log2(M); % Number of bits per symbol
n = 30000; % Number of bits to process
numSamplesPerSymbol = 1; % Oversampling factor
```

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

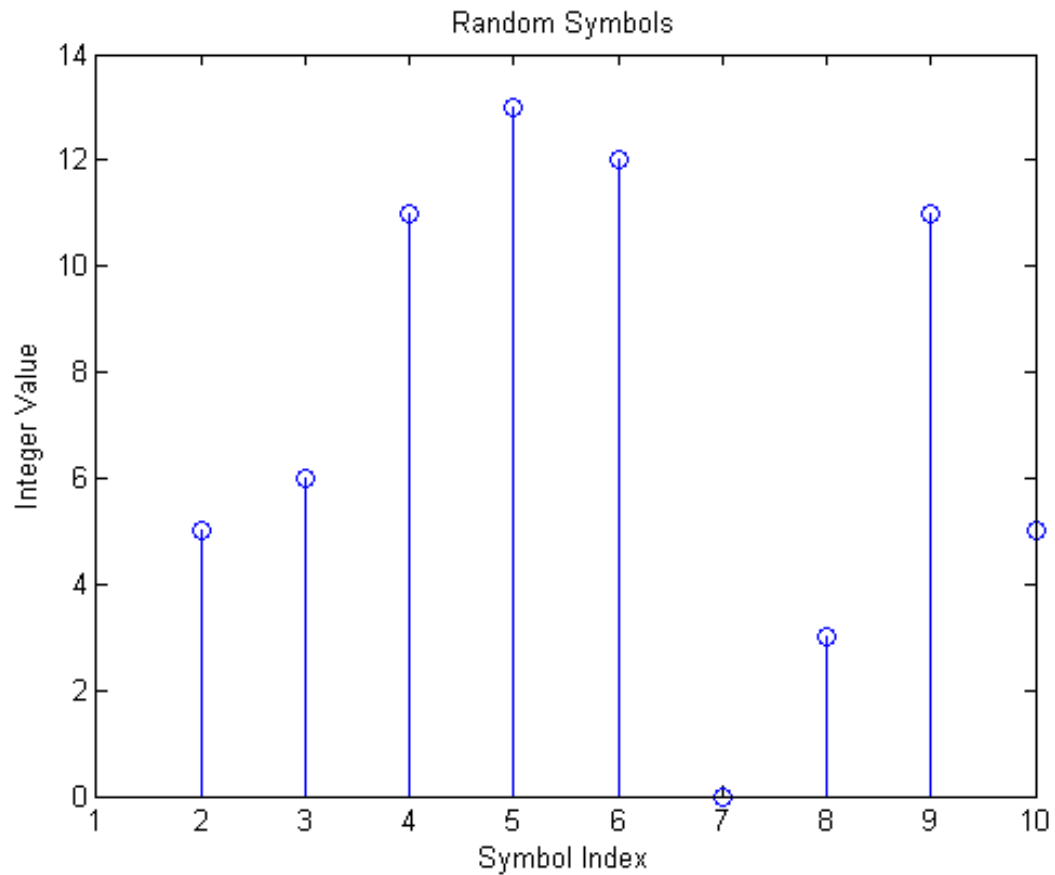
The `qammod` function implements a rectangular, M-ary QAM modulator, M being 16 in this example. The default configuration is such that the object receives integers between 0 and 15 rather than 4-tuples of bits. In this example, we preprocess the binary data stream `dataIn` before using the `qammod` 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)/4, 4); % Reshape data into bi
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. Recall that `M` is 16, the alphabet size.

Apply modulation.

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

The result is a complex column vector 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 2-23. 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.

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

Pass the signal through the AWGN channel.

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

Create a Constellation Diagram

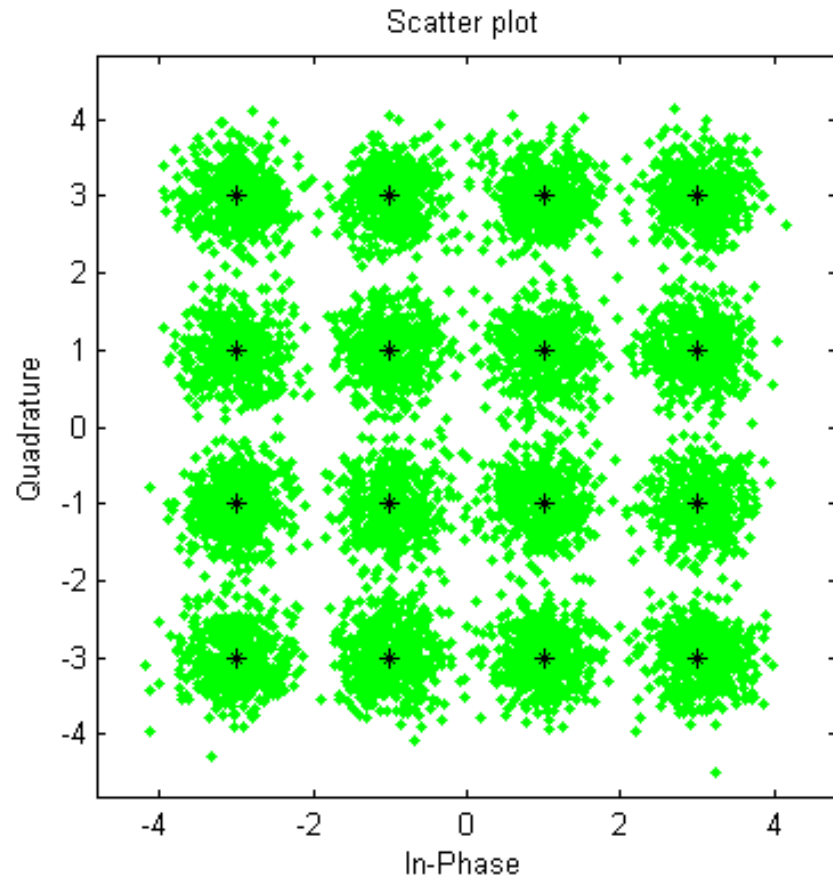
The `scatterplot` function is used to display the in-phase and quadrature components of the modulated signal, `dataMod`, and its received, noisy version, `receivedSignal`. By looking at the resultant diagram, the effects of AWGN are readily observable.

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

```
sPlotFig = scatterplot(receivedSignal, 1, 0, 'g.');
```

```
hold on
```

```
scatterplot(dataMod, 1, 0, 'k*', sPlotFig)
```



Demodulate 16-QAM

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

Demodulate the received signal using the `qamdemod` function.

```
dataSymbolsOut = qamdemod(receivedSignal, 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- $N_{\text{bits/sym}}$, where N_{sym} is the total number of QAM symbols and $N_{\text{bits/sym}}$ 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.

Reverse the bit-to-symbol mapping performed earlier.

```
dataOutMatrix = de2bi(dataSymbolsOut,k);  
dataOut = dataOutMatrix(:); % 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 stream, `dataOut`.

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

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

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

Plot Signal Constellations

The example in the previous section, “Modulate a Random Signal” on page 2-11, 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.

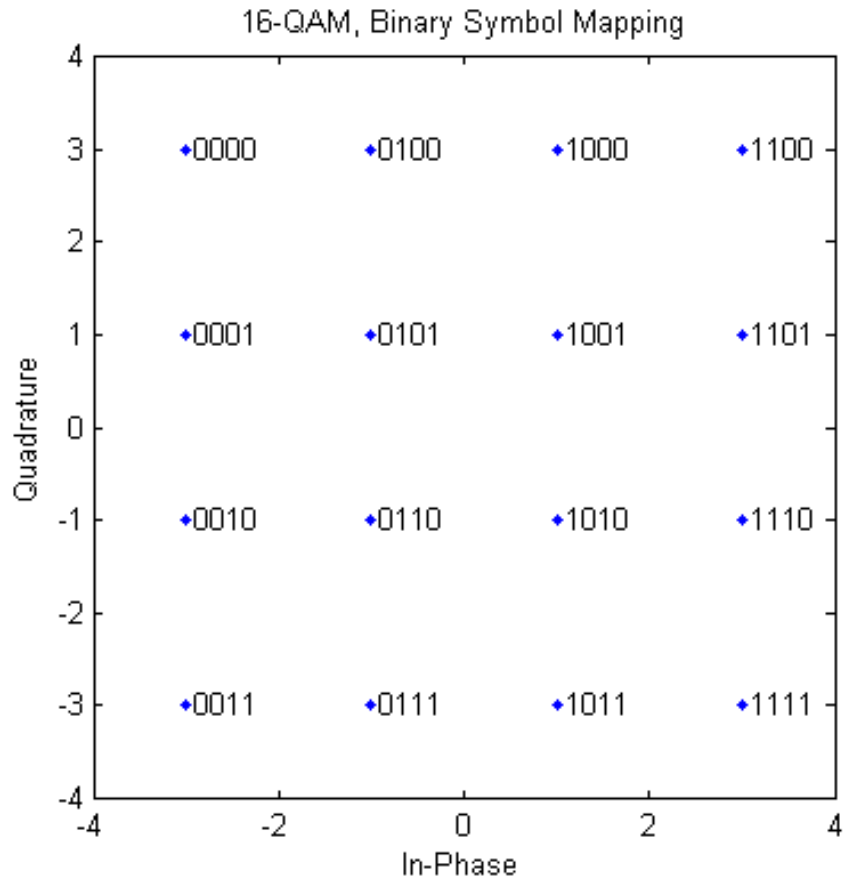
Binary Symbol Mapping for 16-QAM Constellation

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, 0); % 16-QAM output, phase offset = 0
```

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.

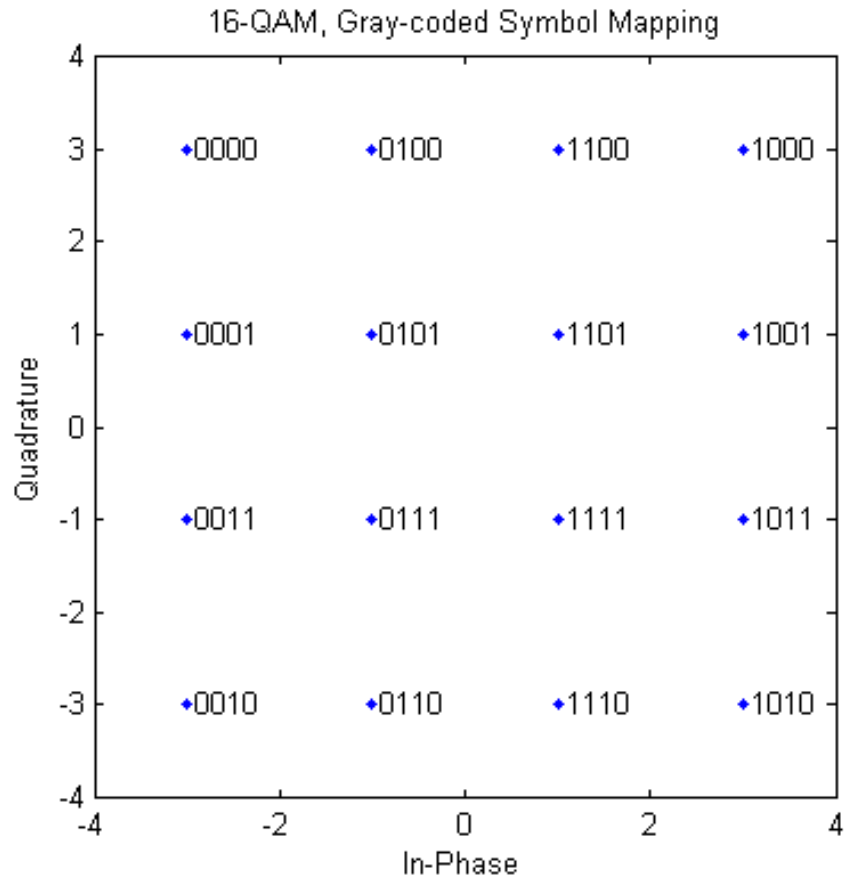
```
y2 = qammod(x, 16, 0, 'gray'); % 16-QAM output, phase offset = 0, Gray-coded
```

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.

Pulse Shaping Using a Raised Cosine Filter

The “Modulate a Random Signal” on page 2-11 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 2-30, 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.

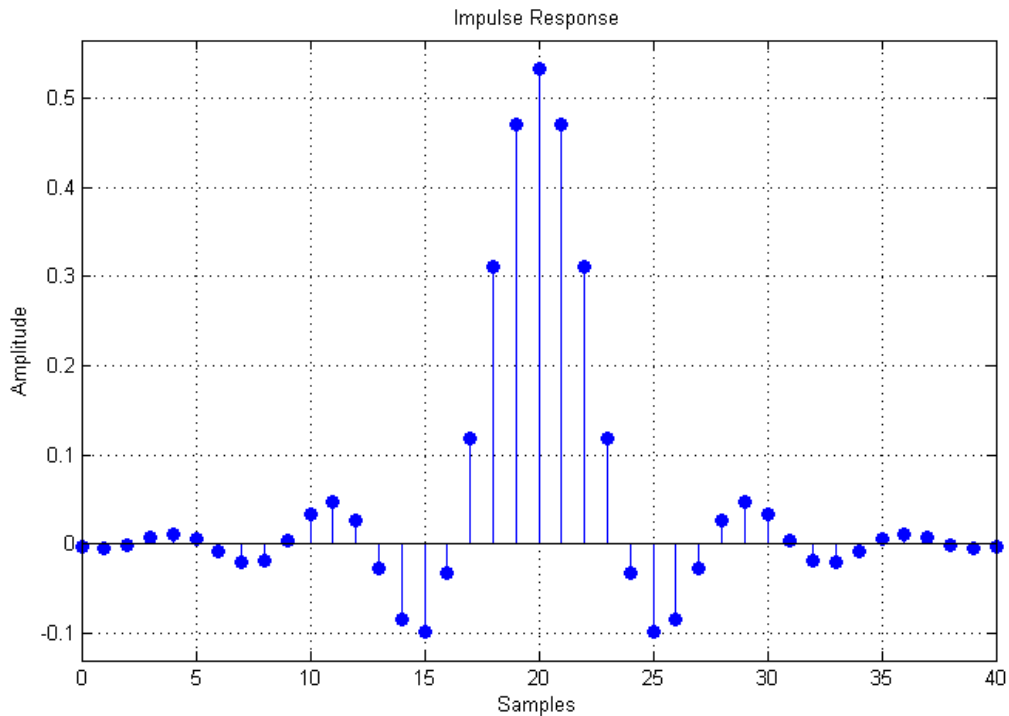
```
span = 10; % Filter span in symbols
rolloff = 0.25; % Roloff factor of filter
```

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 bi  
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); % Downs
rxFiltSignal = rxFiltSignal(span+1:end-span); % Accou
```

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.

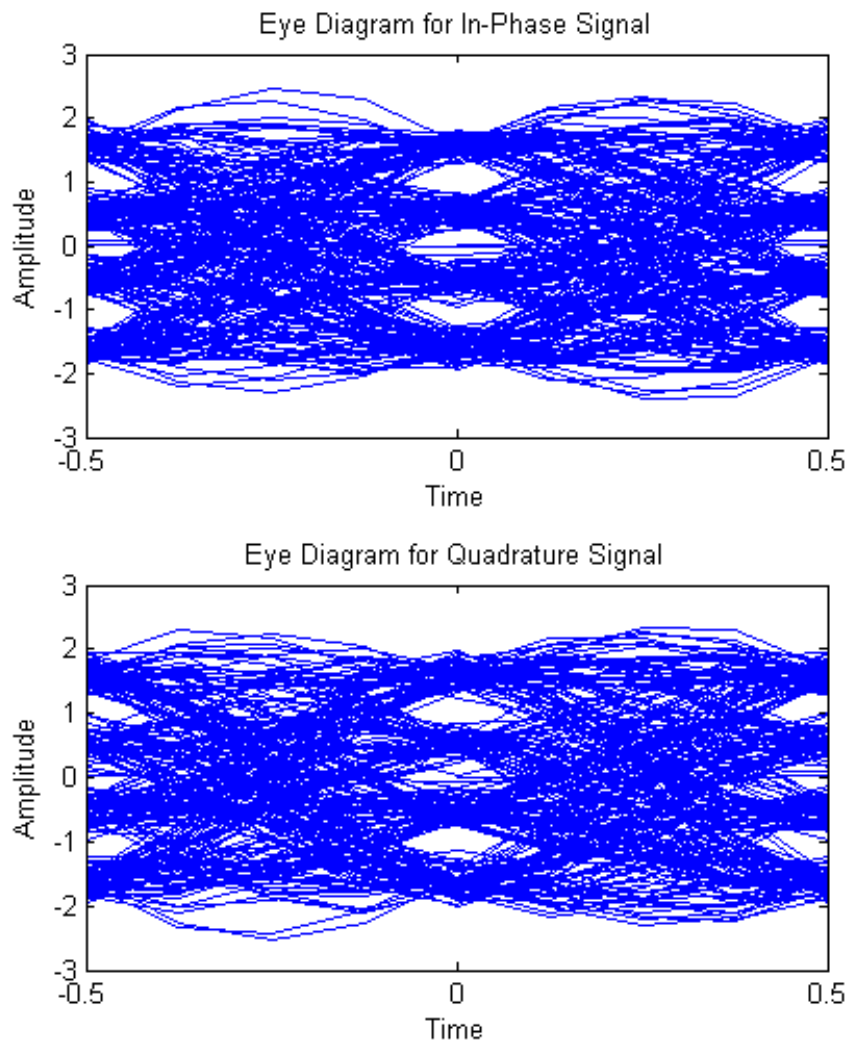
```
[numErrors, ber] = biterr(dataIn, dataOut);  
fprintf('\nThe bit error rate = %5.2e, based on %d errors\n', ...  
        ber, numErrors)
```

```
The bit error rate = 2.42e-03, based on 727 errors
```

Visualization of Filter Effects

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

```
eyediagram(txSignal(1:2000), numSamplesPerSymbol*2);
```

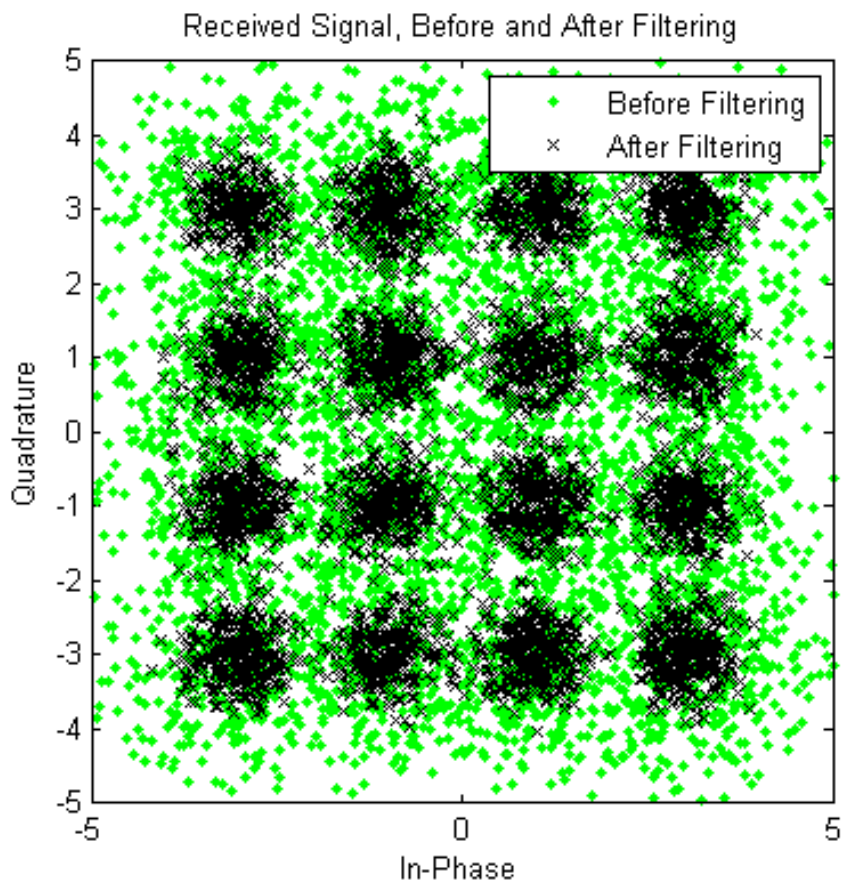



The `eyediagram` function creates an eye diagram for part of the filtered noiseless signal. This diagram illustrates the effect of the pulse shaping. Note

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.

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



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 2-23 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 2-23 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
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 2-23 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_o , the number of samples per symbol, and the code rate. Converting from E_b/N_o 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 $k*\text{codeRate}$ 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); % Downs  
rxFiltSignal = rxFiltSignal(span+1:end-span); % Accou
```

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.

```
traceBack = 16;                               % Traceback length
numCodeWords = floor(length(codedDataOut)*2/3); % Number of code words
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 $2 \times \text{traceBack}$ bits. As a result, the first $2 \times \text{traceBack}$ bits in the decoded vector, `dataOut`, are zeros. When computing the bit error rate, the first $2 \times \text{traceBack}$ bits in `dataOut` and the last $2 \times \text{traceBack}$ bits in the original vector, `dataIn`, are discarded. Without delay compensation, the BER computation is meaningless.

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

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

```
The bit error rate = 6.90e-04, based on 69 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 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 2-36

“Introduce convolutional coding and hard-decision Viterbi decoding” on page 2-41

“Improve results using soft-decision decoding” on page 2-46

“Use turbo coding to improve BER performance” on page 2-51

“Apply a Rayleigh channel model” on page 2-54

“Use OFDM-based equalization to correct multipath fading” on page 2-59

“Use multiple antennas to further improve system performance” on page 2-62

“Accelerate the simulation using MATLAB Coder” on page 2-66

This example illustrates a design workflow that represents the iterative steps for creating a wireless communications system with the Communications System Toolbox. Because Communications System 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 (E_n/N_0), 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 System 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**.

