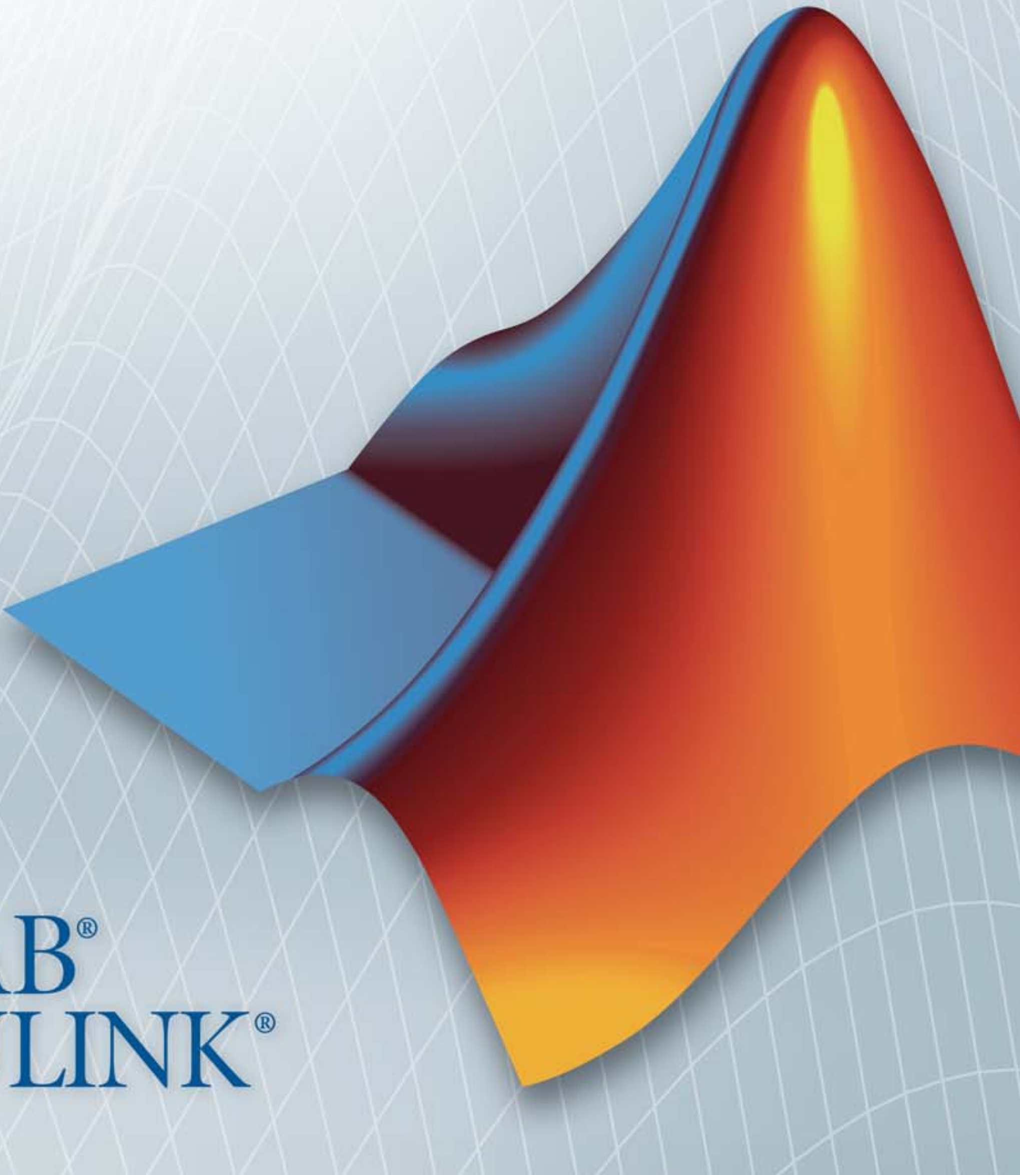
Communications System Toolbox™ 5

Getting Started Guide



MATLAB®
&SIMULINK®

MathWorks®

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

Revision History

April 2011	First printing	New for Version 5.0 (Release 2011a)

Contents

3 System Objects

Introduction

Communications System Toolbox Product Overview

Communications System Toolbox™ provides algorithms and tools for the design, simulation, and analysis of communications systems. These capabilities are provided as MATLAB® functions, MATLAB System objects, and Simulink® blocks. The system toolbox includes algorithms for source coding, channel coding, interleaving, modulation, equalization, synchronization, and channel modeling. Tools are provided for bit error rate analysis, generating eye and constellation diagrams, and visualizing channel characteristics. The system toolbox also provides adaptive algorithms that let you model dynamic communications systems that use OFDM, OFDMA, and MIMO techniques. Algorithms support fixed-point data arithmetic and C or HDL code generation.