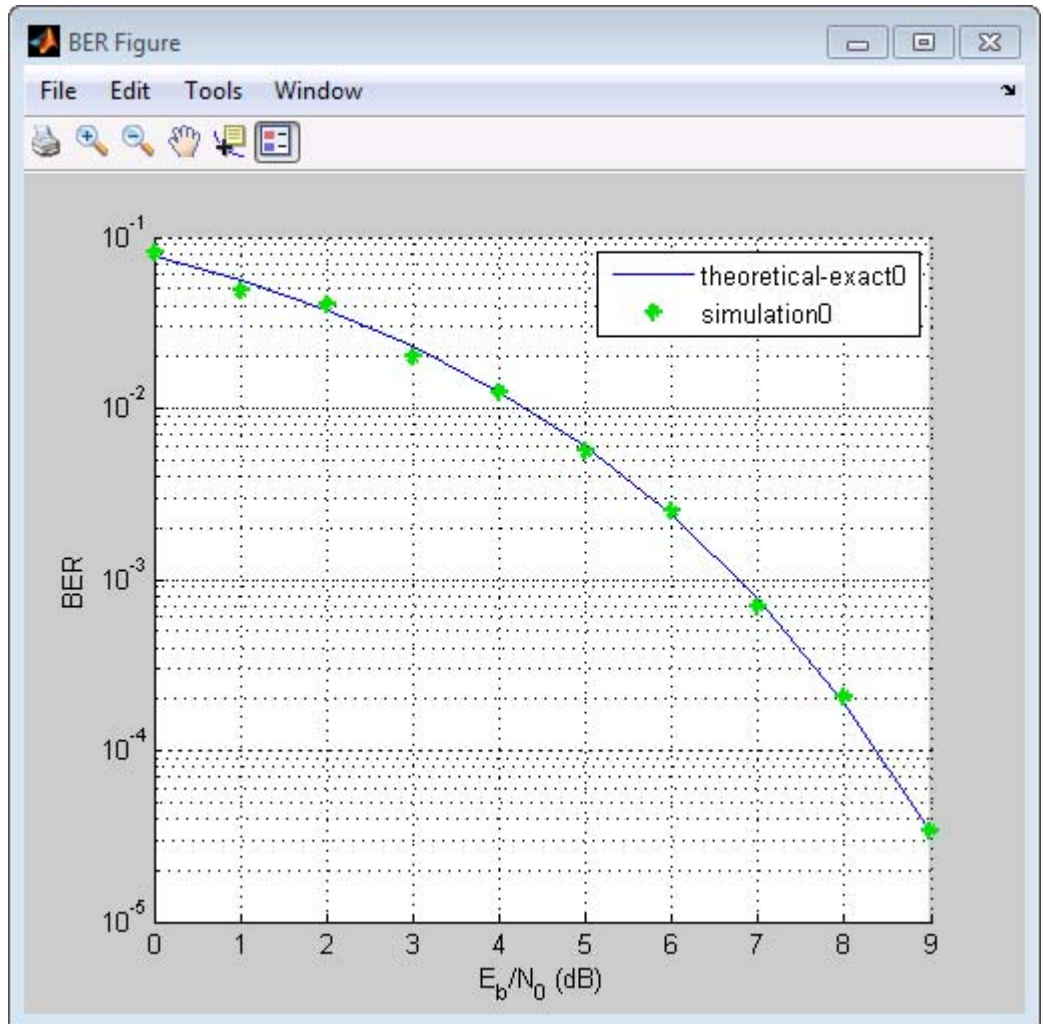
The **Number of bits** is also a stop criteria for the simulation. The simulation stops when it transmits the number of bits you specify for this parameter. In this example, the simulation either stops when it transmits 10 million bits or when it detects 200 errors.

- 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 E_b/N_0 .

- 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**.

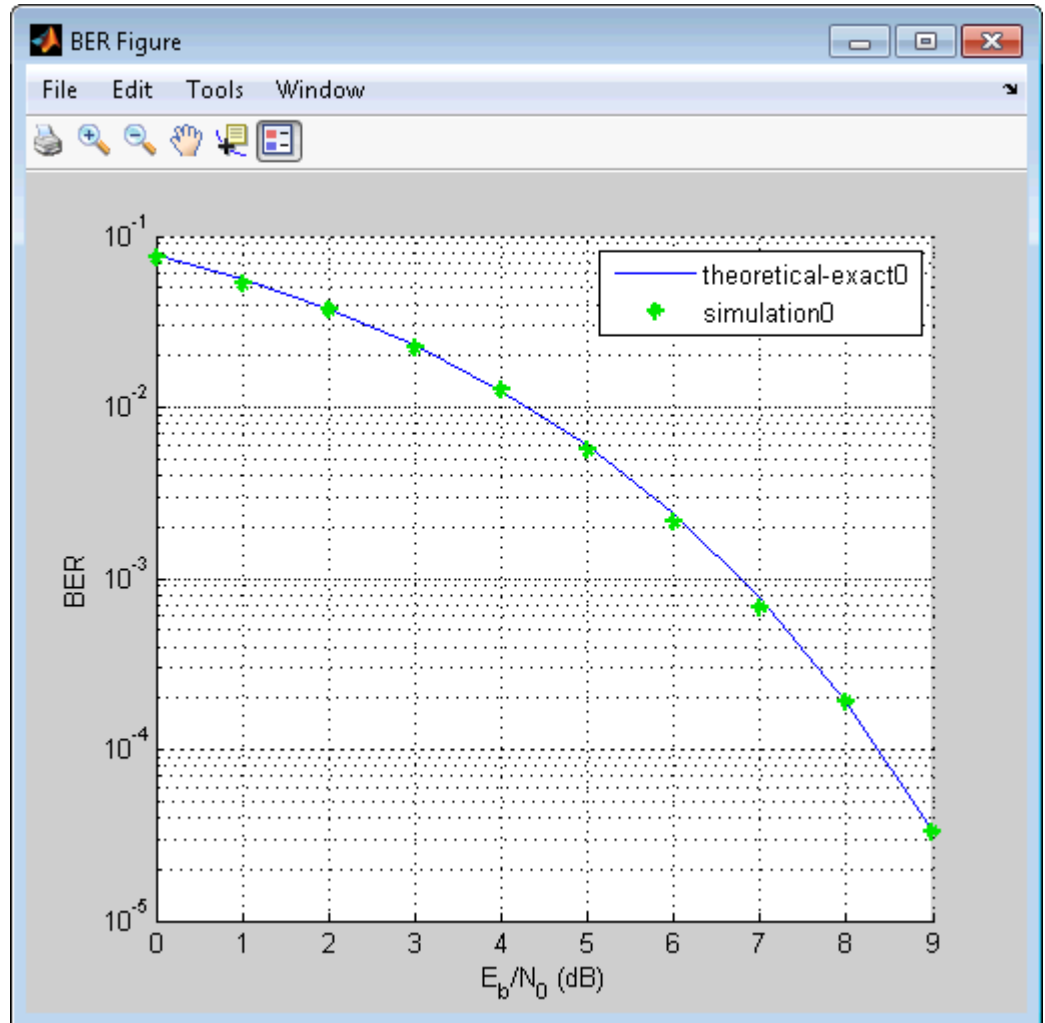
The **Number of errors** is one of the stop criteria for the simulation. The simulation stops when it reaches either the **Number of errors** or the **Number of bits**.

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.

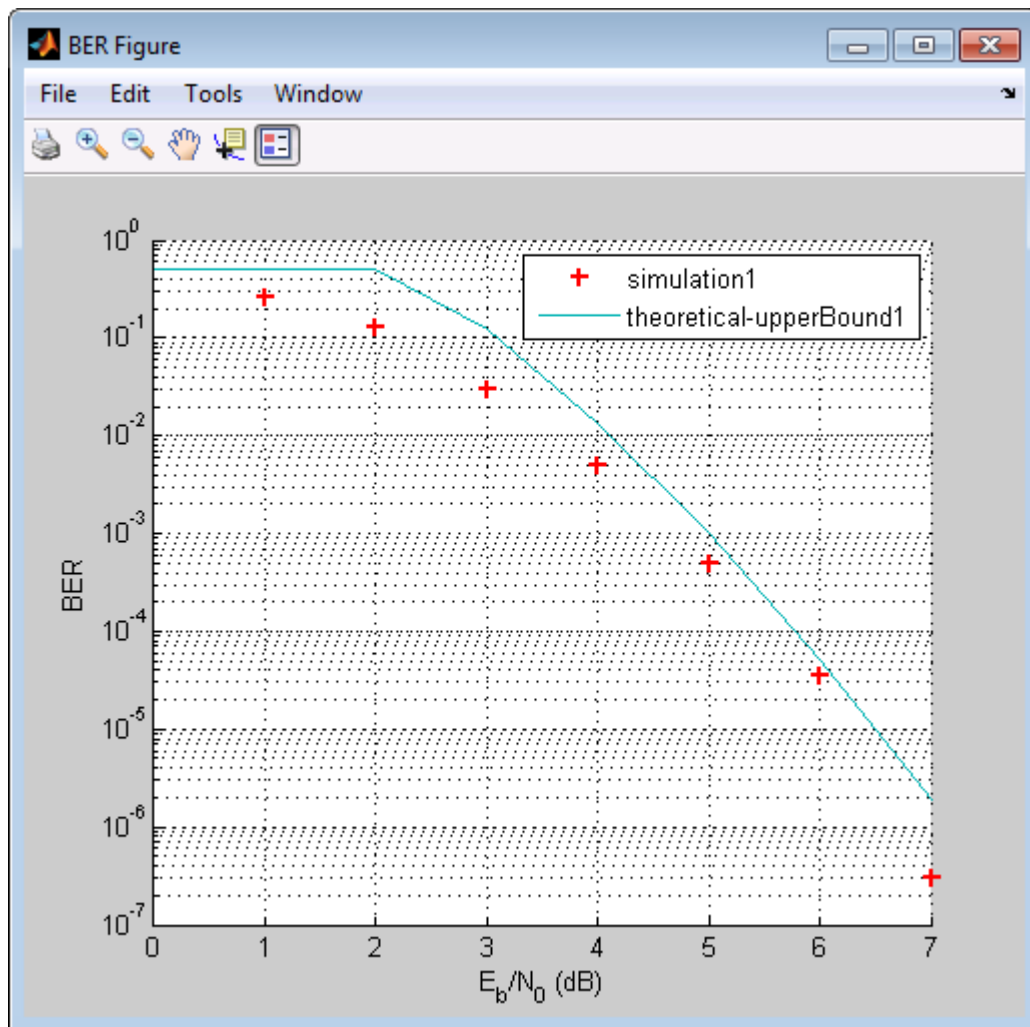
- 7 Click **Plot**.

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**.

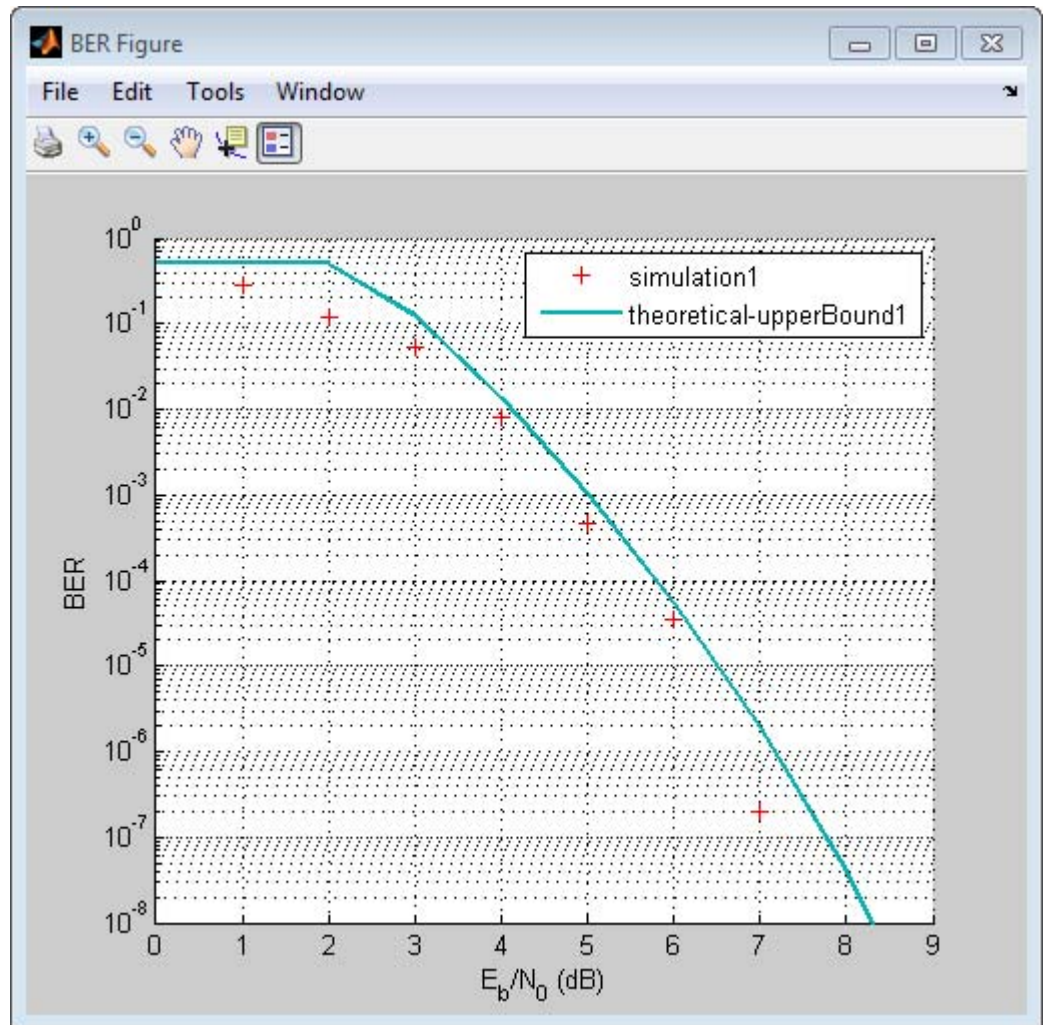
9 Enter 200 for the **Number of errors**.

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**.

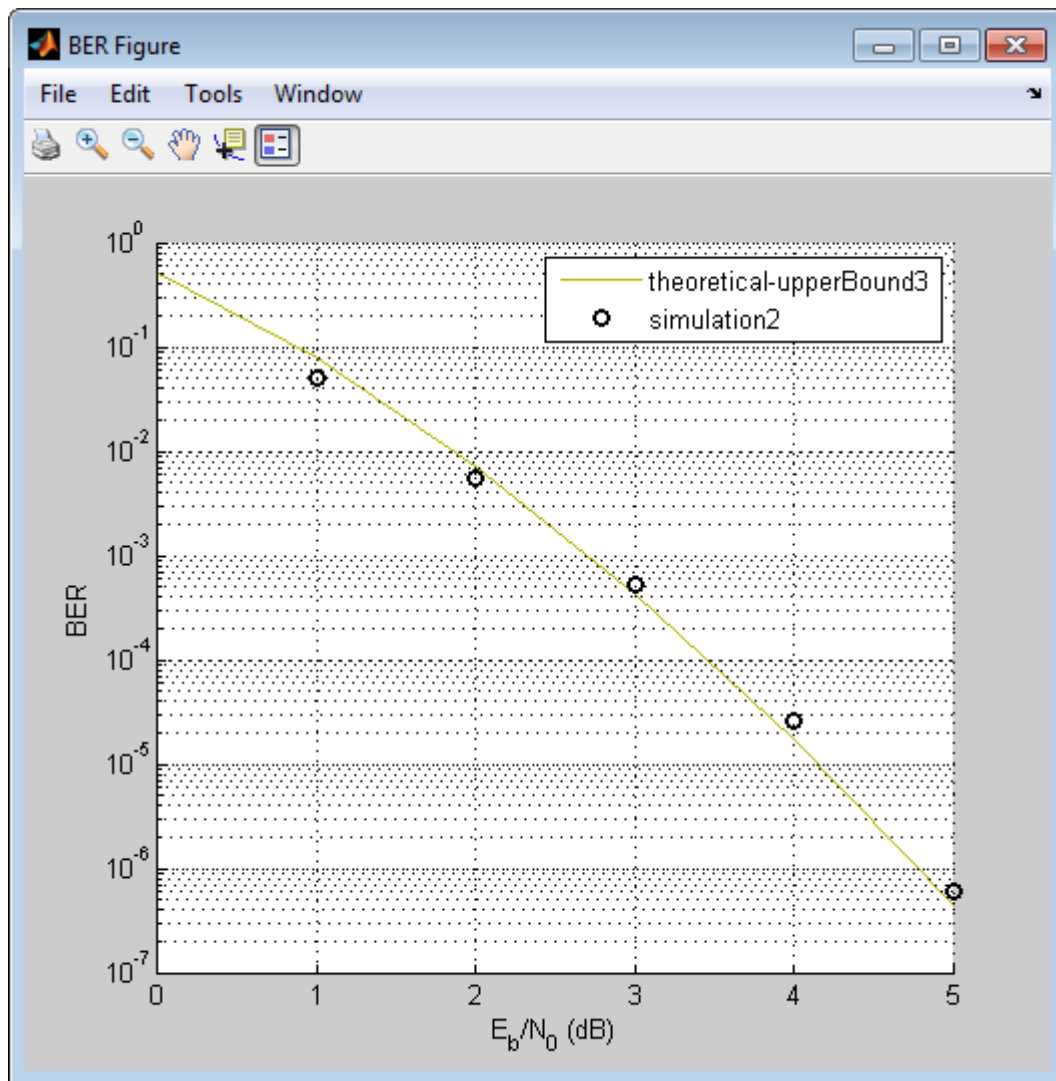
This example uses soft-decision Viterbi decoding. The demodulator maps the received signal to log likelihood ratios, improving BER performance results.

- 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.

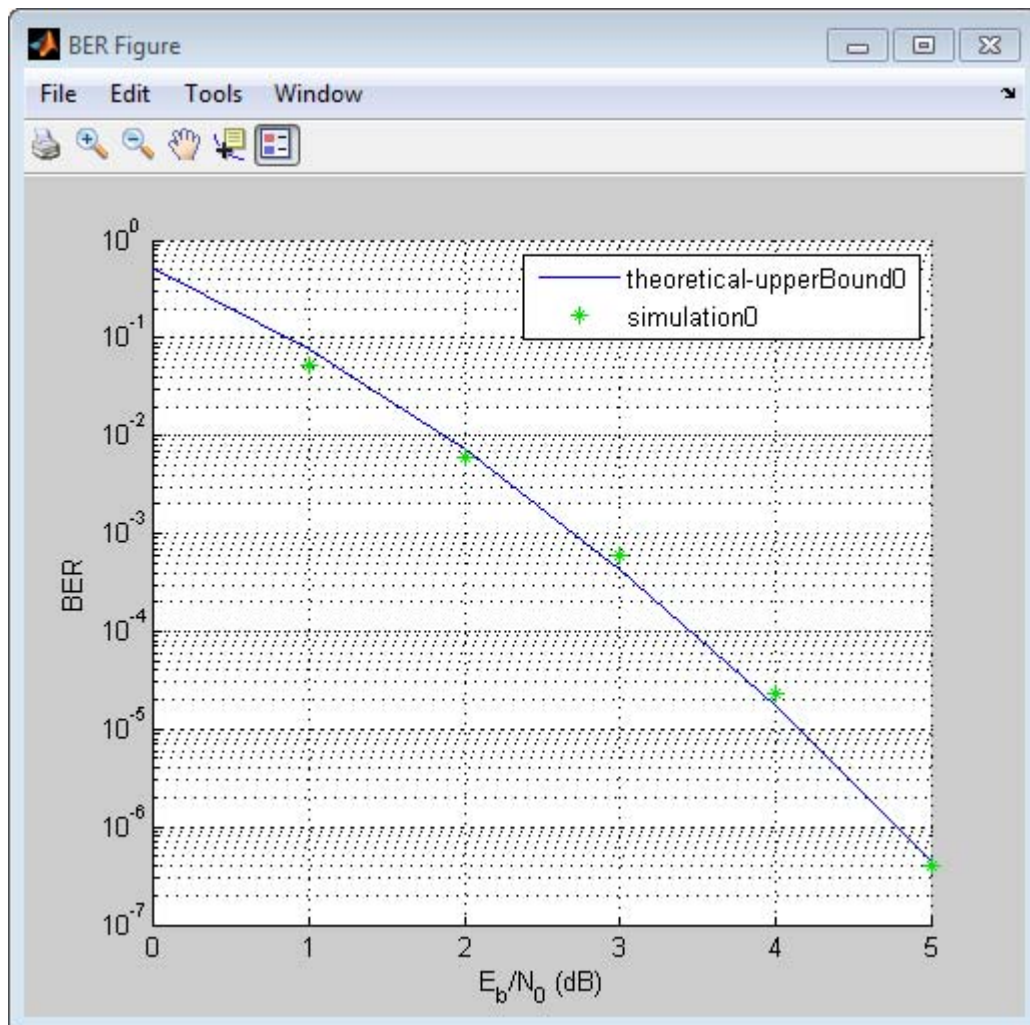
- 2** Clear the **Plot** check boxes for the two plots BERTool generated in the previous step.
- 3** Click **Theoretical**.
- 4** Enter 0:5 for the **EbNo range**.
- 5** Select **Soft** for the **Decision method**.

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.