The key features of this product include:

- Algorithms available as MATLAB functions, MATLAB system objects, and Simulink blocks
- Algorithms for designing the physical layer of communications systems, including source coding, channel coding, interleaving, modulation, channel models, equalization, and synchronization
- Visualization tools, including eye diagrams, constellations, and channel scattering functions
- Graphical tool for comparing the bit error rate of a system with analytical results
- Channel models, including AWGN, Multipath Rayleigh Fading, Rician Fading, COST 207, GSM/EDGE, HF ionospheric, and MIMO
- Interactive tool for visualizing time-varying communications channels
- Basic RF impairments, including nonlinearity, phase noise, thermal noise, and phase and frequency offsets
- Support for fixed-point modeling and C and HDL code generation

Installing the Communications System Toolbox Software and Documentation

Before you begin working, you must install the product on your computer.

Installing the Communications System Toolbox Software

The Communications System Toolbox software follows the same installation procedure as the MATLAB toolboxes. See the MATLAB installation documentation for instructions.

Installing Online Documentation

Installing the documentation is part of the installation process:

- Installation from a DVD — Start the MathWorks® installer. When prompted, select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.
- Installation from a Web download — If you update the Communications System Toolbox software using a Web download and you want to view the documentation with the MATLAB Help browser, you must install the documentation on your hard drive.

  Download the files from the Web. Then, start the installer, and select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.

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/commblockset/requirements.html`.

Related Products

MathWorks provides several products that are relevant to the kinds of tasks you can perform with Communications System Toolbox software.

For more information about any of these products, see either

- The online documentation for that product if it is installed on your system
- The MathWorks Web site, at `http://www.mathworks.com/communications-systems/`.

Expected Background

This documentation assumes that you already have background knowledge in the subject of 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 appear in many topics.

For New Users

The discussion and examples in this chapter are aimed at new users. Continue reading this chapter and try out the examples. Then read those subsequent chapters that address the specific areas that concern you. When you find out which functions you want to use, refer to the online reference pages that describe those functions.

For Experienced Users

The online reference descriptions are probably the most relevant parts of this guide for you. Each reference description includes the function’s syntax as well as a complete explanation of its options and operation. Many reference descriptions also include examples, a description of the function’s algorithm, and references to additional reading material.

Product Demos

In this section...

Demos in the Help Browser

You can find interactive Communications System Toolbox demos in the MATLAB Help browser. This example shows you how to locate and open some typical demos:

1. To open the Help browser, type `doc` at the MATLAB command line.
2. Expand the **Communications System Toolbox** node in the Help browser, then the **Demos** node.

   There are two entries under the Communications System Toolbox **Demos** node:

   - **MATLAB Demos** — Expand this entry to see a categorical list of Communications System Toolbox demos that you can run in MATLAB.
   - **Simulink Demos** — Expand this entry to see a categorical list of block-based Communications System Toolbox demos that you can run in Simulink.

   You can find more demos for the Simulink software by typing `demo` at the MATLAB command line.

Demos on the Web

The MathWorks Web site contains demos that show you how to use Communications System Toolbox software. You can find these demos at `http://www.mathworks.com/communications-systems/demos.html`.