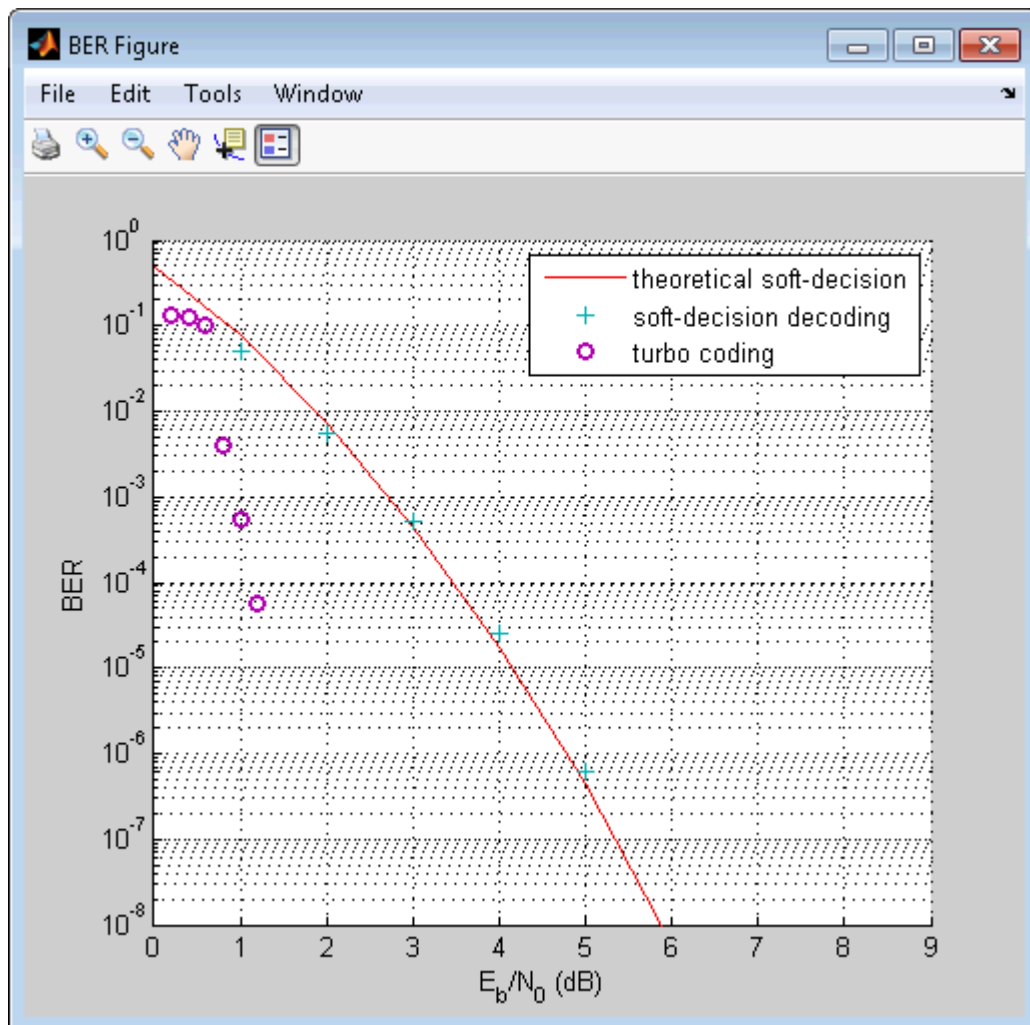
Use turbo coding to improve BER performance

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.

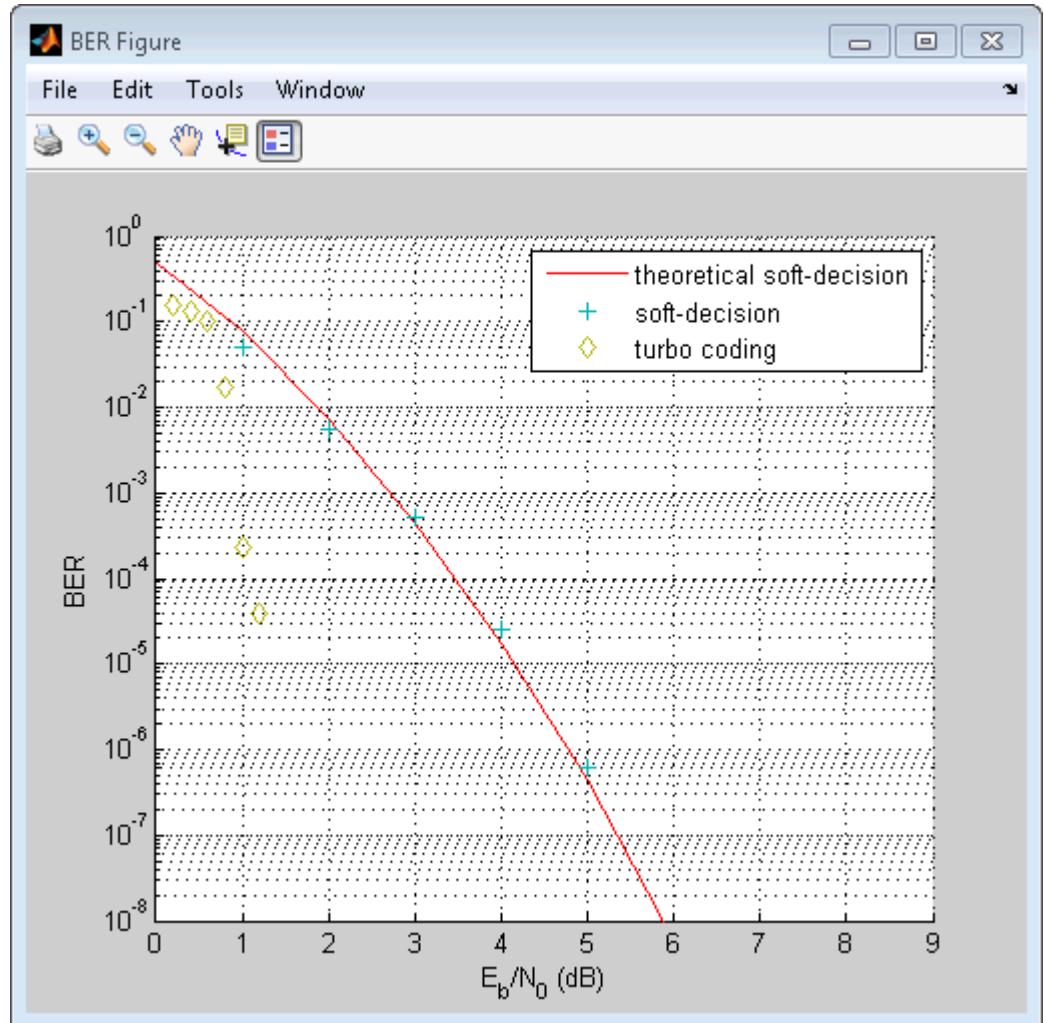


In Simulink

- 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

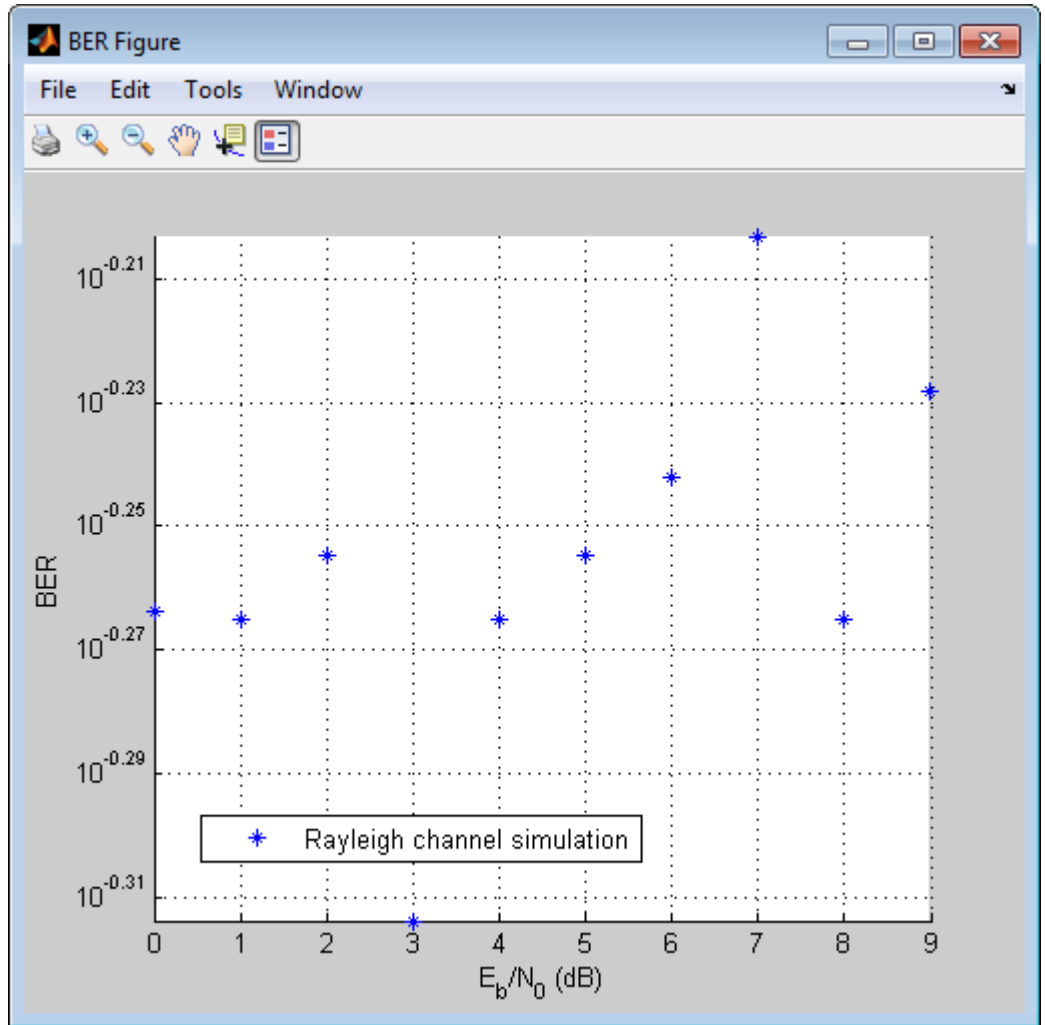
The previous design iterations model narrowband communications systems that can be adequately represented using an AWGN channel. However, high data rate communications systems require a wideband channel. Wideband communications channels are highly susceptible to the effects of multipath propagation, 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.



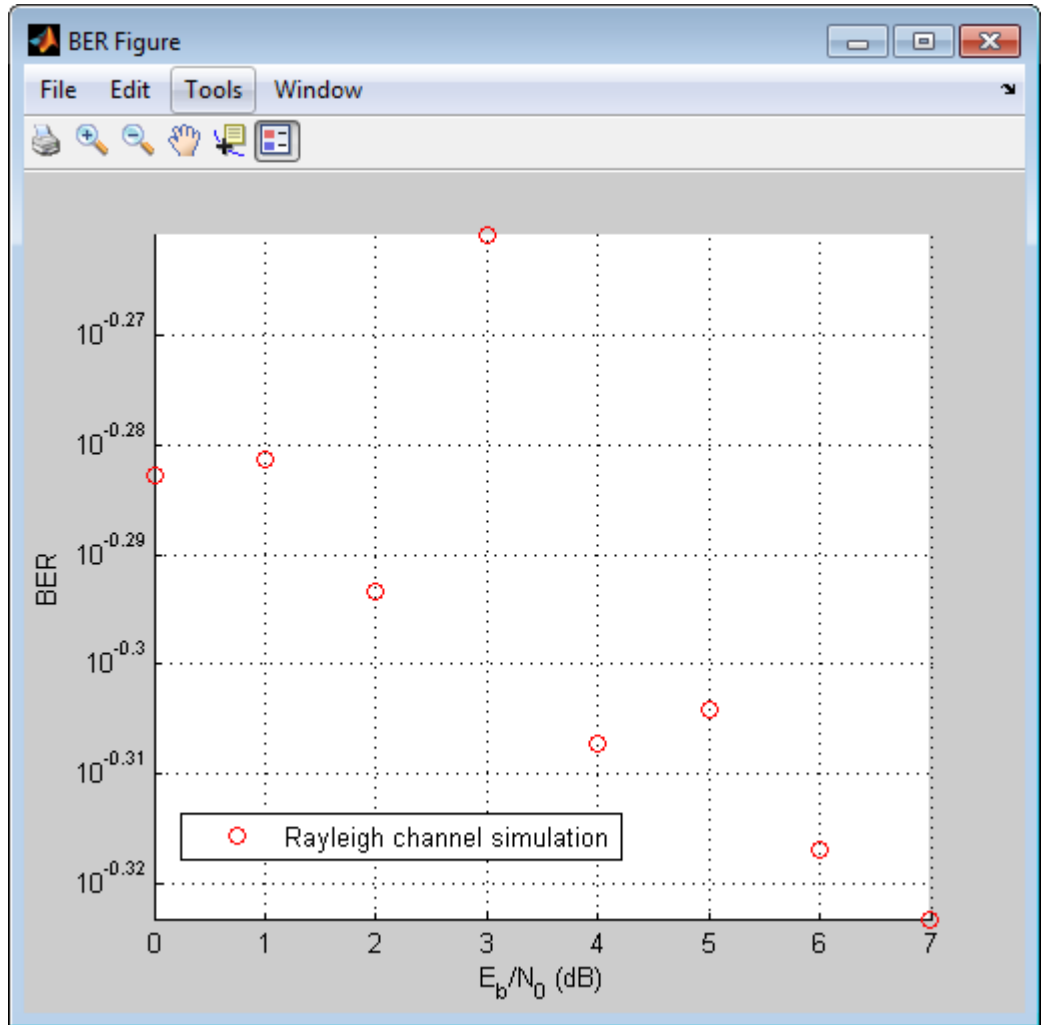
In the presence of multipath fading, the BER performance reduces to that of a binary channel with a consistent value of one-half. To correct the effect of multipath fading, you must add equalization to the communications system.

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.

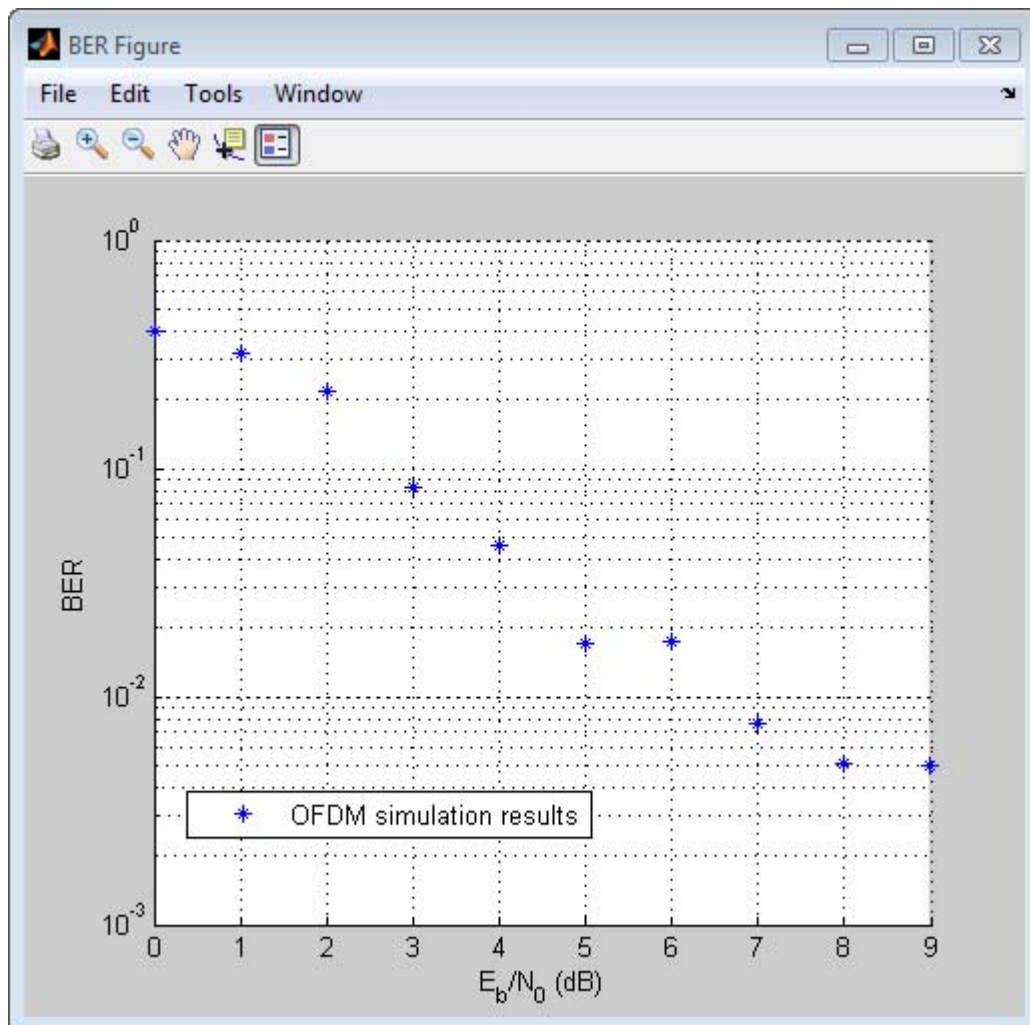
Use OFDM-based equalization to correct multipath fading

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.

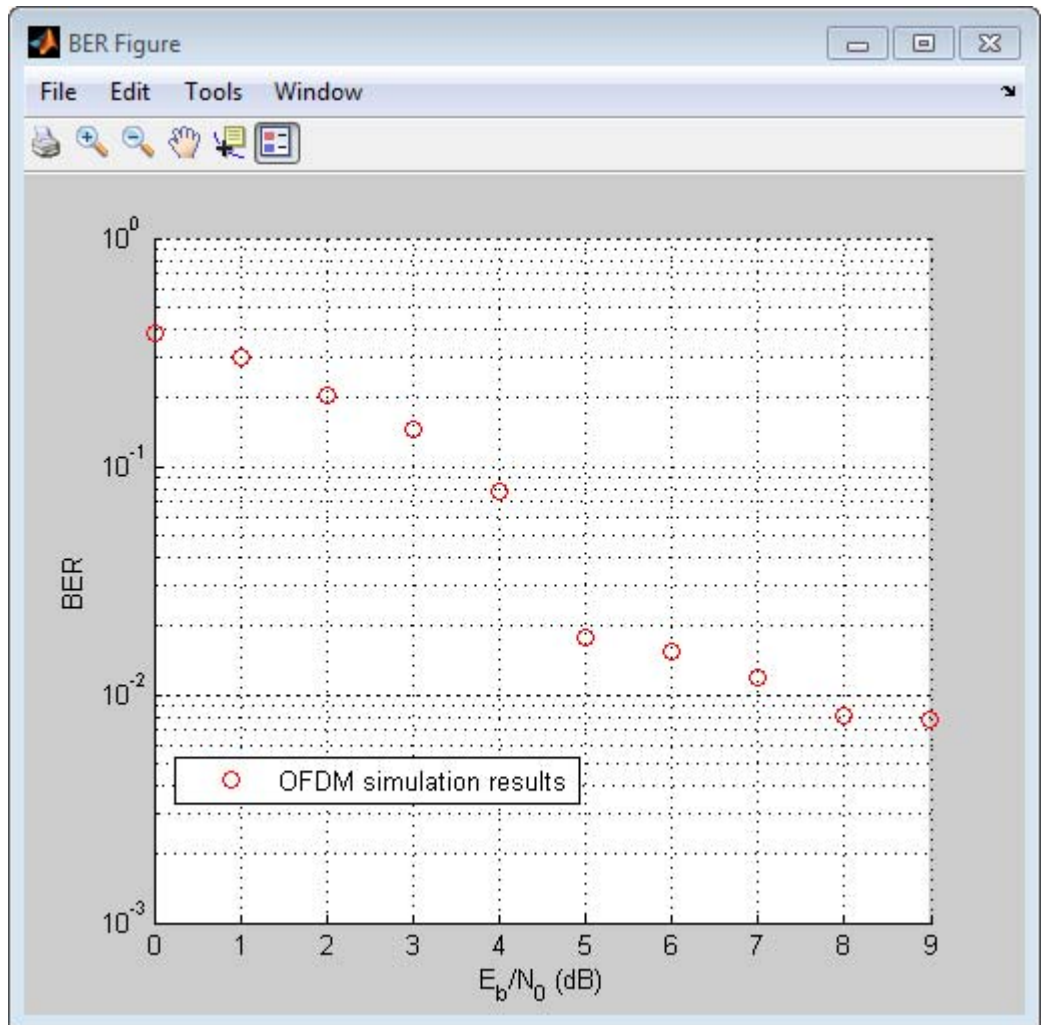


In Simulink

- 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

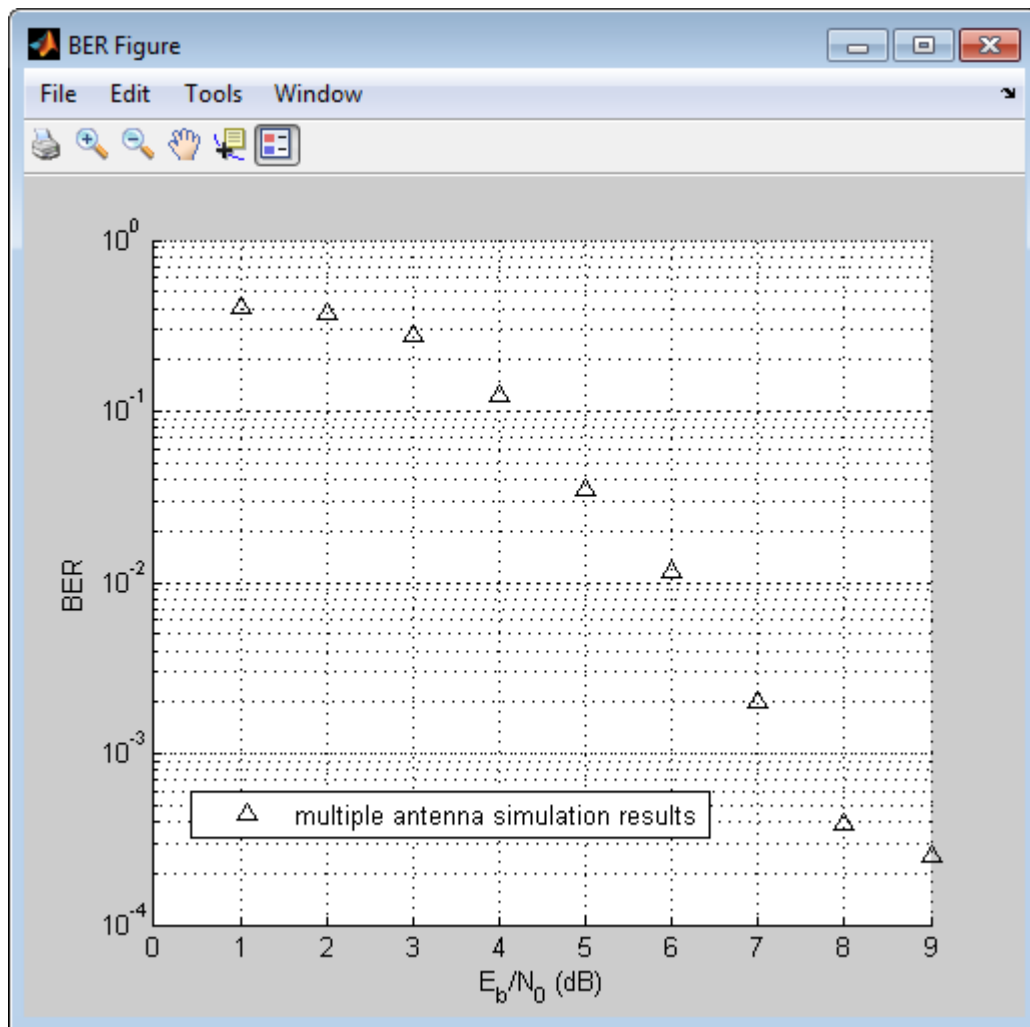
Simultaneously transmitting copies of a signal using multiple antennas can significantly increase the likelihood that the receiver correctly recovers the transmitted signal. This phenomenon is known as transmit diversity.