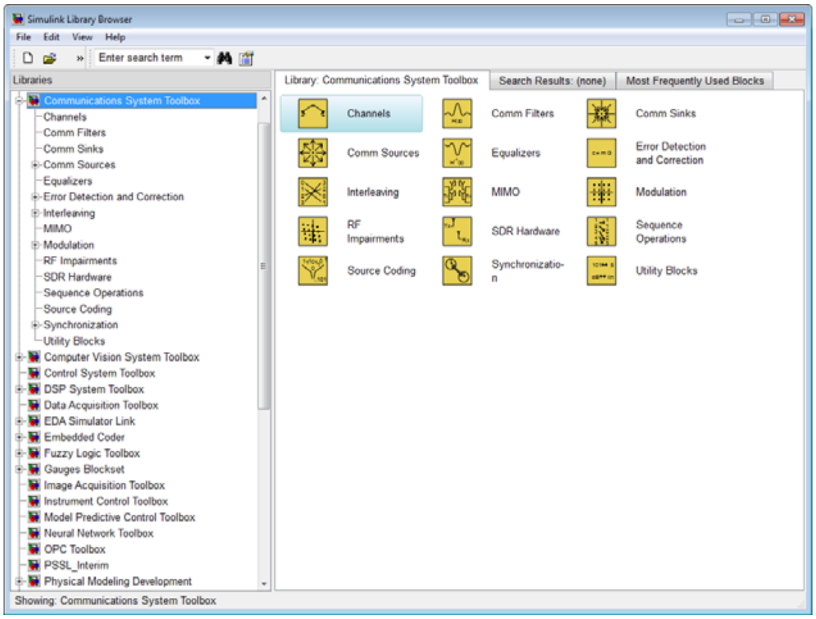
You can view these demos without having MATLAB or the DSP System Toolbox™ product installed on your system.

Demos on MATLAB Central

MATLAB Central contains files, including demos, contributed by users and developers of Communications System Toolbox, MATLAB, Simulink, and other products. Contributors submit their files to one of a list of categories. You can browse these categories to find submissions that pertain to Communications System Toolbox software or a specific problem that you want to solve. MATLAB Central is located at `http://www.mathworks.com/matlabcentral/`.

Accessing 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.

System Simulation

Compute BER for a QAM System with AWGN and Phase Noise Using Simulink

In this section...

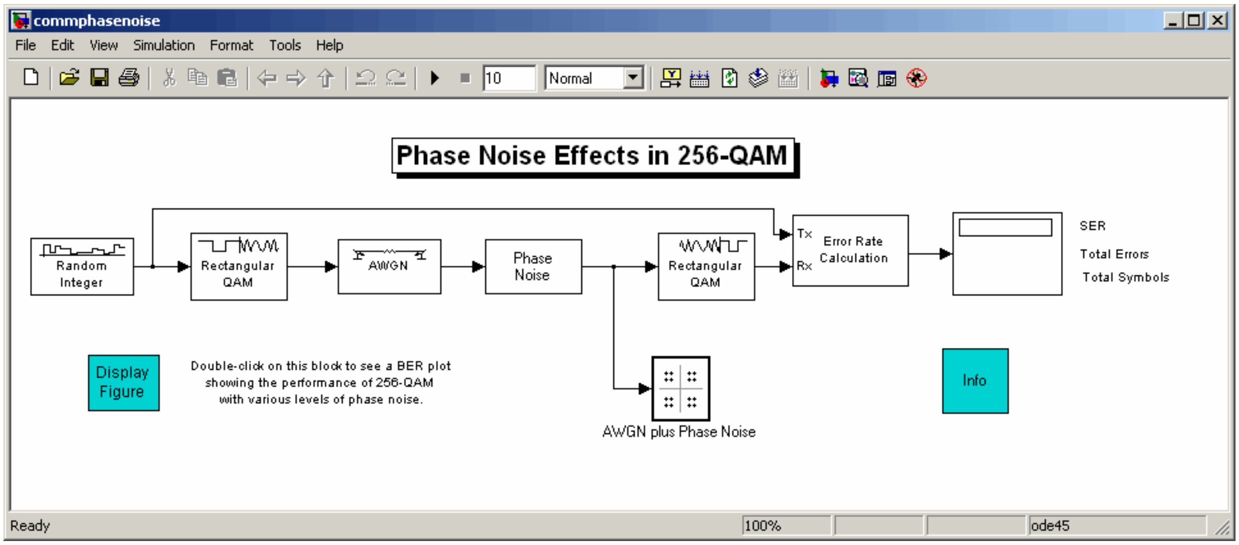
Section Overview

This section describes a demo model of a communications system that comes with Communications System Toolbox software. 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.

The section takes you through some key elements of working with this model.

Opening the Model

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


commphasenoise
File Edit View Simulation Format Tools Help
10
Normal
Phase Noise Effects in 256-QAM
Random Integer
Rectangular QAM
AWGN
Phase Noise
Rectangular QAM
Tx
Rx
Error Rate Calculation
SER
Total Errors
Total Symbols
Display Figure
Double-click on this block to see a BER plot showing the performance of 256-QAM with various levels of phase noise.
AWGN plus Phase Noise
Info
Ready
100%
ode45

Overview of the Model

The Simulink 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.