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**.

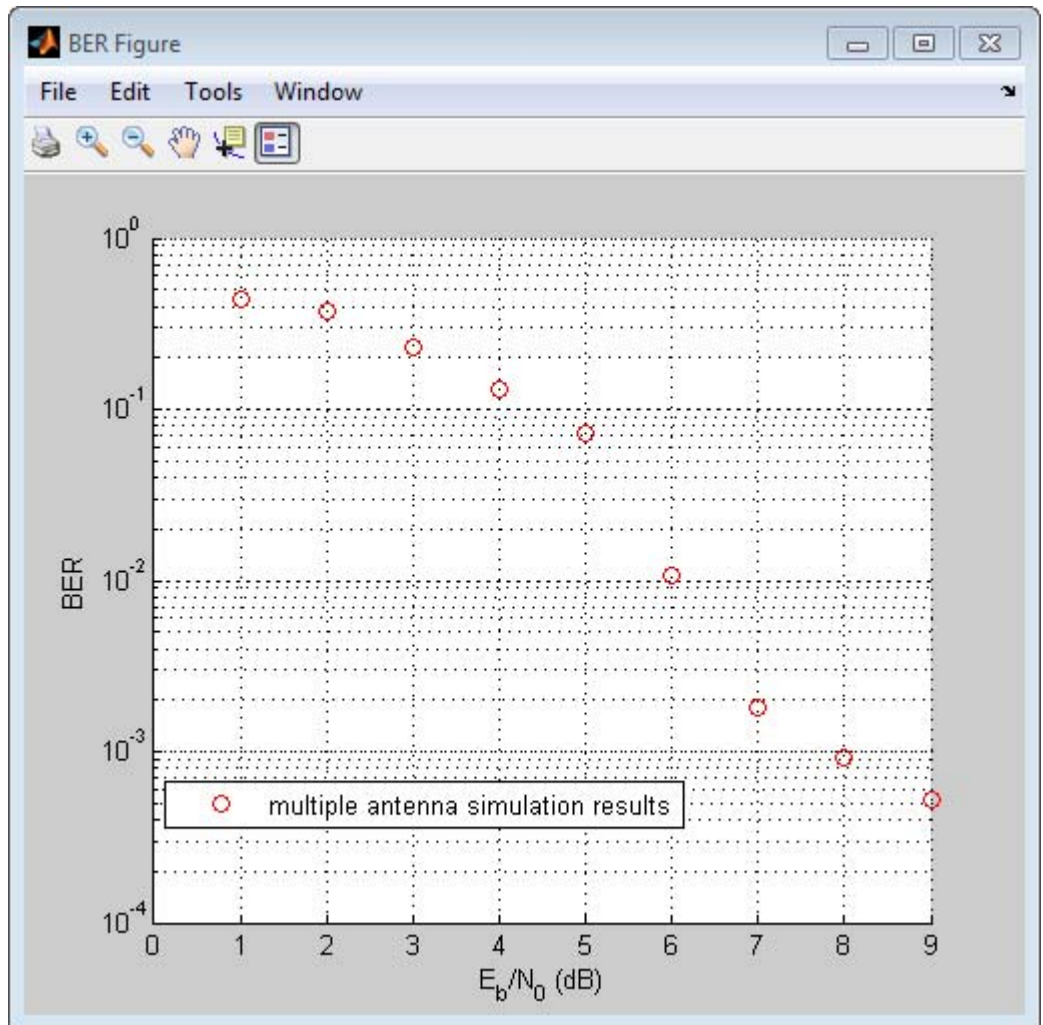
BERTool runs the simulation and generates simulated points along the BER curve. Compare the simulation BER curve with the theoretical BER curve.



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
```

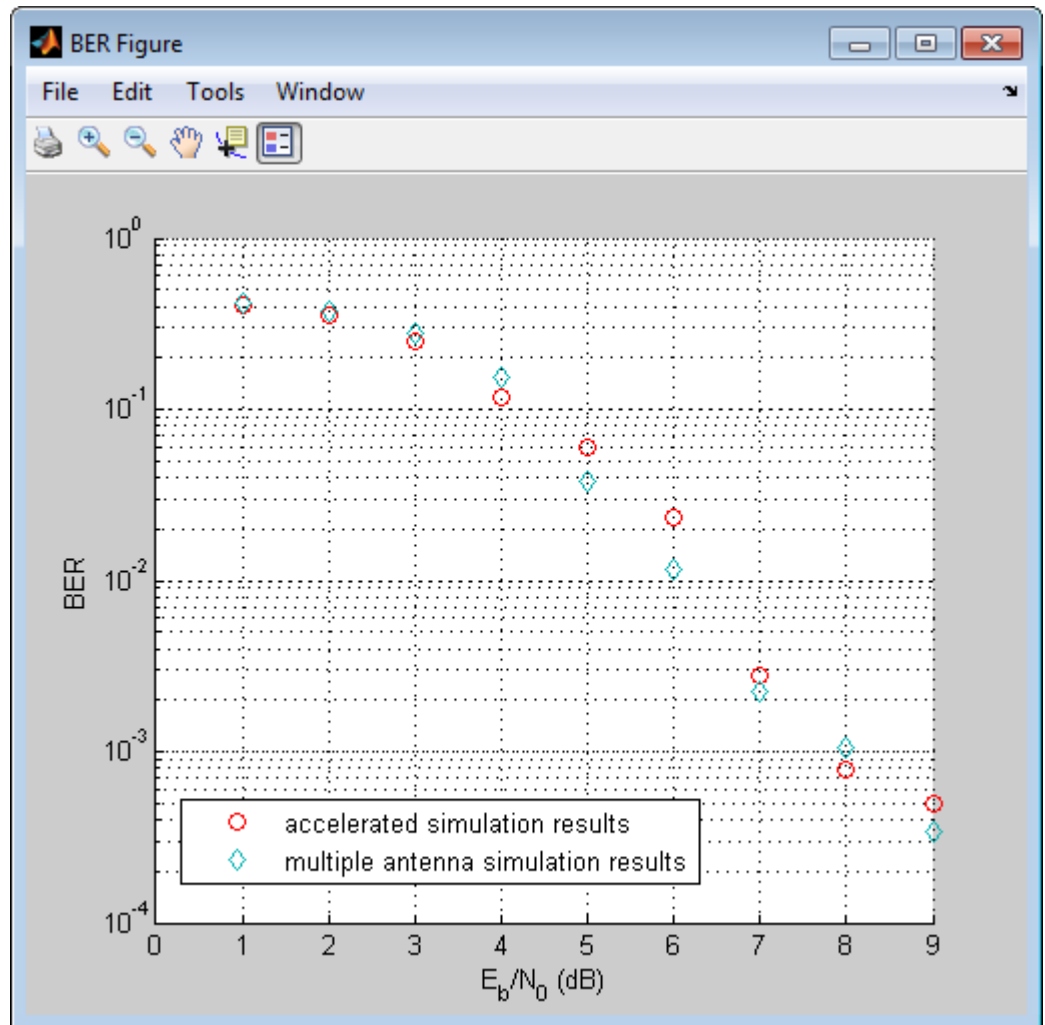
- 5 The file extension of the MATLAB executable file that gets generated depends upon your operating system. For example, on 64-bit Windows® the file extension will be `.mexw64`, and the full file name will be `doc_design_iteration_viterbi_rayleigh_OFDM_MIMO_m_mex.mexw64`.

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 to complete.



QPSK and OFDM with MATLAB System Objects

In this section...

“Simulate a basic communications system” on page 2-71

“Introduce convolutional coding and hard-decision Viterbi decoding” on page 2-74

“Improve results using soft-decision decoding” on page 2-76

“Use turbo coding to improve BER performance” on page 2-79

“Apply a Rayleigh channel model” on page 2-80

“Use OFDM-based equalization to correct multipath fading” on page 2-83

“Use multiple antennas to further improve system performance” on page 2-84

“Accelerate the simulation using MATLAB Coder” on page 2-86

This example illustrates a design workflow that represents the iterative steps for creating a wireless communications system with the Communications System Toolbox. This example showcases a design path using MATLAB System objects.

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 bit energy-to-noise density ratio (E_b/N_0), 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 E_b/N_0 , 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.

- 1 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 System Toolbox: `comm.PSKModulator`, `comm.AWGNChannel`, `comm.PSKDemodulator`, and `comm.ErrorRate`. For each E_b/N_0 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 functions that this example uses have a similar structure.

- 2 Type `bertool` at the MATLAB command line to open the Bit Error Rate Analysis Tool.
- 3 When the BERTool application appears, click the **Theoretical** tab.

The first plot that you will generate is a theoretical curve.

- 4 Enter `0:9` for the **EbNo range**.

E_b/N_0 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.

5 Select 4 for **Modulation order**.

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

6 Click **Plot**.

The BERTool application generates the theoretical BER curve.

7 Click the **Monte Carlo** tab.

Monte Carlo techniques use random sampling to compute data. Therefore, the plot for the second simulation uses random sampling.

8 Enter 0:9 for the **EbNo range**.

9 Enter ber for the **BER variable name**.

10 Enter 200 for the **Number of errors**.

The **Number of errors** is one of the stop criteria for the simulation.

11 Enter 1e7 for the **Number of bits**.

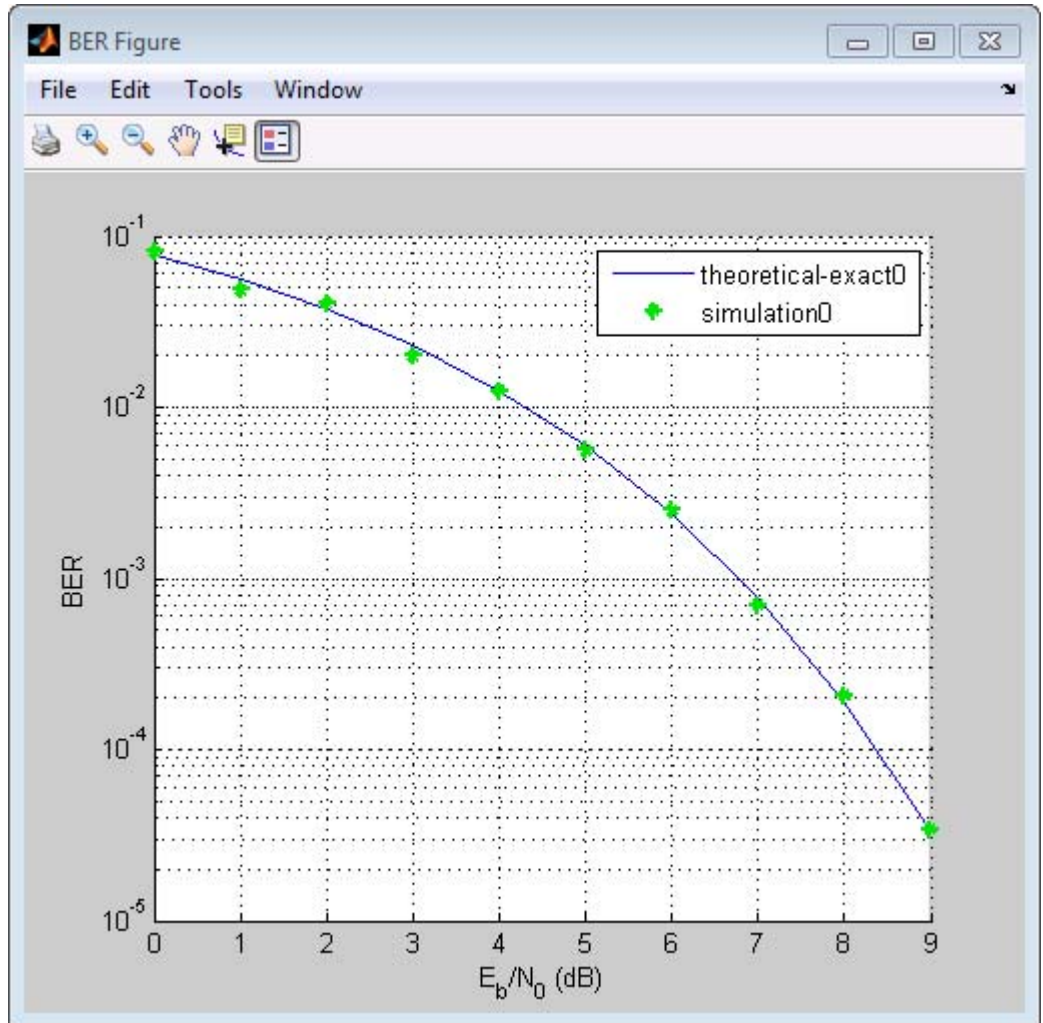
The **Number of bits** is also a stop criteria for the simulation. The simulation stops when it transmits the number of bits you specify for this parameter. In this example, the simulation either stops when it transmits 10 million bits or when it detects 200 errors.

12 Click the **Browse** button.

13 Navigate to `matlab/help/toolbox/comm/examples`, and select `doc_design_iteration_basic_m.m`.

14 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 E_b/N_0 .
- 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 stopping the simulation.

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 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 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**.

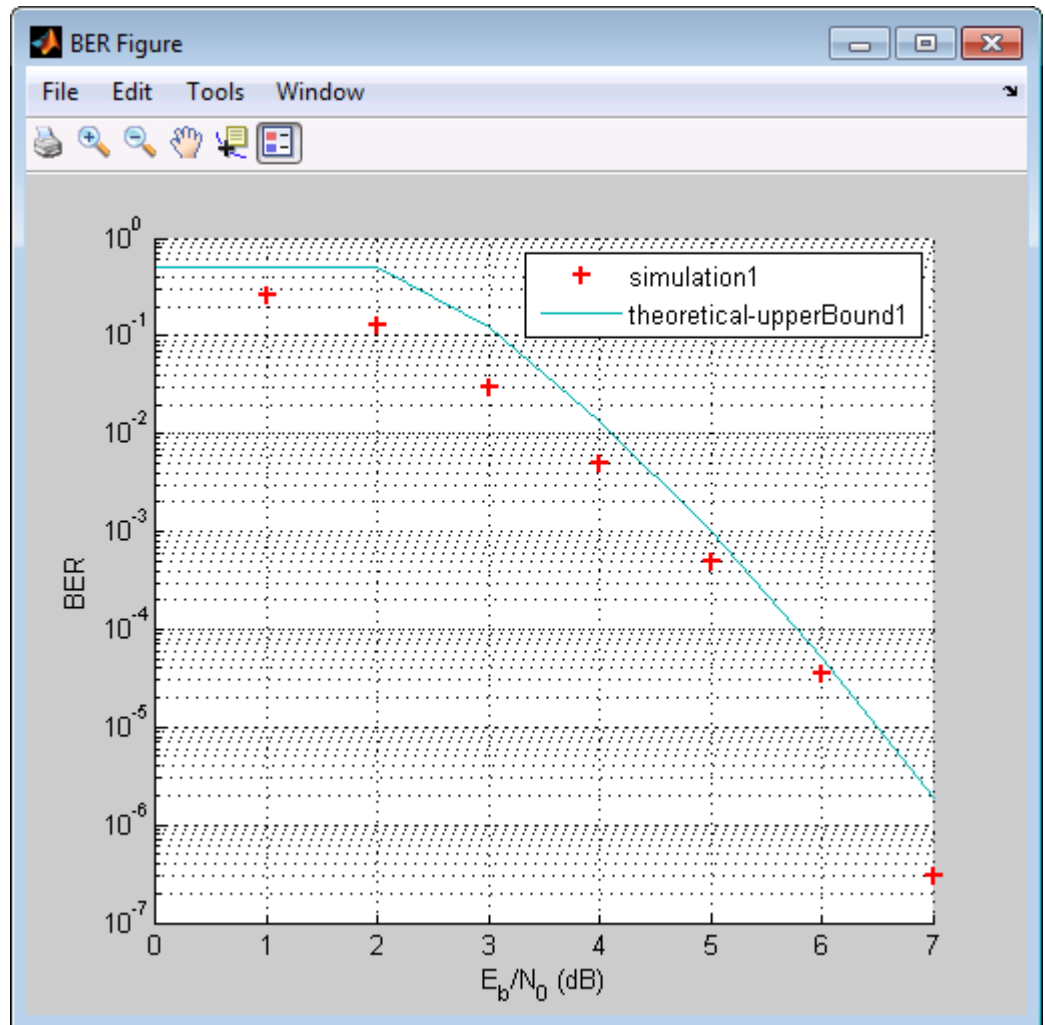
11 Enter 1e7 for the **Number of bits**.

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.



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.

- 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**.

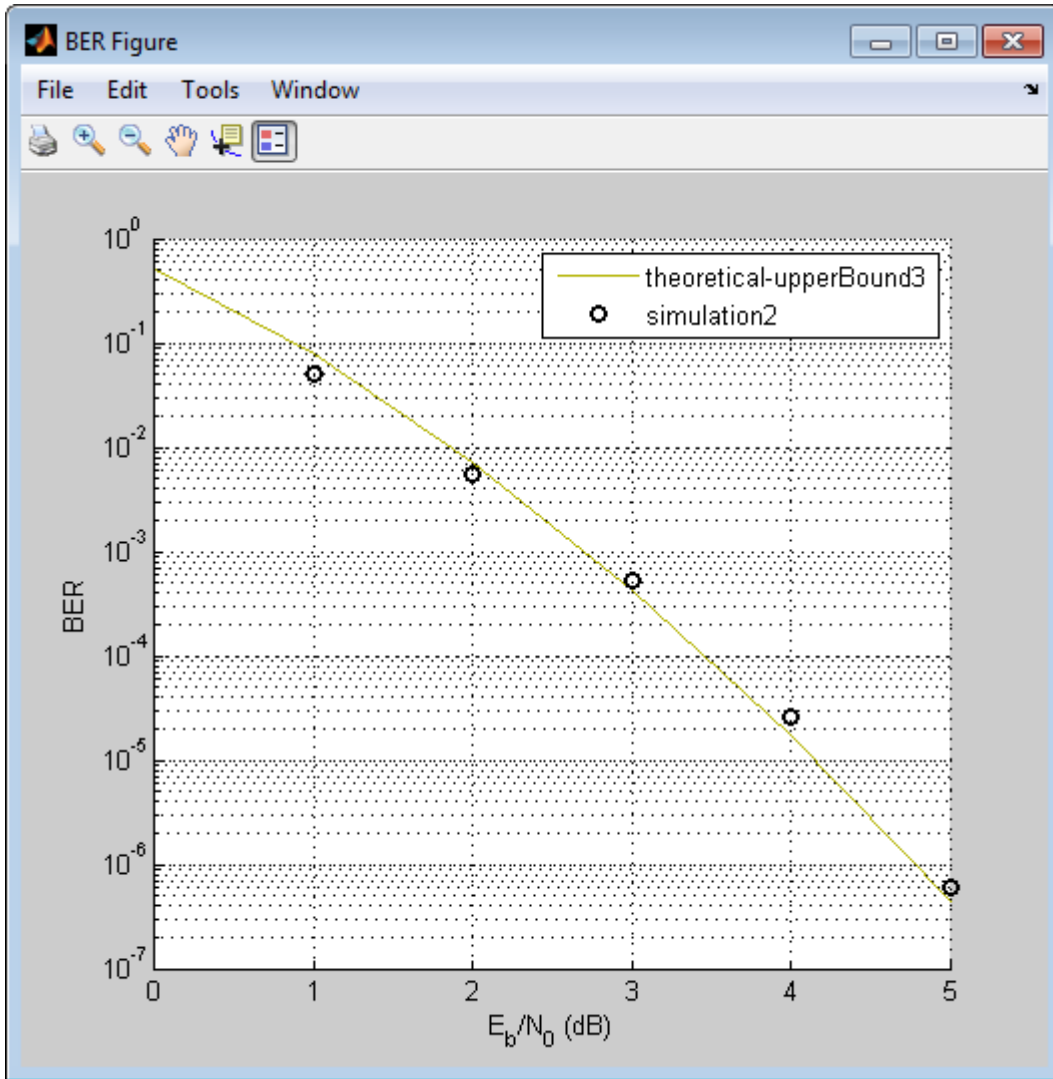
This example uses soft-decision Viterbi decoding. The demodulator maps the received signal to log likelihood ratios, improving BER performance results.

- 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.



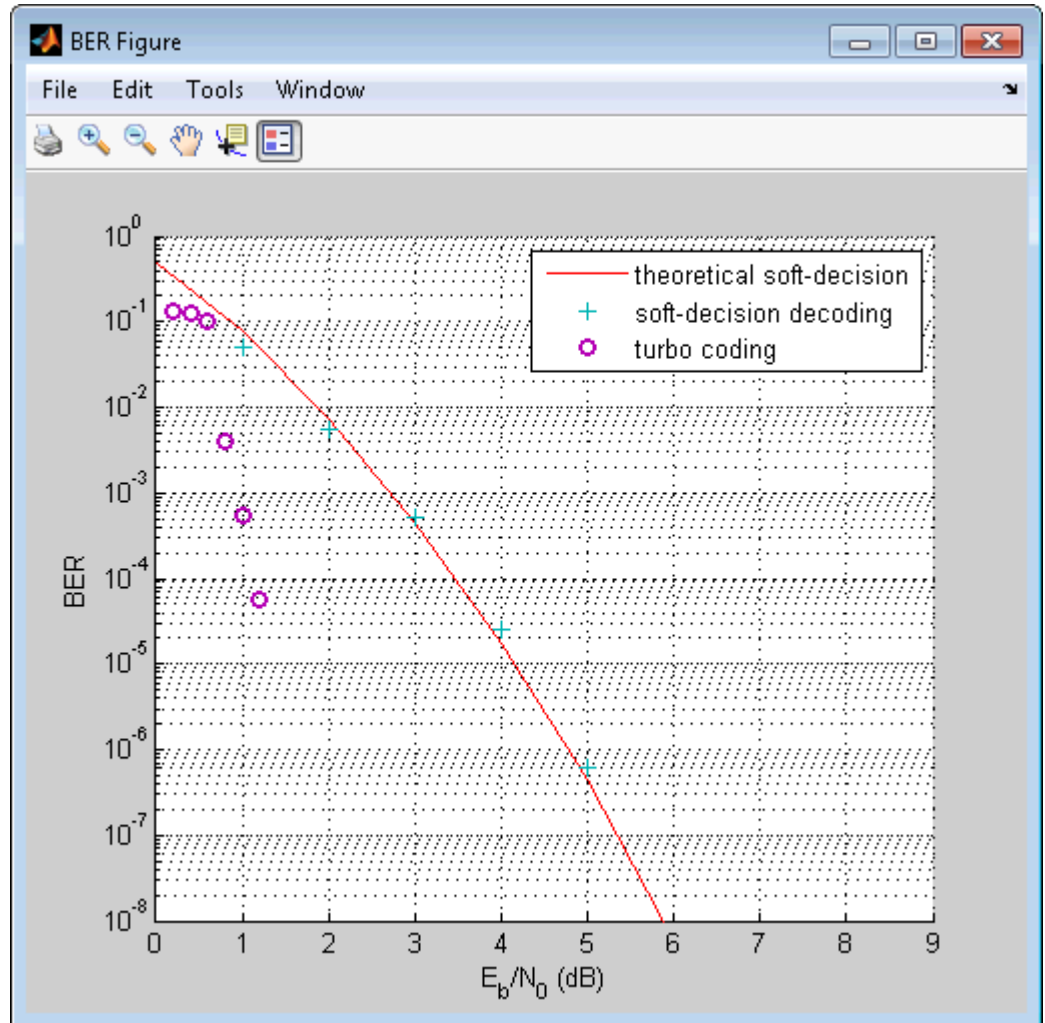
When you plot the soft-decision theoretical curve, you will observe BER curve improvements of about 2 dB relative to the hard-decision decoding. Notice that the simulation results also reflects a similar BER improvement.

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 a specified BER at much lower SNR values than do convolutional encoders. As a result, this iteration uses a lower range of E_b/N_0 values than the previous section.

- 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.



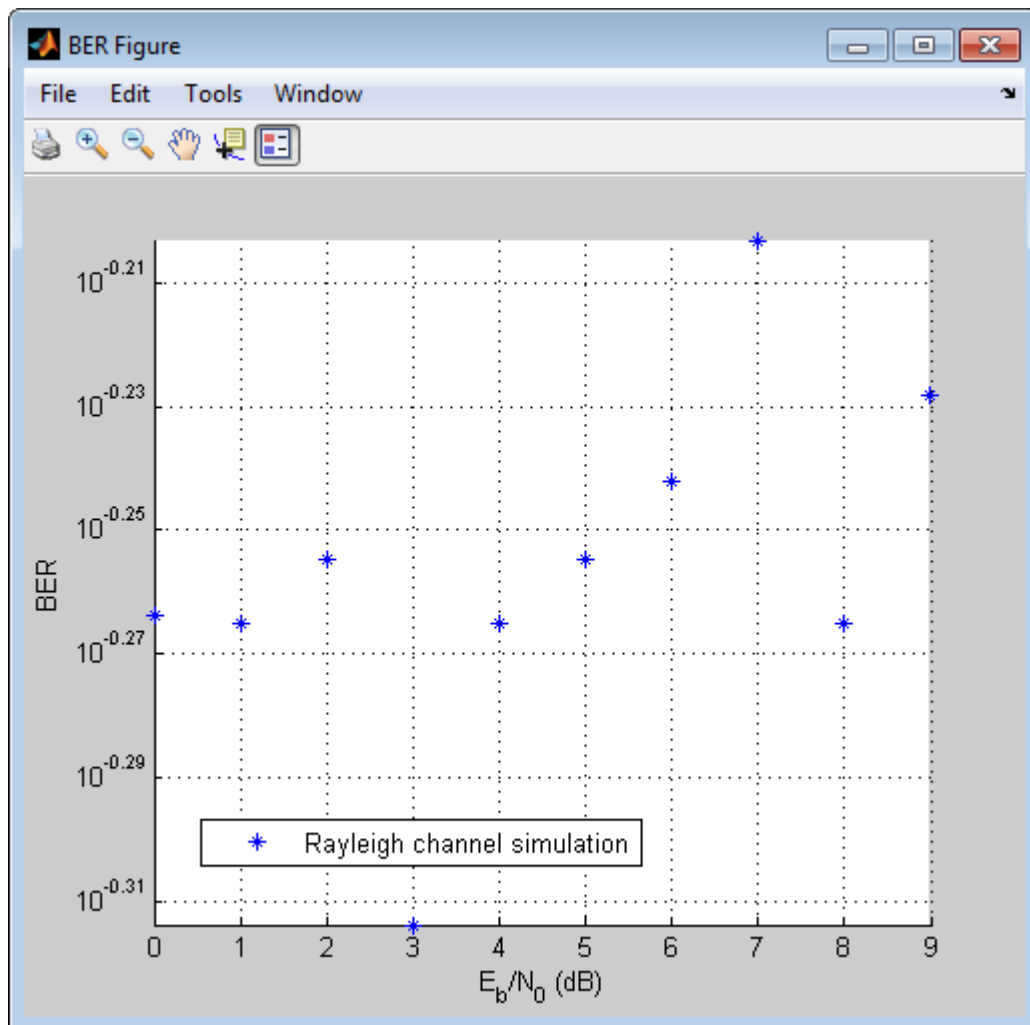
Apply a Rayleigh channel model

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. Many wireless wideband communications channels (4G LTE, WiFi, etc.) are highly susceptible to the effects of multipath propagation, which introduces

intersymbol interference (ISI). Therefore, you must model those 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.

- 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.



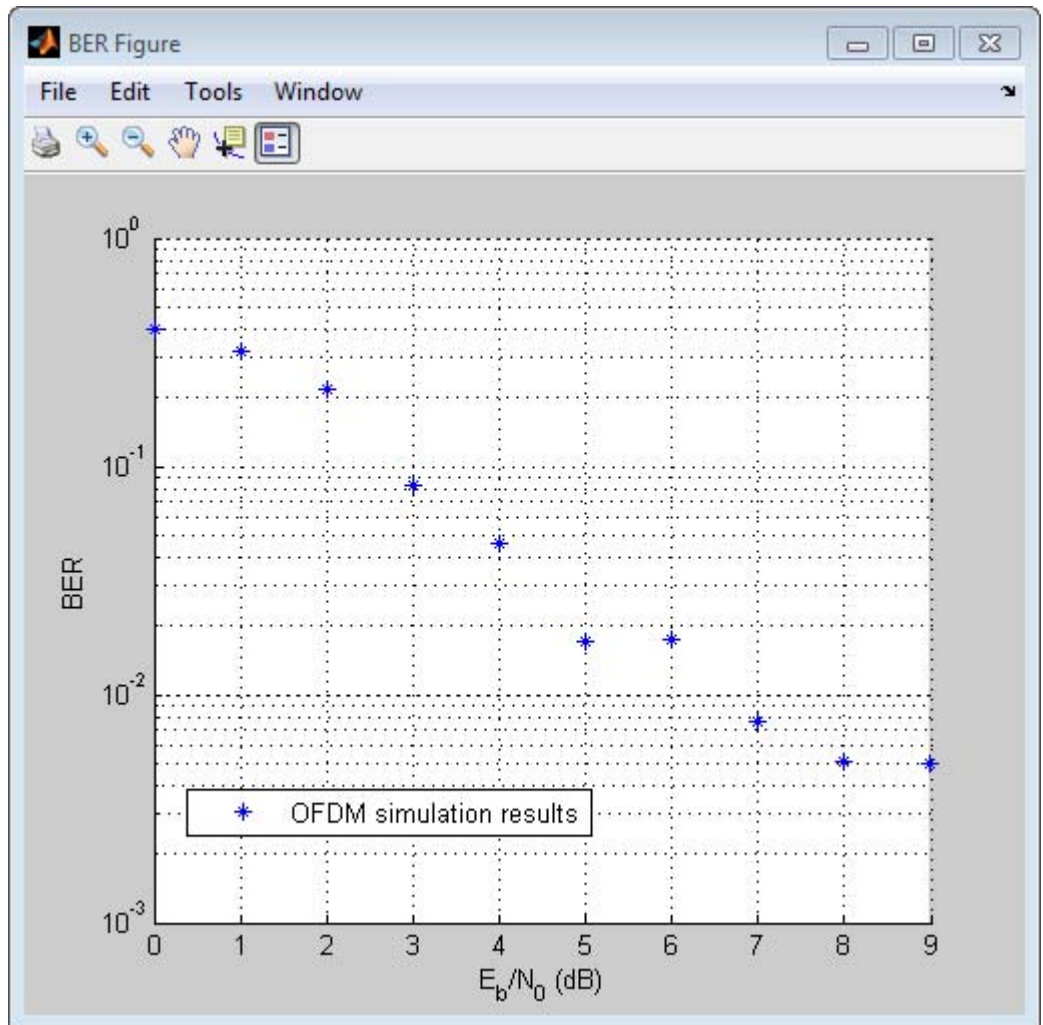
In the presence of multipath fading, the BER performance reduces to that of a binary channel with a consistent value of one-half. To correct the effect of multipath fading, you must add equalization to the communications system.

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. The `comm.OFDMModulator` and `comm.OFDMDemodulator` System objects are added in this phase of the example.

- 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.



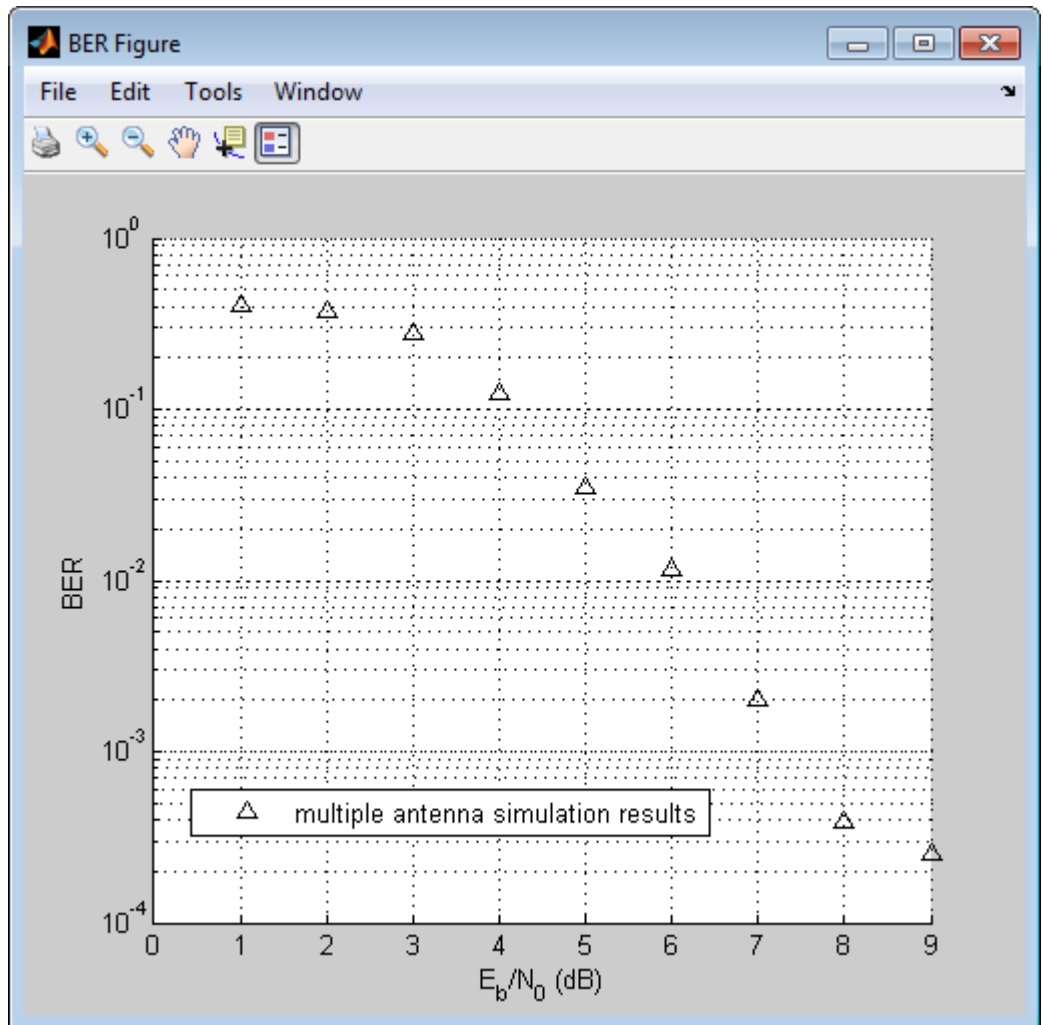
Use multiple antennas to further improve system performance

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.

- 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.



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.

- 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`.

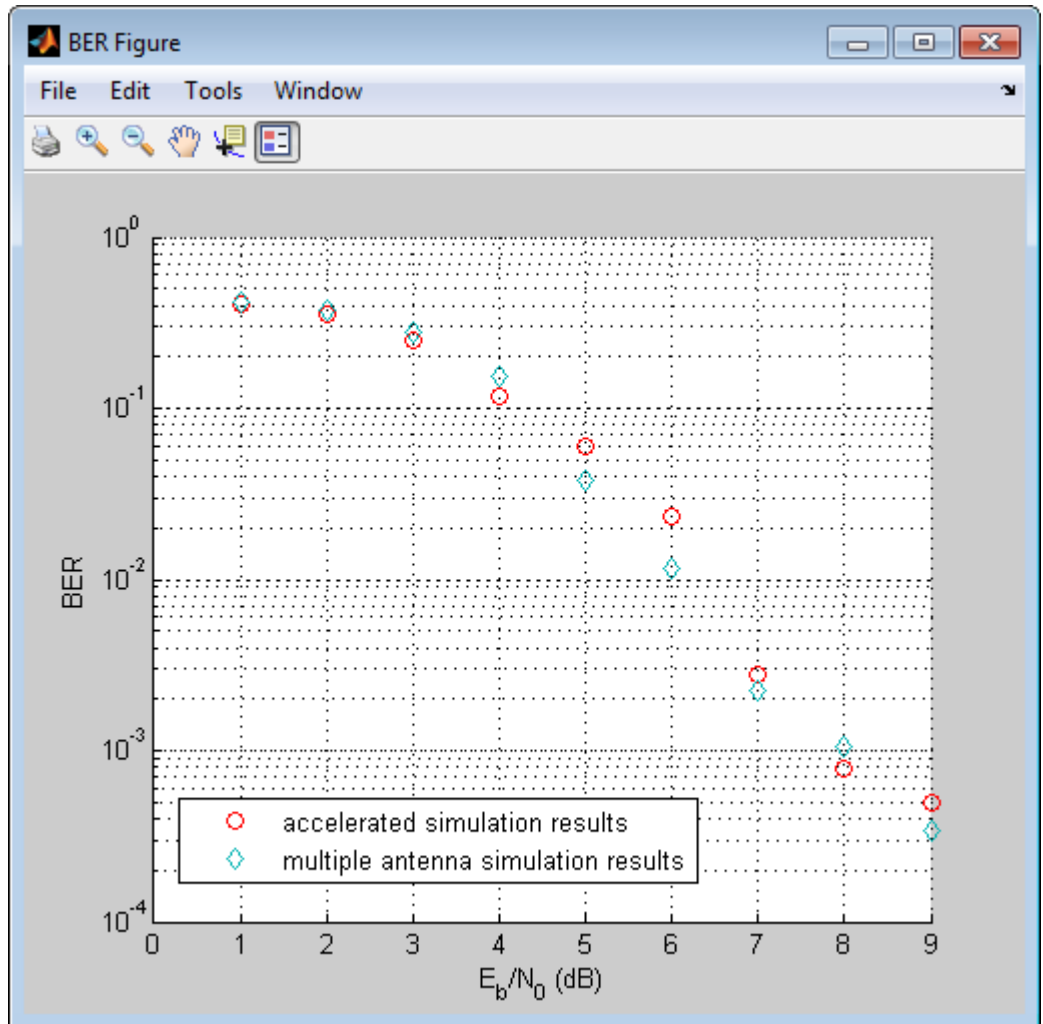
If you run the mex file you just generated in BERTool, you will obtain the simulation results more quickly.

- 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 to complete.



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, `doc_fcn_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 = doc_fcn_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;
```

Starting parallel pool (parpool) using the 'local' profile ... connected to

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, doc_fcn_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 = doc_fcn_qpsk_sim_with_awgn(EbNoVec, idx, ...  
            totalErrors/numWorkers, totalBits/numWorkers);
```

```
        numErrors(idx,n) = errorStats(idx,2);
```

```
        numBits(idx,n) = errorStats(idx,3);
```

```
    end
```

```
end
```

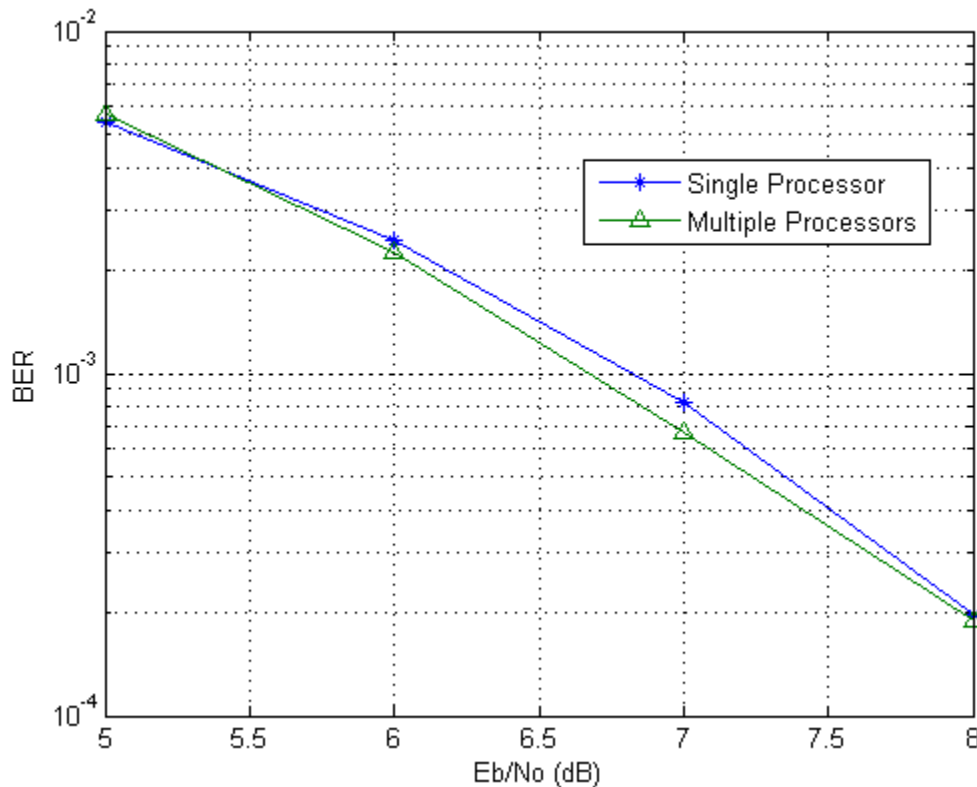
```
simParallelTime = toc;
```

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 = 170.1 sec for one worker
```

```
Simulation time = 52.7 sec for multiple workers
```

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

Visualization and Measurements

- “Scatter Plot and Eye Diagram with MATLAB” on page 3-2
- “Scatter Plot and Eye Diagram with MATLAB” on page 3-8
- “EVM and MER Measurements with Simulink” on page 3-14
- “ACPR and CCDF Measurements with MATLAB” on page 3-22

Scatter Plot and Eye Diagram with MATLAB

This example shows how to use the Communication System Toolbox 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 filter.

Set the RRC filter parameters.

```
span = 10;           % Filter span
rolloff = 0.2;       % Rolloff factor
sps = 8;             % Samples per symbol
```

Create the filter coefficients using the `rcosdesign` function.