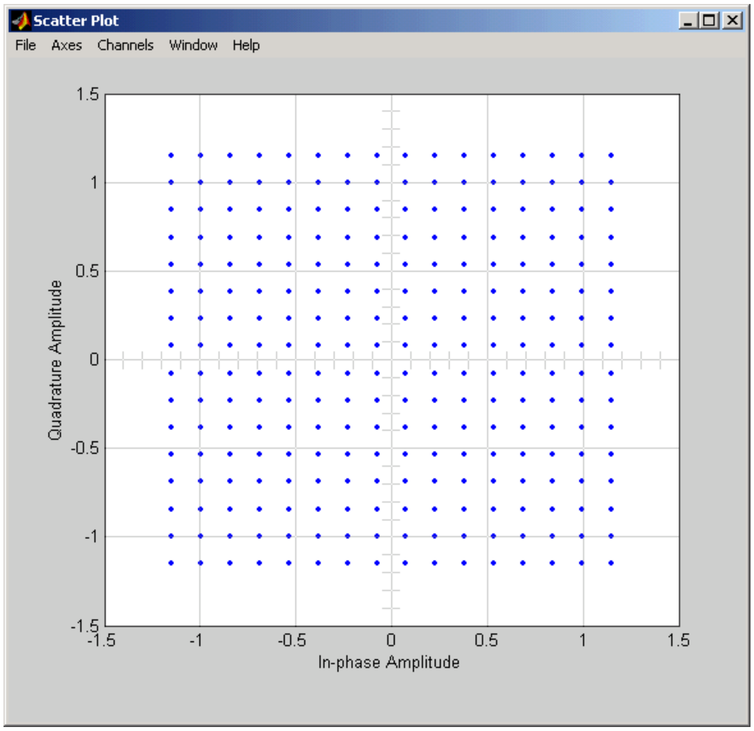
In addition, the following blocks in the model help you interpret the simulation:

- The Discrete-Time Scatter Plot Scope 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 and Simulink applications. 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.

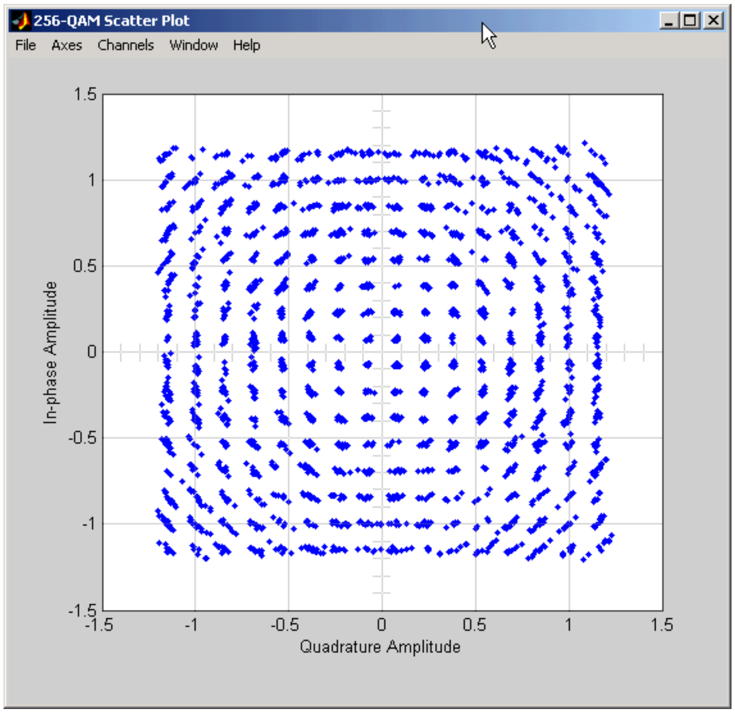


Constellation for 256-ary QAM

Run a Simulation

To run a simulation, select **Simulation > Start** from 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.

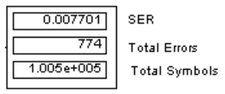


Scatter Plot of Signal Plus Noise

The points in the scatter plot do not lie exactly on the constellation shown in the figure Constellation for 256-ary QAM on page 2-5 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.