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

Generate random symbols for an alphabet size of 4.

```
rng('default')
data = randi([0 3],5000,1);
```

Apply QPSK modulation.

```
dataMod = pskmod(data, 4, pi/4);
```

Filter the modulated data.

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

Calculate the SNR for an oversampled QPSK signal.

```
EbNo = 20;
snr = EbNo + 10*log10(2) - 10*log10(sps);
```

Add noise to the transmitted signal.

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

Apply the RRC receive filter.

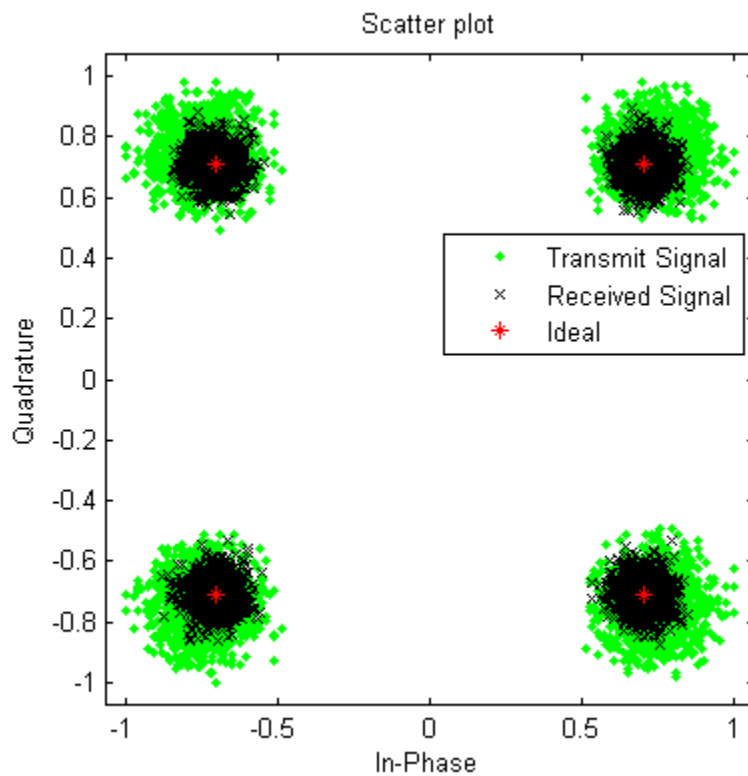
```
rxSigFilt = upfirdn(rxSig, filtCoeff, 1, sps);
```

Demodulate the filtered signal.

```
% Demodulate the filtered signal.  
dataOut = pskdemod(rxSigFilt, 4, pi/4, 'gray');
```

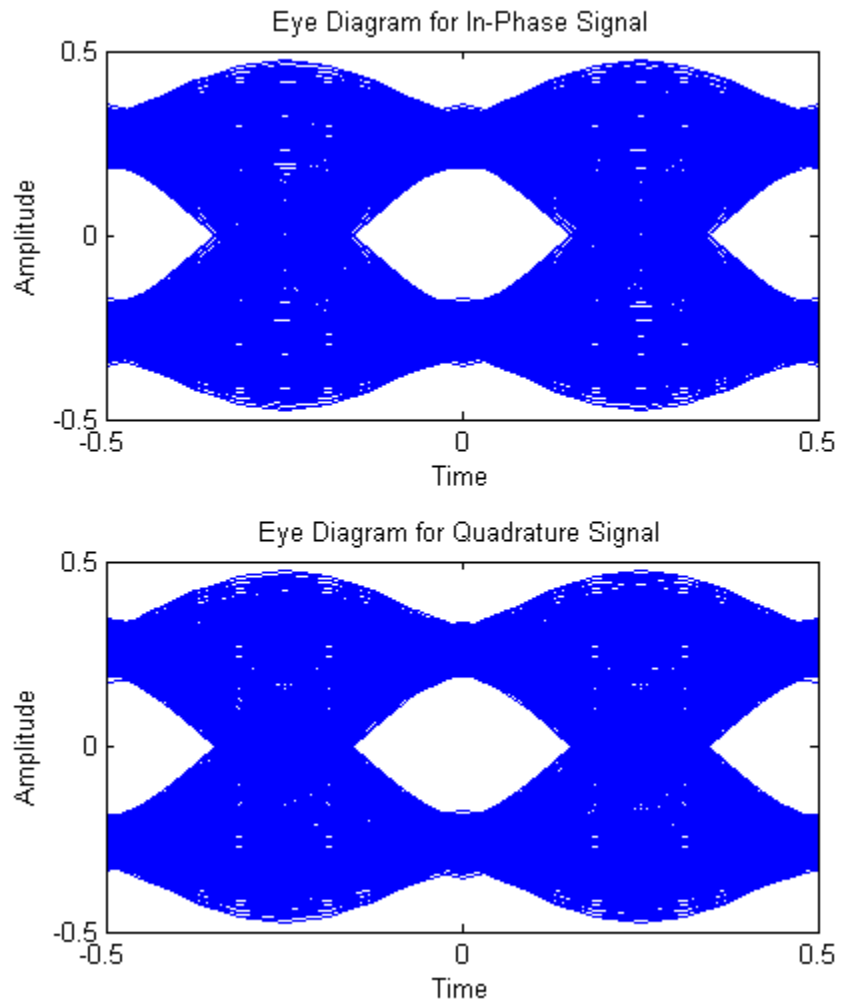
Use the `scatterplot` function to show scatter plots of the signal before and after filtering. You can see that the receive filter improves performance as the constellation more closely matches the ideal values. The first `span` symbols and the last `span` symbols represent the cumulative delay of the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

```
h = scatterplot(sqrt(sps) * txSig(sps*span+1:end-sps*span), sps, 0, 'g. ');  
hold on  
scatterplot(rxSigFilt(span+1:end-span), 1, 0, 'kx', h)  
scatterplot(dataMod, 1, 0, 'r*', h)  
legend('Transmit Signal','Received Signal','Ideal','location','best')
```



Display the eye diagram for two symbol periods.

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



Change the rolloff factor to visualize its effect on the eye diagram as its value is increased.

```
rolloff = 0.5;
```

Generate new filter coefficients.

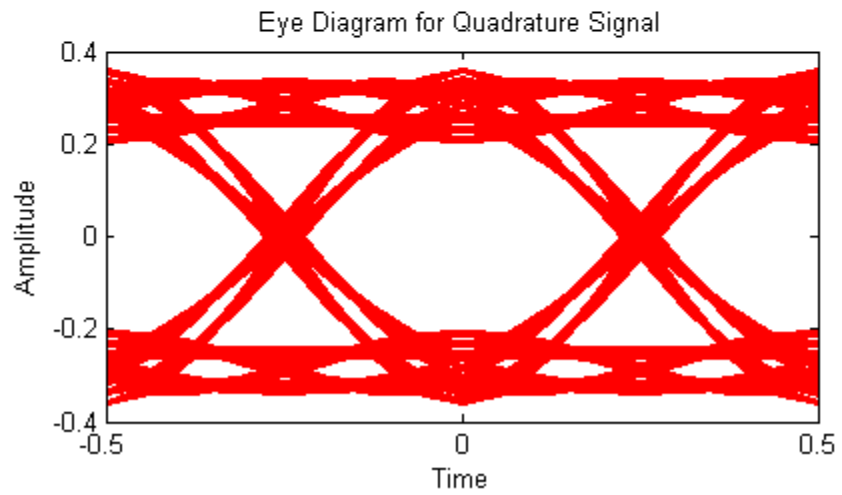
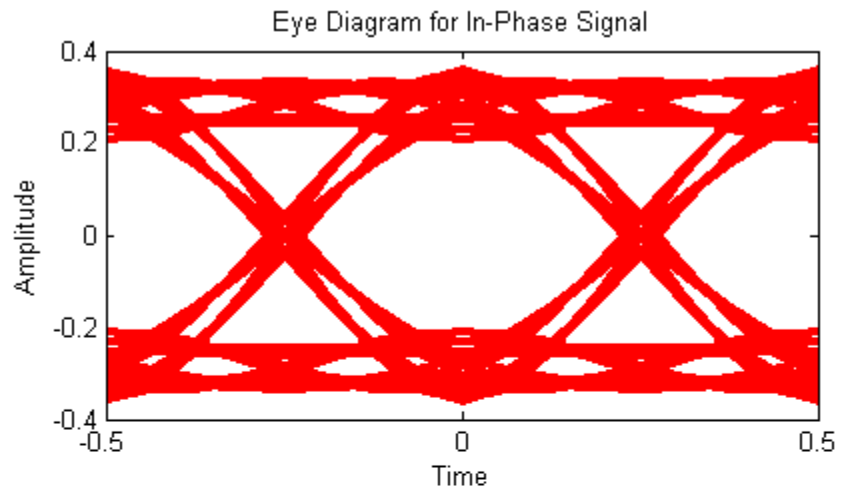
```
filtCoeff = rcosdesign(rolloff, span, sps);
```

Apply the RRC filter.

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

Display the eye diagram. You can see that the eye is more "open" when the filter rolloff factor is increased.

```
eyediagram(txSig(sps*span+1:end-sps*span), 2*sps, 1, 0, 'r')
```

Scatter Plot and Eye Diagram with MATLAB

This example shows how to use the Communication System Toolbox 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 filter.

Scatter Plot

Set the RRC filter parameters.

```
span = 10;           % Filter span
rolloff = 0.2;       % Rolloff factor
sps = 8;             % Samples per symbol
```

Create the filter coefficients using the `rcosdesign` function.

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

Generate random symbols for an alphabet size of 4.

```
rng('default')
data = randi([0 3],5000,1);
```

Apply QPSK modulation.

```
dataMod = pskmod(data, 4, pi/4);
```

Filter the modulated data.

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

Calculate the SNR for an oversampled QPSK signal.

```
EbNo = 20;
snr = EbNo + 10*log10(2) - 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.

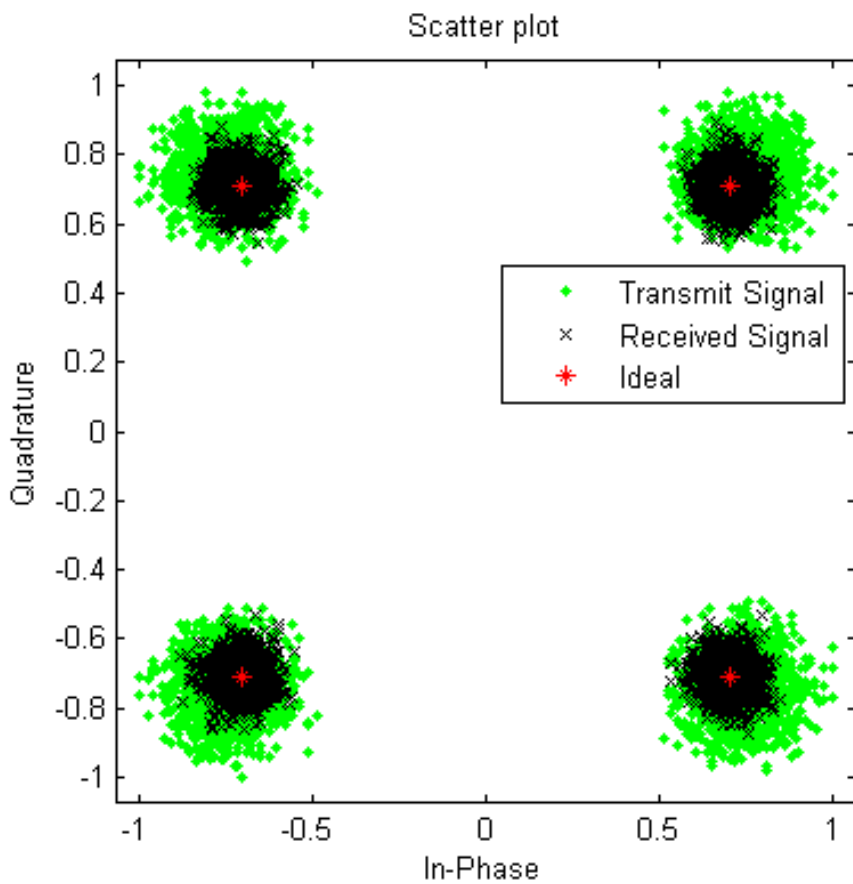
```
dataOut = pskdemod(rxSigFilt, 4, pi/4, 'gray');
```

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, 0, 'g.');
```

hold on

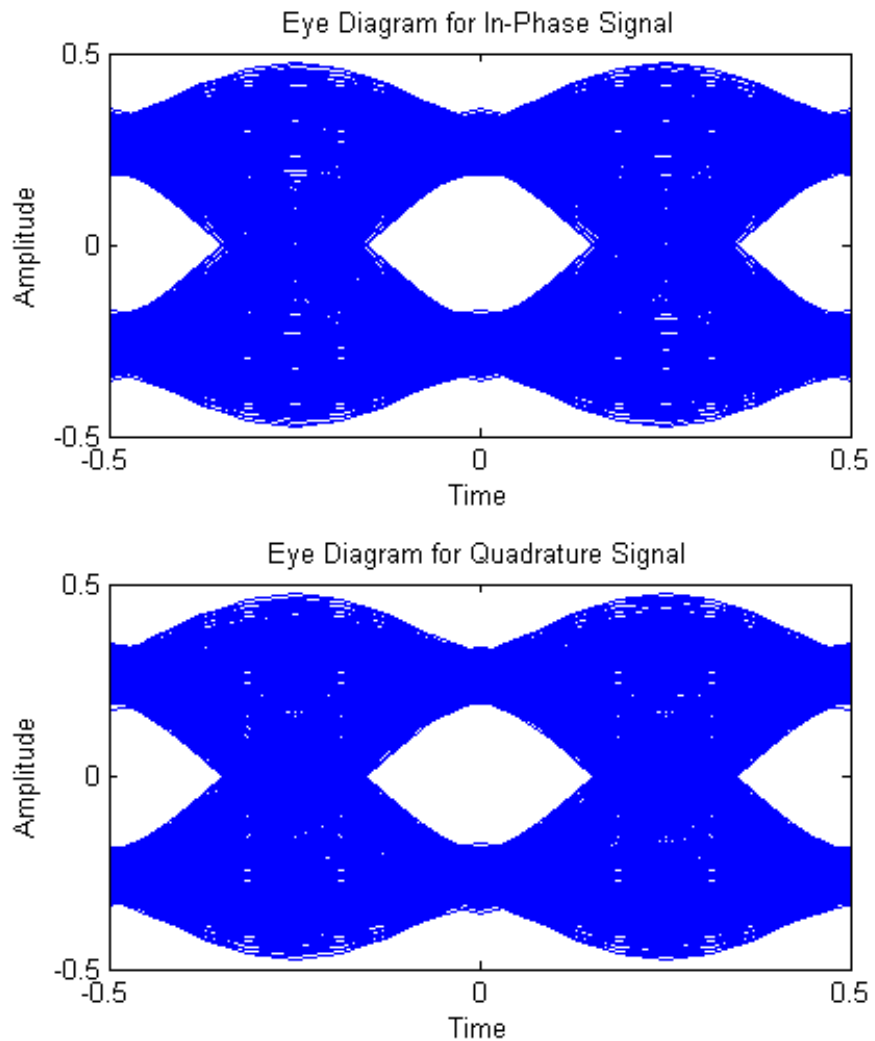
```
scatterplot(rxSigFilt(span+1:end-span), 1, 0, 'kx', h)
scatterplot(dataMod, 1, 0, 'r*', h)
legend('Transmit Signal','Received Signal','Ideal','location','best')
```



Eye Diagram

Display the eye diagram for two symbol periods.

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



Change the rolloff factor to visualize its effect on the eye diagram as its value is increased.

```
rolloff = 0.5;
```

Generate new filter coefficients.

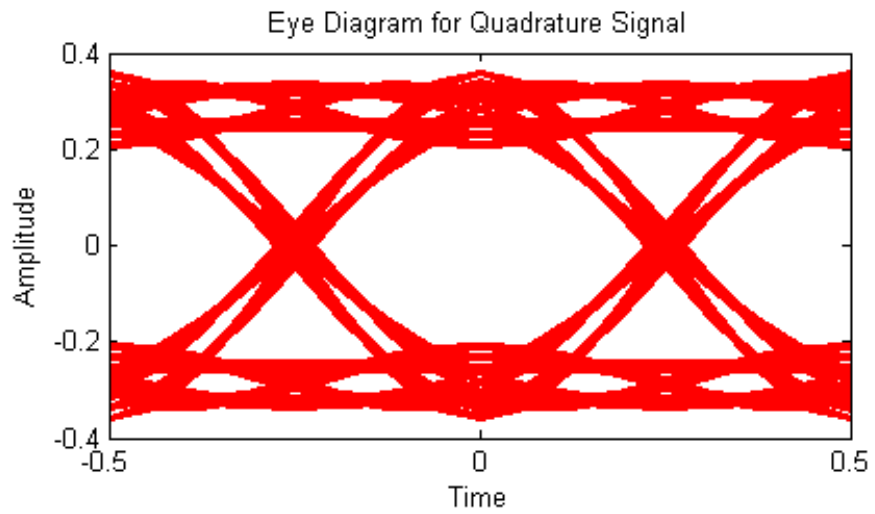
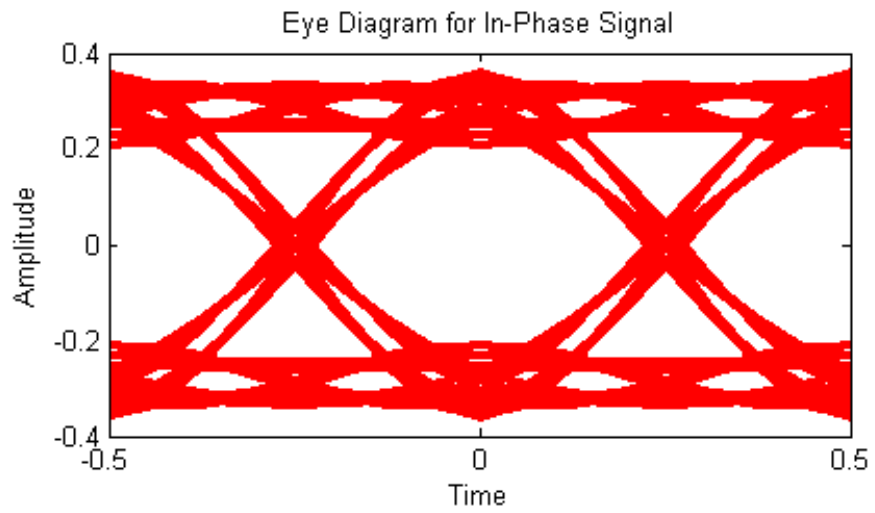
```
filtCoeff = rcosdesign(rolloff, span, sps);
```

Apply the RRC filter.

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

Display the eye diagram. You can see that the eye is more "open" when the filter rolloff factor is increased.

```
eyediagram(txSig(sps*span+1:end-sps*span), 2*sps, 1, 0, 'r')
```



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.

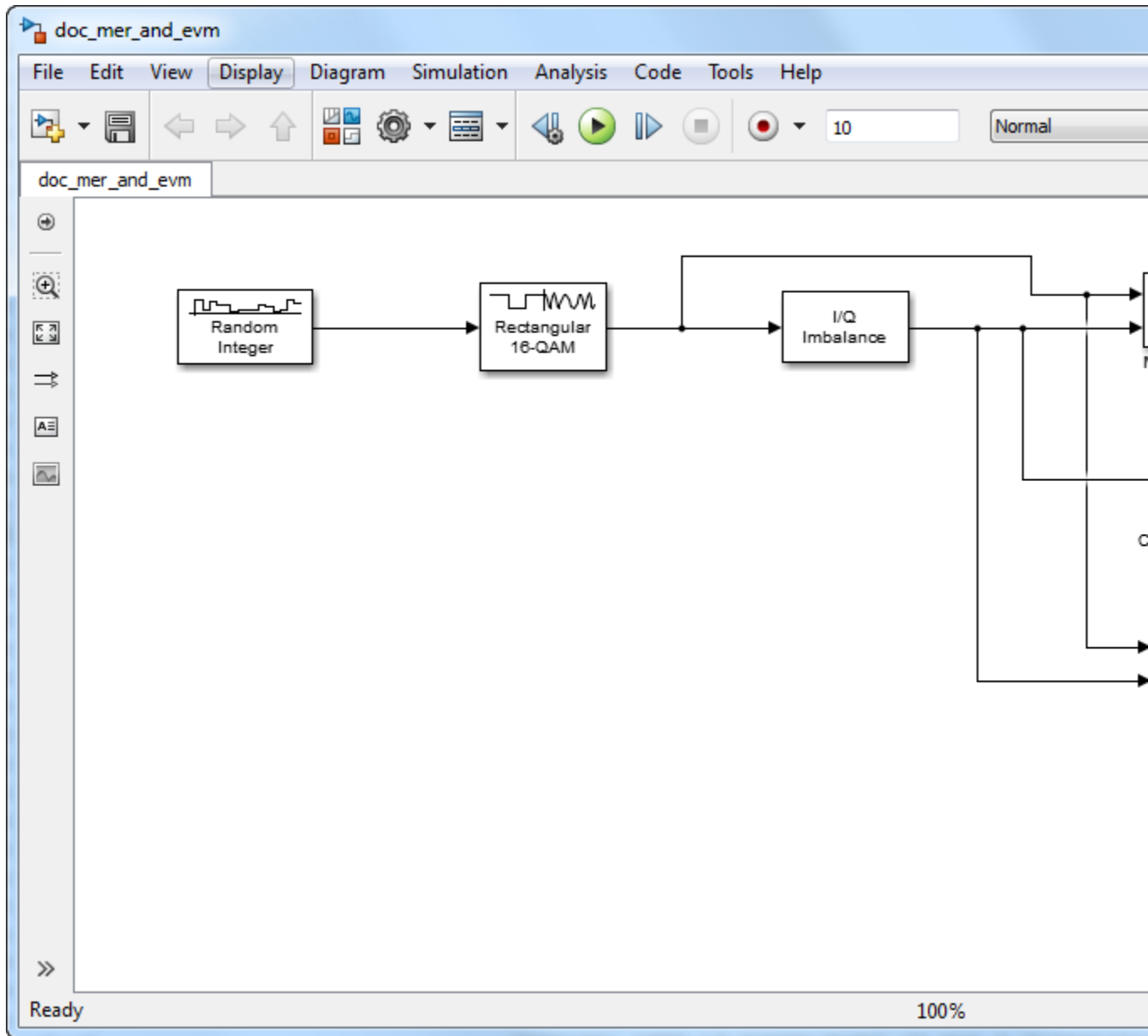
Load the model `doc_mer_and_evm` from the MATLAB command prompt.

```
doc_mer_and_evm
```

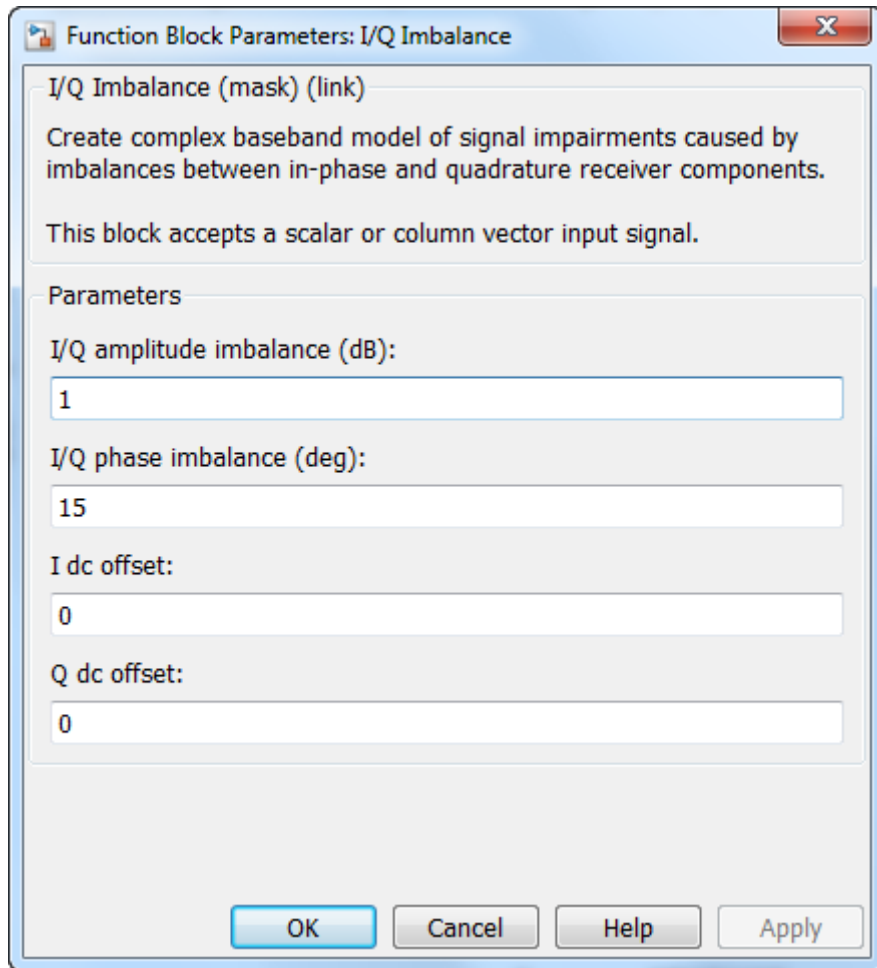
This example includes:

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

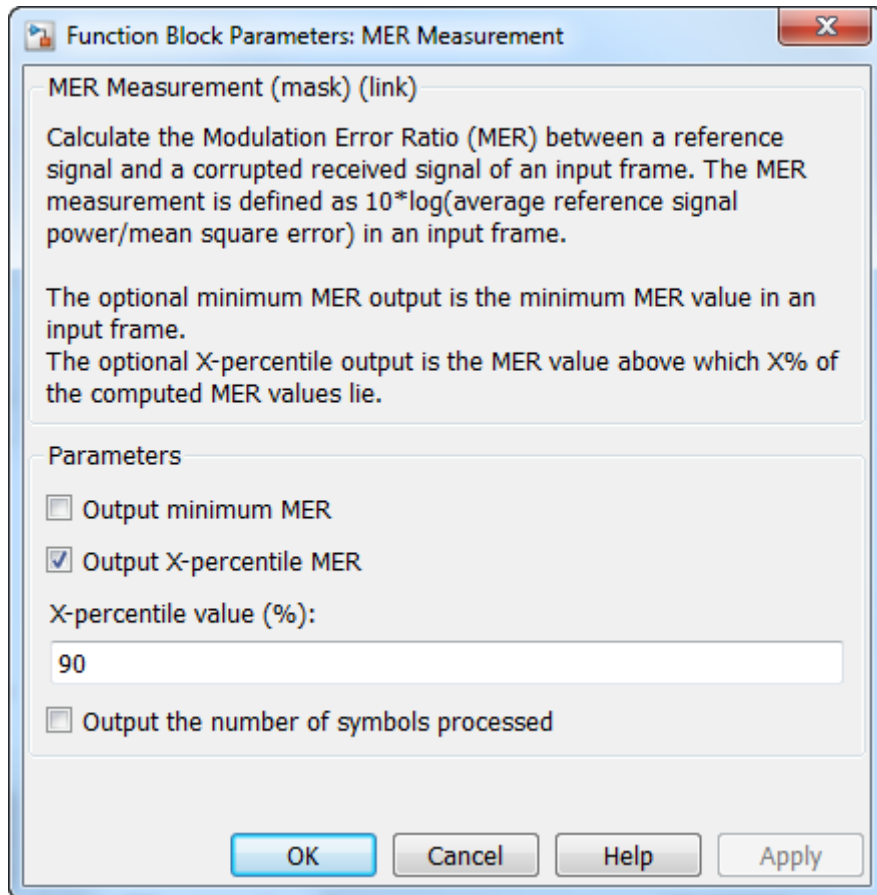
The model applies an I/Q imbalance to a QAM-modulated signal at which point MER and EVM measurements are made. The constellation diagram provides a visual representation of the effects the imbalance has on the modulation performance indicators.



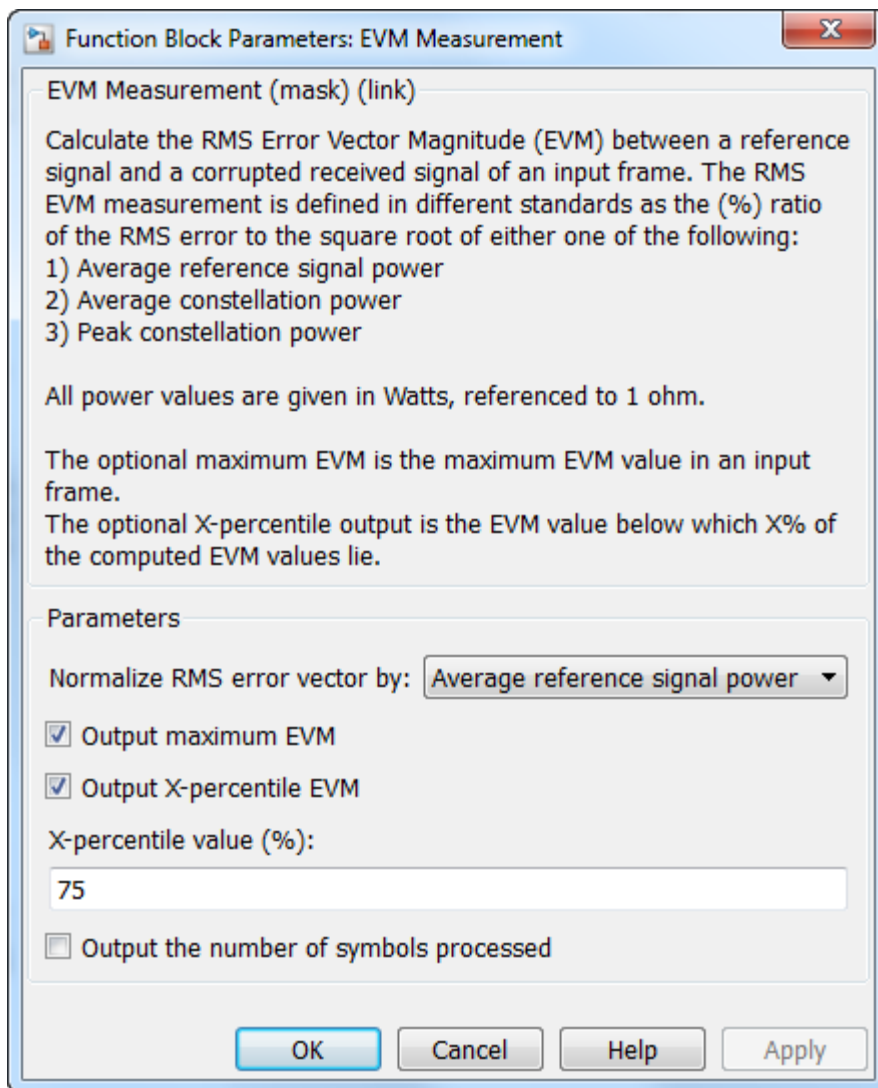
The I/Q Imbalance block is set to that the **I/Q amplitude imbalance (dB)** is set to 1 and the **I/Q phase imbalance (deg)** is set to 15.



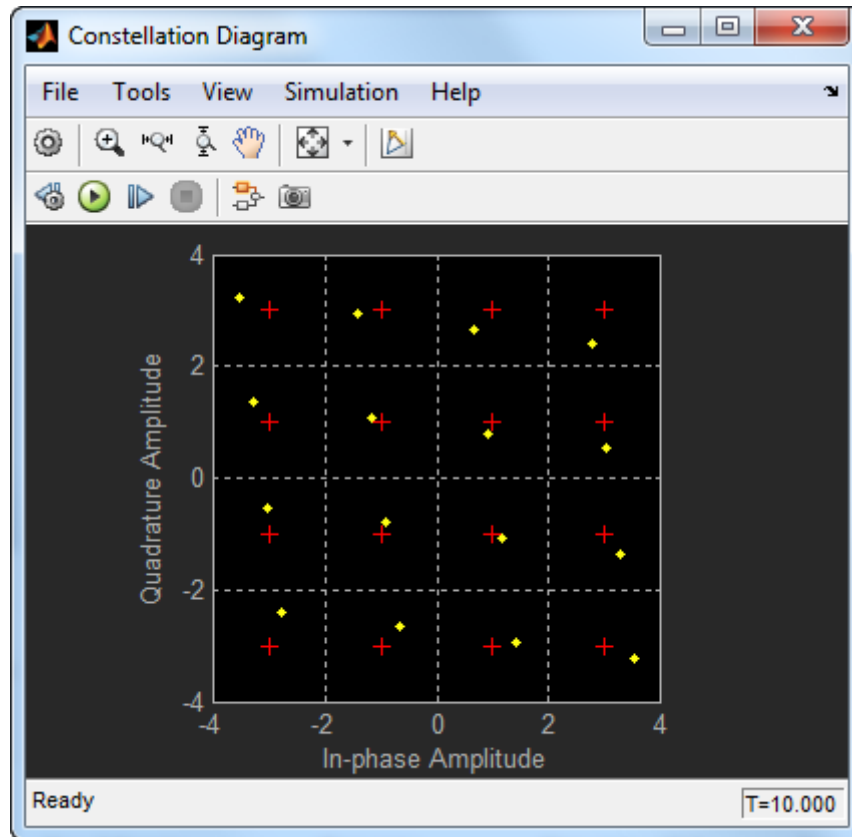
The MER Measurement block is set so that it outputs the X-percentile MER value which is set to 90%.



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

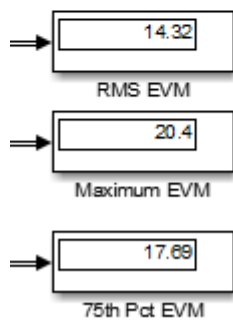
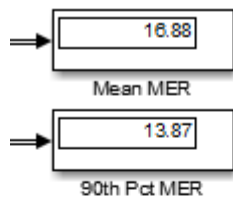


Run the model.

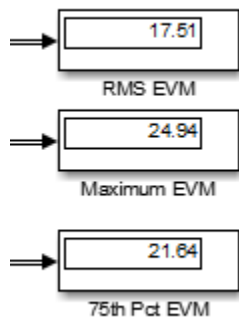
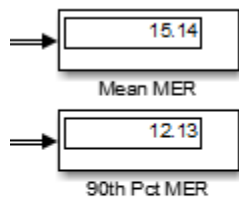


You can see that I/Q amplitude and phase imbalance has shifted the constellation diagram so that each symbol is not exactly equal to its reference symbol (shown in red). Change the I/Q Imbalance block to see the effects of differing imbalances on the constellation diagram.

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.



ACPR and CCDF Measurements with MATLAB

In this section...
“ACPR Measurements” on page 3-22
“CCDF Measurements” on page 3-26

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 random number generator so that results are repeatable.

```
prevState = rng;  
rng(577)
```

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.

```
hMod = comm.QPSKModulator;  
x = step(hMod, 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.

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

Use the step method of the comm.ACPR System object to output the ACPR, the main channel power, and the adjacent channel power for the signal, *y*.

```
[ACPR, mainPower, adjPower] = step(hACPR, y)
```

```
ACPR =  
  
    -9.5103
```

```
mainPower =  
  
    28.9634
```

```
adjPower =  
  
    19.4531
```

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

```
release(hACPR)
hACPR.AdjacentChannelOffset = 75e3;
ACPR = step(hACPR, y)
```

```
ACPR =

    -13.2317
```

Reset hACPR and specify a 50 kHz adjacent channel offset.

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

Create a raised cosine filter and filter the modulated signal.

```
hFilt = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol', sps);
z = step(hFilt, 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.

```
ACPR = step(hACPR, z)
```

```
ACPR =

    -17.7338
```

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.

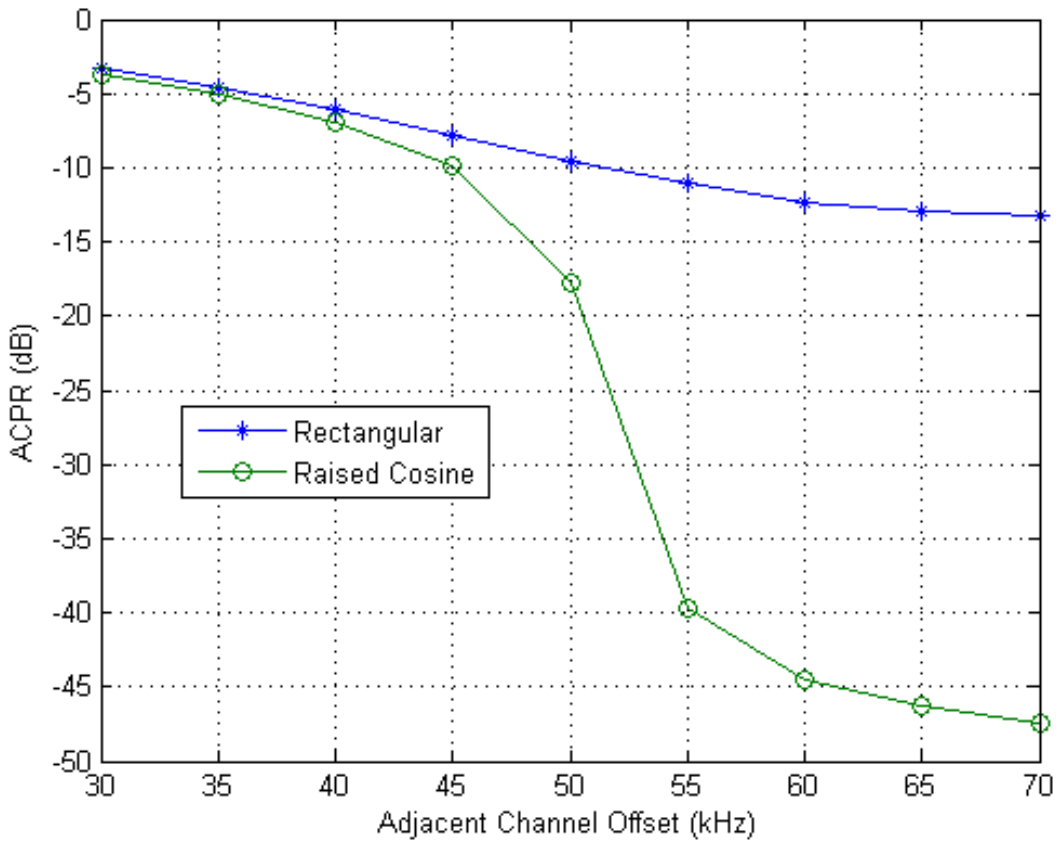
```
freqOffset = 1e3*(30:5:70);
release(hACPR)
hACPR.AdjacentChannelOffset = freqOffset;
```

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

```
ACPR1 = step(hACPR, y);  
ACPR2 = step(hACPR, 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
```



Return the random number generator to its initial state.

```
rng(prevState)
```

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.

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

Create a 64-QAM modulator and an OFDM modulator. The QAM modulated signal will be evaluated by itself and evaluated again after OFDM modulation is applied.

```
hMod = comm.RectangularQAMModulator('ModulationOrder',64);  
hOFDM = comm.OFDMModulator('FFTLength', 256, 'CyclicPrefixLength', 32);
```

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

```
info(hOFDM)  
ofdmInputSize = hOFDM.info.DataInputSize;  
ofdmOutputSize = hOFDM.info.OutputSize;
```

```
ans =
```

```
    DataInputSize: [245 1]  
    OutputSize: [288 1]
```

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);
```

Use the default random number generator to ensure repeatability.

```
rng default
```

Generate the 64-QAM and OFDM signals for evaluation.

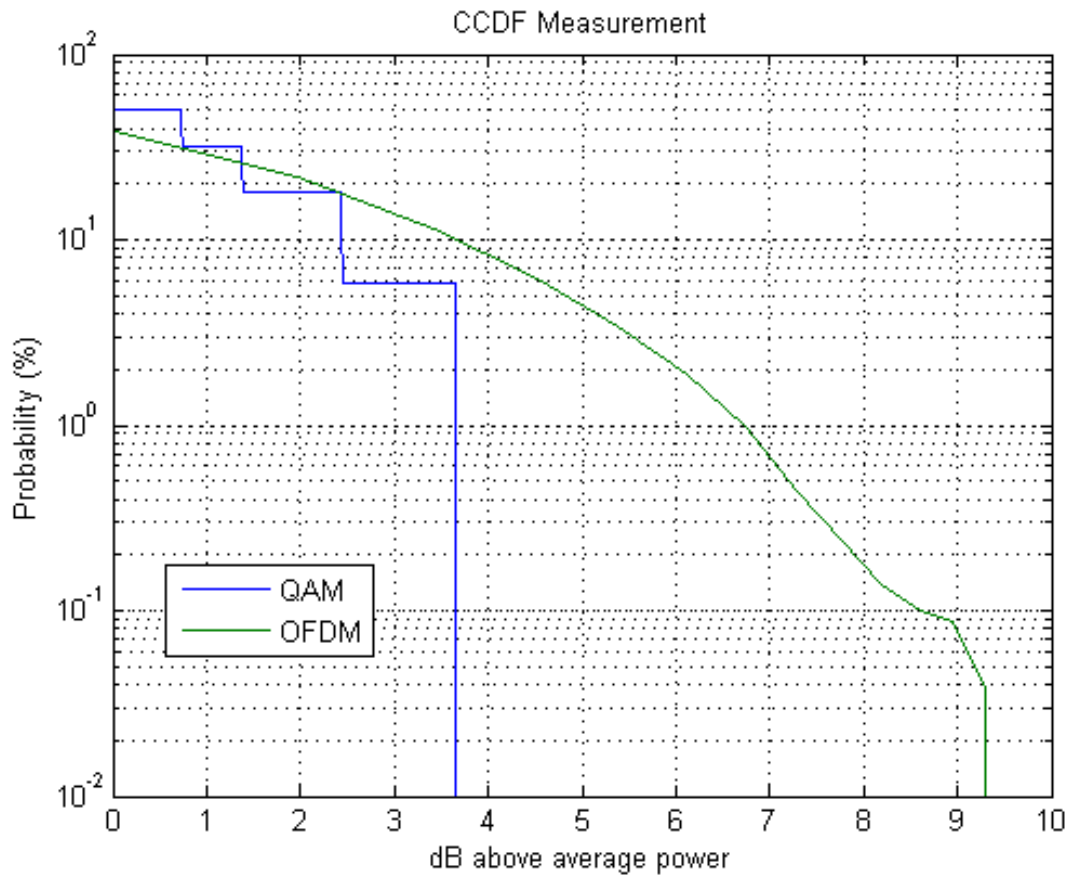
```
for k = 1:numFrames
    % Generate random data symbols
    data = randi([0, 63], ofdmInputSize);
    % Apply 64-QAM modulation
    tmpQAM = step(hMod, data);
    % Apply OFDM modulation to the QAM-modulated signal
    tmpOFDM = step(hOFDM, tmpQAM);
    % Save the signal data
    qamSig((1:ofdmInputSize)+(k-1)*ofdmInputSize(1)) = 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] = step(hCCDF, [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(hCCDF)
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.

System Objects

- “What Is a System Toolbox?” on page 4-2
- “What Are System Objects?” on page 4-3
- “When to Use System Objects Instead of MATLAB Functions” on page 4-5
- “System Design and Simulation in MATLAB” on page 4-8
- “System Design and Simulation in Simulink” on page 4-9
- “System Objects in MATLAB Code Generation” on page 4-10
- “System Objects in Simulink” on page 4-17
- “System Object Methods” on page 4-18
- “System Design in MATLAB Using System Objects” on page 4-21
- “System Design in Simulink Using System Objects” on page 4-28

What Is a System Toolbox?