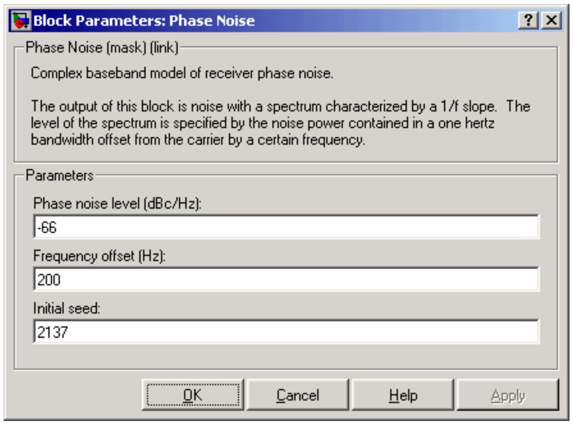
Error Rate Display

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.



Block Parameters: Phase Noise

Phase Noise (mask) (link)

Complex baseband model of receiver phase noise.

The output of this block is noise with a spectrum characterized by a 1/f slope. The level of the spectrum is specified by the noise power contained in a one hertz bandwidth offset from the carrier by a certain frequency.

Parameters

Phase noise level (dBc/Hz): -66

Frequency offset (Hz): 200

Initial seed: 2137

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 = 2`. Setting parameters in the Command Window is convenient if you need to run multiple simulations with different parameter values. See the section .

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.

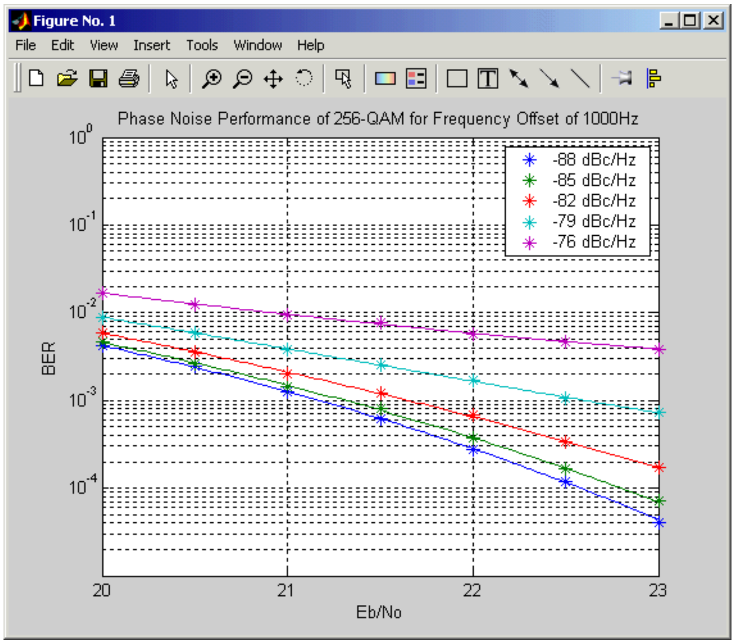
You can experiment with the model by changing these or other parameters and then running a simulation. For example,

- 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 **Es/No** parameters. describes how to do this with a MATLAB script, using variables for the parameters.

More Demos

You can find Communications System Toolbox demos in the MATLAB Help browser. For more information, see “Demos in the Help Browser” on page 1-7 software by typing `demo` at the MATLAB command line.

Compute BER for a QAM System with AWGN Using MATLAB

In this section...

Section Overview

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.

While the later chapters of this document discuss various features in more depth, 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 first example addresses the following problem:

Problem Process a binary data stream using a communication system that consists of a baseband modulator, channel, and demodulator. Compute the system's bit error rate (BER). Also, display the transmitted and received signals in a scatter plot.

The following table indicates the key tasks in solving the problem, along with relevant Communications System Toolbox functions. The solution arbitrarily chooses baseband 16-QAM (quadrature amplitude modulation) as the modulation scheme and AWGN (additive white Gaussian noise) as the channel model.

Task	Function or Method
Generate a random binary data stream	`randint`
Modulate using 16-QAM	`modulate` method on `modem.qammod` object
Add white Gaussian noise	`awgn`
Create a scatter plot	`scatterplot`
Demodulate using 16-QAM	`modulate` method on `modem.qamdemod` object
Compute the system's BER	`biterr`

Solution of Problem

The discussion below describes each step in more detail, introducing MATLAB code along the way. To view all the code in one editor window, enter the following in the MATLAB Command Window.