System Toolbox products provide algorithms and tools for designing, simulating, and deploying dynamic systems in MATLAB and Simulink. These toolboxes contain MATLAB functions, System objects, and Simulink blocks that deliver the same design and verification capabilities across MATLAB and Simulink, enabling more effective collaboration among system designers. Available System Toolbox products include:

- DSP System Toolbox
- Communications System Toolbox
- Computer Vision System Toolbox
- Phased Array System Toolbox

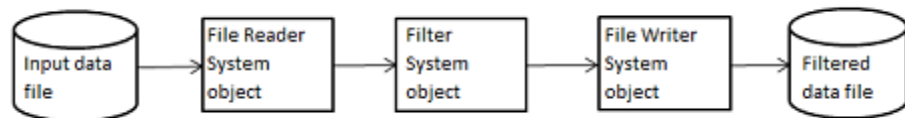
System Toolboxes support floating-point and fixed-point streaming data simulation for both sample- and frame-based data. They provide a programming environment for defining and executing code for various aspects of a system, such as initialization and reset. System Toolboxes also support code generation for a range of system development tasks and workflows, such as:

- Rapid development of reusable IP and test benches
- Sharing of component libraries and systems models across teams
- Large system simulation
- C-code generation for embedded processors
- Finite wordlength effects modeling and optimization
- Ability to prototype and test on real-time hardware

What Are System Objects?

A System object is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer™ license)
- C code generation (requires a MATLAB Coder or Simulink Coder license)
- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Define New System Objects”.

When to Use System Objects Instead of MATLAB Functions

In this section...

“System Objects vs. MATLAB Functions” on page 4-5

“Process Audio Data Using Only MATLAB Functions Code” on page 4-5

“Process Audio Data Using System Objects” on page 4-6

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, you can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB function-only code for reading audio data.

The code reads audio data from a file, filters it, and then plays the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audiointro(fname);  
maxSamples = audioInfo.TotalSamples;  
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160, .15);
```

Initialize the filter states.

```
z = zeros(1, numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;  
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1  
    audio = audioread(fname, [nIdx nIdx+frameSize-1]);  
    [y,z] = filter(b,1,audio,z);  
    sound(y,fs);  
    nIdx = nIdx+frameSize;  
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows how to write System objects code for reading audio data.

The code uses System objects from the DSP System Toolbox software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname,'OutputDataType','single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

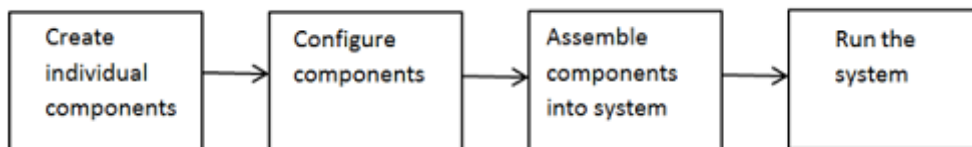
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);  % Filter the data
    step(audioOut,y);        % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects in MATLAB as shown in this diagram.



- 1** Create individual components — Create the System objects to use in your system. See “Create Components for Your System” on page 4-21 for information. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Define New System Objects”.
- 2** Configure components — If necessary, change the objects’ property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See “Configure Components for Your System” on page 4-22 for information.
- 3** Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variable as inputs and outputs to simulate your system. See “Assemble Components to Create Your System” on page 4-23 for information.
- 4** Run the system — Run your program, which uses the `step` method to run your system’s System objects. You can change tunable properties while your system is running. See “Run Your System” on page 4-25 and “Reconfigure Your System During Runtime” on page 4-25 for information.

System Design and Simulation in Simulink

You can use System objects in your model to simulate in Simulink.

- 1** Create a System object to be used in your model. See “Define New Kinds of System Objects for Use in Simulink” on page 4-28 for information.
- 2** Test your new System object in MATLAB. See “Test New System Objects in MATLAB” on page 4-34
- 3** Add the System object to your model using the MATLAB System block. See “Add System Objects to Your Simulink Model” on page 4-35 for information.
- 4** Add other Simulink blocks as needed and connect the blocks to construct your system.
- 5** Run the system

System Objects in MATLAB Code Generation

In this section...
“System Objects in Generated Code” on page 4-10
“System Objects in codegen” on page 4-16
“System Objects in the MATLAB Function Block” on page 4-16
“System Objects in the MATLAB System Block” on page 4-16
“System Objects and MATLAB® Compiler™ Software” on page 4-16

System Objects in Generated Code

You can generate C/C++ code in MATLAB from your system that contains System objects by using the MATLAB Coder product. Using this product, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. You do not need the MATLAB Coder product to generate code in Simulink.

Note Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects Code with Persistent Objects for Code Generation

This example shows how to use System objects to make MATLAB code suitable for code generation. The example highlights key factors to consider, such as passing property values and using extrinsic functions. It also shows that by using persistent objects, the object states are maintained between calls.

```
function w = lmssystem(x, d)
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent
    persistent hlms;
```

```

% Initialize persistent System objects only once.
% Do this with 'if isempty(persistent variable).'
% This condition will be false after the first time.

if isempty(hlms)
    % Create LMS adaptive filter used for system
    % identification. Pass property value arguments
    % as constructor arguments. Property values must
    % be constants during compile time.

    hlms = dsp.LMSFilter(11,'StepSize',0.01);
end

[~,~,w] = step(hlms,x,d);      % Filter weights
end

```

This example shows how to compile the `lmssystem` function and produce a MEX file with the same name in the current directory.

```

% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter

coefs = fir1(10,.25);
hfilt = dsp.FIRFilter('Numerator', coefs);

x = randn(1000,1);           % Input signal
hSrc = dsp.SignalSource(x,100); % Use x as input-signal with
                                % 100 samples per frame

% Generate code for lmssystem
codegen lmssystem -args {ones(100,1),ones(100,1)}

while ~isDone(hSrc)
    in = step(hSrc);
    d = step(hfilt,in) + 0.01*randn(100,1); % Desired signal
    w = lmssystem_mex(in,d);               % Call generated mex file
    stem([coefs.',w]);
end

```

For another detailed code generation example, see “Generate Code for MATLAB Handle Classes and System Objects” in the MATLAB Coder product documentation.

System Objects Code Without Persistent Objects for Code Generation

The following example, using System objects, does not use the persistent keyword because calling a persistent object with different data types causes a data type mismatch error. This example filters the input and then performs a discrete cosine transform on the filtered output. Each call to the `FilterAndDCTLib` function is independent and state information is not retained between calls.

```
function [out] = FilterAndDCTLib(in)
    hFIR = dsp.FIRFilter('Numerator',fir1(10,0.5));
    DCT = dsp.DCT;

    % Run the objects to get the filtered spectrum
    firOut = hFIR.step(in);
    out = hDCT.step(firOut);

function [out1, out2] = CompareRealInt(in1)
    % Call the library function, FilterAndDCTLib, which can
    % generate code for multiple calls each with a different data type.

    % Convert input data from double to int16
    in2 = int16(in1);

    % Call the library function for both data types, double and int16
    out1 = FilterAndDCTLib(in1);
    out2 = FilterAndDCTLib(in2);

function RunDCTExample
    % Execute everything needed at the command line to run the example

    warnState = warning('off','SimulinkFixedPoint:util:fxpParameterUnderflow
```

```

% Create vector, length 256, of data containing noise and sinusoids
dataLength = 256;
sampleData = rand(dataLength,1) + 3*sin(2*pi*[1:dataLength]*.085)' ...
            + 2*cos(2*pi*[1:dataLength]*.02)';

% Generate code and run generated file
codegen CompareRealInt -args {sampleData}
[out1,out2] = CompareRealInt_mex(sampleData);

% Compare the the floating point results, in blue
% with the int16 results, in red
plot(out1,'b');
hold on;
plot(out2,'r');
hold off

warning(warnState.state,warnState.identifier);
end

```

Usage Rules and Limitations for System Objects in Generated MATLAB Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an if statement with a call to `isempty()`.
- Set arguments to System object constructors as compile-time constants.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- The data type of the inputs should not change.

- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note Variable-size properties in MATLAB Function block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the MATLAB Function block.
- Do not use the Save and Restore Simulation State as `SimState` option for any System object in a MATLAB Function block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
 - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
 - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation.

- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- Do not use cell arrays.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

```
f = coder.MEXConfig;  
f.GlobalSyncMethod='NoSync'
```

Then, include '-config f' in your codegen command.

Methods

- Code generation support is available only for these System object methods:
 - get
 - getNumInputs
 - getNumOutputs
 - isDone (for sources only)
 - release
 - reset
 - set (for tunable properties)
 - step
- Code generation support for using dot notation depends on whether the System object is predefined in the software or is one that you defined.
 - For System objects that are predefined in the software, you cannot use dot notation to call methods.

- For System objects that you define, you can use dot notation or function call notation, with the System object as first argument, to call methods.

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” and “MATLAB Classes” for more information.

Note Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “What Is a MATLAB Function Block?” in the Simulink documentation.

System Objects in the MATLAB System Block

Using the MATLAB System block, you can include in a Simulink model individual System objects that you create with a class definition file. The model can then generate embeddable code. For more information, see “What Is the MATLAB System Block?” in the Simulink documentation.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

System Objects in Simulink

In this section...
“System Objects in the MATLAB Function Block” on page 4-17
“System Objects in the MATLAB System Block” on page 4-17

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using the MATLAB Function block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Objects in the MATLAB System Block

You can include individual System objects that you create with a class definition file into Simulink using the MATLAB System block. This provides one way to add your own algorithm blocks into your Simulink models. For information, see “System Object Integration” in the Simulink documentation.

System Object Methods

In this section...
“What Are System Object Methods?” on page 4-18
“The Step Method” on page 4-18
“Common Methods” on page 4-19

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The step method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object.

For more information about the `step` method and other available methods, see the descriptions in “Common Methods” on page 4-19.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
<code>step</code>	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the <code>step</code> method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The <code>step</code> method returns regular MATLAB variables. Example: <code>Y = step(H,X)</code>
<code>release</code>	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the <code>release</code> method instead of a destructor.
<code>reset</code>	Resets the internal states of the object to the initial values for that object
<code>getNumInputs</code>	Returns the number of inputs (excluding the object itself) expected by the <code>step</code> method. This number varies for an object depending on whether any properties enable additional inputs.
<code>getNumOutputs</code>	Returns the number of outputs expected from the <code>step</code> method. This number varies for an object depending on whether any properties enable additional outputs.

Method	Description
<code>getDiscreteState</code>	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the <code>step</code> method on it or after you have released the object), the states are empty. If the object has no discrete states, <code>getDiscreteState</code> returns an empty structure.
<code>clone</code>	Creates another object of the same type with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object is locked.
<code>isDone</code>	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns <code>false</code> .
<code>info</code>	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design in MATLAB Using System Objects

In this section...

“Create Components for Your System” on page 4-21

“Configure Components for Your System” on page 4-22

“Assemble Components to Create Your System” on page 4-23

“Run Your System” on page 4-25

“Reconfigure Your System During Runtime” on page 4-25

Create Components for Your System

This example shows how to create components for a system that processes a long stream of audio data. The data is read from a file, filtered, and then played.

A System object is a component you can use to create your system in MATLAB. System objects support fixed- or variable-size data. *Variable-size data* is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at initialization time, and therefore, cannot change at run time.

Many System objects are predefined in the software. You can also create your own System objects (see “Define New System Objects”).

The particular predefined components you need are:

- `dsp.AudioFileReader` — Read the file of audio data
- `dsp.FIRFilter` — Filter the audio data
- `dsp.AudioPlayer` — Play the filtered audio data

First, you create the component objects, using default property settings:

```
audioIn = dsp.AudioFileReader;  
filtLP = dsp.FIRFilter;  
audioOut = dsp.AudioPlayer;
```

Next, you configure each System object for your system. See “Configure Components for Your System” on page 4-22. Alternately, if desired, you can “Create and Configure Components at the Same Time” on page 4-23.

Configure Components for Your System

When to Configure Components

If you did not set an object’s properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See “Reconfigure Your System During Runtime” on page 4-25 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object’s handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure the components for your system by setting the component objects’ properties.

Use this procedure if you have created your components as described in “Create Components for Your System” on page 4-21. If you have not yet created your components, use the procedure in “Create and Configure Components at the Same Time” on page 4-23

For the file reader object, specify the file to read and set the output data type.

```
audioIn.FileName = 'speech_dft_8kHz.wav';  
audioIn.OutputDataType = 'single';
```

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

```
filtLP.Numerator = fir1(160,.15);
```

For the audio player object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut.SampleRate = audioIn.SampleRate;
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...  
    'OutputDataType','single')
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

After you create the components, you can assemble them in your system. See “Assemble Components to Create Your System” on page 4-23.

Assemble Components to Create Your System

- “Connect Inputs and Outputs” on page 4-24

- “Code for the Whole System” on page 4-24

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object’s `step` method as the input to another object’s `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see “System Object Methods” on page 4-18.

Code for the Whole System

This example shows how to write the full code for reading, filtering, and playing a file of audio data.

You can type this code on the command line or put it into a program file.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
    'OutputDataType','single');
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);

while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);  % Filter the data
    step(audioOut,y);        % Play the filtered data
end
```


The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system. See “Run Your System” on page 4-25.

Run Your System

- “How to Run Your System” on page 4-25
- “What You Cannot Change While Your System Is Running” on page 4-25

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see “Reconfigure Your System During Runtime” on page 4-25.

Reconfigure Your System During Runtime

- “When Can You Change Component Properties?” on page 4-26

- “Change a Tunable Property in Your System” on page 4-26
- “Change Input Complexity or Dimensions” on page 4-26

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object’s reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change a Tunable Property in Your System

This example shows how to change a tunable property.

You can change the filter type to a high-pass filter as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as at the next iteration of the while loop).

```
reset(audioIn);                % Reset audio file
filtLP.Numerator = fir1(160,0.15,'high');
while ~isDone(audioIn)
    audio = step(audioIn);      % Read audio source file
    y = step(filtLP,audio);     % Filter the data
    step(audioOut,y);          % Play the filtered data
end
```

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

System Design in Simulink Using System Objects

In this section...
“Define New Kinds of System Objects for Use in Simulink” on page 4-28
“Test New System Objects in MATLAB” on page 4-34
“Add System Objects to Your Simulink Model” on page 4-35

Define New Kinds of System Objects for Use in Simulink

This example shows the general steps to create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter and is similar to the System Identification Using MATLAB System Blocks Simulink example.

A System object is a component you can use to create your system in MATLAB. You can write the code in MATLAB and use that code to create a block in Simulink. To define your own System object, you write a class definition file, which is a text-based MATLAB file that contains the code defining your object. See “System Object Integration” in the Simulink documentation. The LMS Adaptive Filter and Integer Delay blocks in this example model are each from a System object class definition file. These files are described below.

Define System Object with Block Customizations

- 1 Create a class definition text file to define your System object. This example creates a least mean squares (LMS) filter and includes customizations to the block icon and dialog appearance.

Note Instead of manually creating your class definition file, you can use the **New > System Object > Simulink Extension** menu option to open a template. This template includes customizations of the System object for use in the Simulink MATLAB System block. You edit the template file, using it as guideline, to create your own System object.

- 2** On the first line of the class definition file, specify the name of your System object and subclass from both `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon.
- 3** Add the appropriate basic System object methods to set up, reset, set the number of inputs and outputs, and run your algorithm. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.
 - Use the `setupImpl` method to perform one-time calculations and initialize variables.
 - Use the `stepImpl` method to implement the block's algorithm.
 - Use the `resetImpl` to reset the state properties or `DiscreteState` properties.
 - Use the `getNumInputsImpl` and `getNumOutputsImpl` methods to specify the number of inputs and outputs, respectively.
- 4** Add the appropriate `CustomIcon` methods to define the appearance of the MATLAB System block in Simulink. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.
 - Use the `getHeaderImpl` method to specify the title and description to display on the block dialog.
 - Use the `getPropertyGroupsImpl` method to specify groups of properties to display on the block dialog.
 - Use the `getIconImpl` method to specify the text to display on the block icon.
 - Use the `getInputNamesImpl` and `getOutputNamesImpl` methods to specify the labels to display for the block input and output ports.

The full class definition file for the least mean squares filter is:

```
classdef lmsSysObj < matlab.System &...  
    matlab.system.mixin.CustomIcon
```

```
%lmsSysObj Least mean squares (LMS) adaptive filtering.
%#codegen

properties
    % Mu Step size
    Mu = 0.005;
end

properties (Nontunable)
    % Weights Filter weights
    Weights = 0;
    % N Number of filter weights
    N = 32;
end

properties (DiscreteState)
    X;
    H;
end

methods(Access=protected)
    function setupImpl(obj, ~, ~)
        obj.X = zeros(obj.N,1);
        obj.H = zeros(obj.N,1);
    end

    function [y, e_norm] = stepImpl(obj,d,u)
        tmp = obj.X(1:obj.N-1);
        obj.X(2:obj.N,1) = tmp;
        obj.X(1,1) = u;
        y = obj.X'*obj.H;
        e = d-y;
        obj.H = obj.H + obj.Mu*e*obj.X;
        e_norm = norm(obj.Weights'-obj.H);
    end

    function resetImpl(obj)
        obj.X = zeros(obj.N,1);
        obj.H = zeros(obj.N,1);
    end
end
```

```

        function num = getNumInputsImpl(~)
            num = 2;
        end
        function num = getNumOutputsImpl(~)
            num = 2;
        end
    end

% Block icon and dialog customizations
methods(Static, Access=protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header(...
            'lmsSysObj', ...
            'Title', 'LMS Adaptive Filter');
    end

    function groups = getPropertyGroupsImpl
        upperGroup = matlab.system.display.SectionGroup(...
            'Title', 'General', ...
            'PropertyList', {'Mu'}); %#ok<*EMCA>

        lowerGroup = matlab.system.display.SectionGroup(...
            'Title', 'Coefficients', ...
            'PropertyList', {'Weights', 'N'}); %#ok<*EMCA>

        groups = [upperGroup, lowerGroup];
    end
end

methods(Access=protected)
    function icon = getIconImpl(~)
        icon = sprintf('LMS Adaptive\nFilter');
    end
    function [in1name, in2name] = getInputNamesImpl(~)
        in1name = 'Desired';
        in2name = 'Actual';
    end
    function [out1name, out2name] = getOutputNamesImpl(~)
        out1name = 'Output';
    end
end

```

```
        out2name = 'EstError';
    end
end
end
```

Define System Object with Nondirect Feedthrough

- 1** Create a class definition text file to define your System object. This example creates an integer delay and includes customizations to the block icon. It implements a System object that you can use for nondirect feedthrough. See “Use System Objects in Feedback Loops” for more information.
- 2** On the first line of the class definition file, subclass from `matlab.System`, `matlab.system.mixin.CustomIcon`, and `matlab.system.mixin.Nondirect`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon. The `Nondirect` mixin enables the methods that let you specify how the block is updated and what it outputs.
- 3** Add the appropriate basic System object methods to set up and reset the object and set and validate the properties. Since this object supports nondirect feedthrough, you do not implement the `stepImpl` method. You implement the `updateImpl` and `outputImpl` methods instead. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.
 - Use the `setupImpl` method to initialize some of the object’s properties.
 - Use the `resetImpl` to reset the property states.
 - Use the `validatePropertiesImpl` to check that the property values are valid.
- 4** Add the following `Nondirect` mixin class methods instead of the `stepImpl` method to specify how the block updates its state and its output. See the reference pages and the full class definition file below for the implementation of each of these methods.
 - Use the `outputImpl` method to implement code to calculate the block output.

- Use the `updateImpl` method to implement code to update the block's internal states.
 - Use the `isInputDirectFeedthroughImpl` to specify that the block is not direct feedthrough. Its inputs do not directly affect its outputs.
- 5** Add the `getIconImpl` method to define the block icon when it is used in Simulink via the MATLAB System block. See the reference page and the full class definition file below for the implementation of this method.

The full class definition file for the delay is:

```
classdef intDelaySysObj < matlab.System &...
    matlab.system.mixin.Nondirect &...
    matlab.system.mixin.CustomIcon
%intDelaySysObj Delay input by specified number of samples.
%#codegen

properties
    %InitialOutput Initial output
    InitialOutput = 0;
end

properties (Nontunable)
    % NumDelays Number of delays
    NumDelays = 1;
end

properties(DiscreteState)
    PreviousInput;
end

methods(Access=protected)
    function setupImpl(obj, ~)
        obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
    end

    function [y] = outputImpl(obj, ~)
        % Output does not directly depend on input
        y = obj.PreviousInput(end);
    end
end
```

```
function updateImpl(obj, u)
    obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
end

function flag = isInputDirectFeedthroughImpl(~,~)
    flag = false;
end

function validatePropertiesImpl(obj)
    if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
        error('Number of delays must be positive non-zero scalar value.
    end
    if (numel(obj.InitialOutput)>1)
        error('Initial output must be scalar value. ');
    end
end

function resetImpl(obj)
    obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
end

function icon = getIconImpl(~)
    icon = sprintf('Integer\nDelay');
end
end
end
```

Test New System Objects in MATLAB

- 1 Create an instance of your new System object. For example, create an instance of the `lmsSysObj`.

```
s = lmsSysObj;
```

- 2 Run the `step` method on the object multiple times with different inputs. This tests for syntax errors and other possible issues before you add it to Simulink. For example,

```
desired = 0;
actual = 0.2;
```

```
step(s,desired,actual);
```

Add System Objects to Your Simulink Model

- 1** Add your System objects to your Simulink model by using the MATLAB System block as described in “Mapping System Objects to Block Dialog Box”.
- 2** Add other Simulink blocks, connect them, and configure any needed parameters to complete your model as described in the Simulink documentation. See the System Identification for an FIR System Using MATLAB System Blocks Simulink example.
- 3** Run your model in the same way you run any Simulink model.