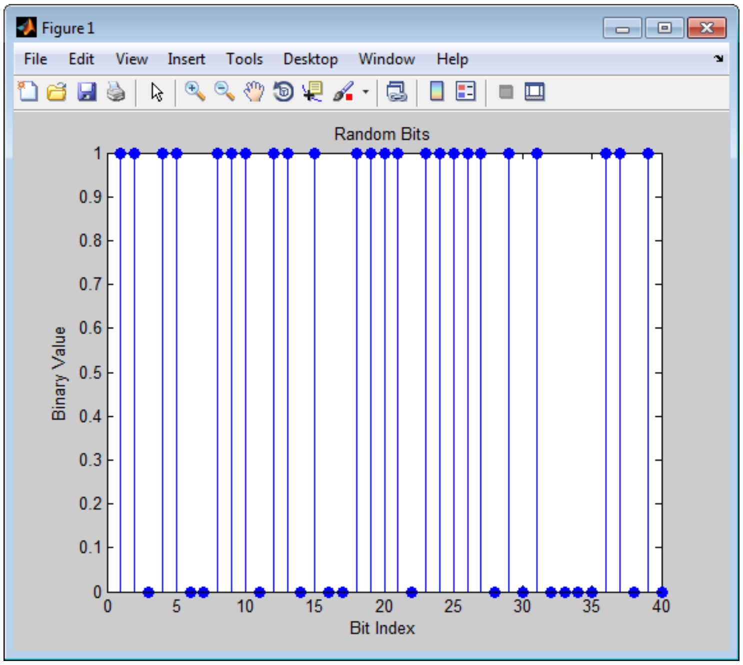
```
edit commdoc_mod
```

1. Generate a Random Binary Data Stream. The conventional format for representing a signal in MATLAB is a vector or matrix. This example uses the `randint` function to create a column vector that lists the successive 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. Your plot might look different because the example uses random numbers. Notice the use of the colon (:) operator in MATLAB to select a portion of the vector. For more information about this syntax, see The Colon Operator in the MATLAB documentation set.

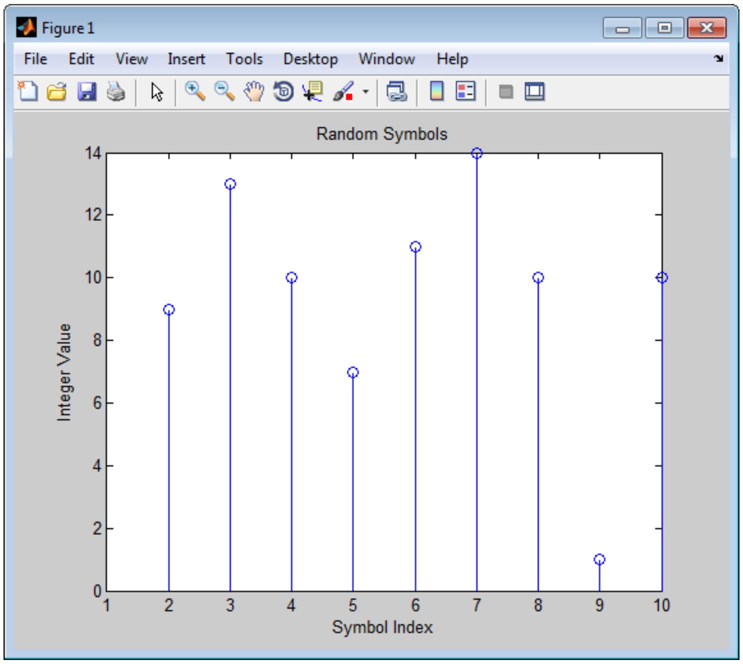
```
%% Setup
% Define parameters.
M = 16;                 % Size of signal constellation
k = log2(M);            % Number of bits per symbol
n = 3e4;                % Number of bits to process
nsamp = 1;              % Oversampling rate
hMod = modem.qammod(M); % Create a 16-QAM modulator

%% Signal Source
% Create a binary data stream as a column vector.
x = randint(n,1); % Random binary data stream

% Plot first 40 bits in a stem plot.
stem(x(1:40),'filled');
title('Random Bits');
xlabel('Bit Index'); ylabel('Binary Value');
```



2. **Prepare to Modulate.** The `modem.qammod` object implements an M-ary QAM modulator, M being 16 in this example. It is configured to receive integers between 0 and 15 rather than 4-tuples of bits. Therefore, you must preprocess the binary data stream `x` before using the `modulate` method of the object. In particular, you arrange each 4-tuple of values from `x` across a row of a matrix, using the `reshape` function in MATLAB, and then apply the `bi2de` function to convert each 4-tuple to a corresponding integer. (The `.'` characters after the `reshape` command form the unconjugated array transpose operator in MATLAB. For more information about this and the similar `'` operator, see Reshaping a Matrix in the MATLAB documentation set.)

```
%% Bit-to-Symbol Mapping
% Convert the bits in x into k-bit symbols.
xsym = bi2de(reshape(x,k,length(x)/k).','left-msb');

%% Stem Plot of Symbols
% Plot first 10 symbols in a stem plot.
figure; % Create new figure window.
```

```
stem(xsym(1:10));
title('Random Symbols');
xlabel('Symbol Index'); ylabel('Integer Value');
```



3. Modulate Using 16-QAM. Having defined `xsym` as a column vector containing integers between 0 and 15, you can use the `modulate` method of the `modem.qammod` object to modulate `xsym` using the baseband representation. Recall that `M` is 16, the alphabet size.

```
%% Modulation
y = modulate(modem.qammod(M),xsym); % Modulate using 16-QAM.
```

The result is a complex column vector whose values are in the 16-point QAM signal constellation. A later step in this example will show what the constellation looks like.

To learn more about modulation functions, see . Also, note that the `modulate` method of the `modem.qammod` object does not apply any pulse shaping. To extend this example to use pulse shaping, see "Pulse Shaping Using a Raised

Cosine Filter" on page 2-24. For an example that uses rectangular pulse shaping with PSK modulation, see `basicsimdemo`.

4. Add White Gaussian Noise. Applying the `awgn` function to the modulated signal adds white Gaussian noise to it. The ratio of bit energy to noise power spectral density, E_b/N_0, is arbitrarily set at 10 dB.

The expression to convert this value to the corresponding signal-to-noise ratio (SNR) involves `k`, the number of bits per symbol (which is 4 for 16-QAM), and `nsamp`, the oversampling factor (which is 1 in this example). The factor `k` is used to convert E_b/N_0 to an equivalent E_s/N_0, which is the ratio of *symbol* energy to noise power spectral density. The factor `nsamp` is used to convert E_s/N_0 in the symbol rate bandwidth to an SNR in the sampling bandwidth.

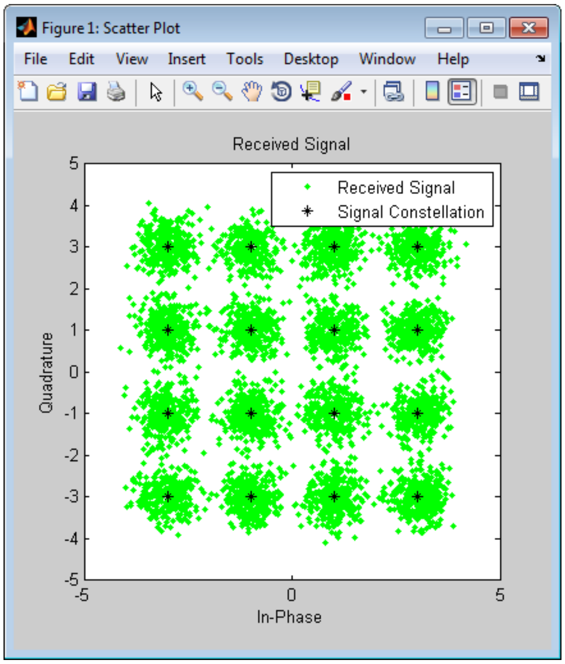
Note The definitions of `ytx` and `yrx` and the `nsamp` term in the definition of `snr` are not significant in this example so far, but will make it easier to extend the example later to use pulse shaping.

```
%% Transmitted Signal
ytx = y;

%% Channel
% Send signal over an AWGN channel.
EbNo = 10; % In dB
snr = EbNo + 10*log10(k) - 10*log10(nsamp);
ynoisy = awgn(ytx,snr,'measured');

%% Received Signal
yrx = ynoisy;
```

To learn more about `awgn` and other channel functions, see .

5. Create a Scatter Plot. Applying the `scatterplot` function to the transmitted and received signals shows what the signal constellation looks like and how the noise distorts the signal. In the plot, the horizontal axis is the in-phase component of the signal and the vertical axis is the quadrature component. The code below also uses the `title`, `legend`, and `axis` functions in MATLAB to customize the plot.

```
%% Scatter Plot
% Create scatter plot of noisy signal and transmitted
% signal on the same axes.
h = scatterplot(yrx(1:nsamp*5e3),nsamp,0,'g.');
hold on;
scatterplot(ytx(1:5e3),1,0,'k*',h);
title('Received Signal');
legend('Received Signal','Signal Constellation');
axis([-5 5 -5 5]); % Set axis ranges.
hold off;
```



To learn more about `scatterplot`, see .

6. Demodulate Using 16-QAM. Applying the `demodulate` method of the `modem.qamdemod` object to the received signal demodulates it. The result is a column vector containing integers between 0 and 15.

```
%% Demodulation
% Demodulate signal using 16-QAM.
```

```
zsym = demodulate(modem.qamdemod(M),yrx);
```

7. Convert the Integer-Valued Signal to a Binary Signal. The previous step produced `zsym`, a vector of integers. To obtain an equivalent binary signal, use the `de2bi` function to convert each integer to a corresponding binary 4-tuple along a row of a matrix. Then use the `reshape` function to arrange all the bits in a single column vector rather than a four-column matrix.

```
%% Symbol-to-Bit Mapping
% Undo the bit-to-symbol mapping performed earlier.
z = de2bi(zsym,'left-msb'); % Convert integers to bits.
% Convert z from a matrix to a vector.
z = reshape(z.',numel(z),1);
```

8. Compute the System's BER. Applying the `biterr` function to the original binary vector and to the binary vector from the demodulation step above yields the number of bit errors and the bit error rate.

```
%% BER Computation
% Compare x and z to obtain the number of errors and
% the bit error rate.
[number_of_errors,bit_error_rate] = biterr(x,z)
```

The statistics appear in the MATLAB Command Window. Your results might vary because the example uses random numbers.

```
number_of_errors =

    71


bit_error_rate =

    0.0024
```

To learn more about `biterr`, see .

Plot Signal Constellations

The example in the previous section 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 between 0 and 15 the modulator mapped to a given constellation point. This section addresses the following problem:

Problem Plot a 16-QAM signal constellation with annotations that indicate the mapping from integers to constellation points.

The solution uses the `scatterplot` function to create the plot and the `text` function in MATLAB to create the annotations.

Solution of Problem

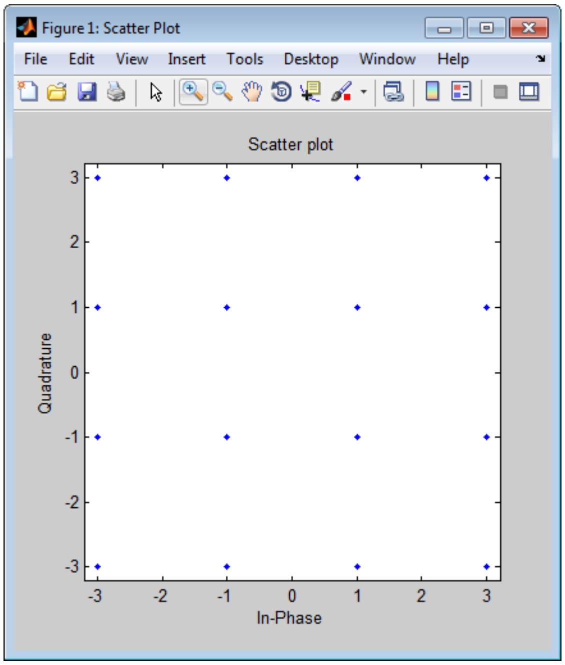
To view a completed MATLAB file for this example, enter `edit commdoc_const` in the MATLAB Command Window.

1. Find All Points in the 16-QAM Signal Constellation. The `Constellation` property of the `modem.qammod` object contains all points in the 16-QAM signal constellation.

```
M = 16; % Number of points in constellation
h=modem.qammod(M); % Modulator object
mapping=h.SymbolMapping; % Symbol mapping vector
pt = h.Constellation; % Vector of all points in constellation
```

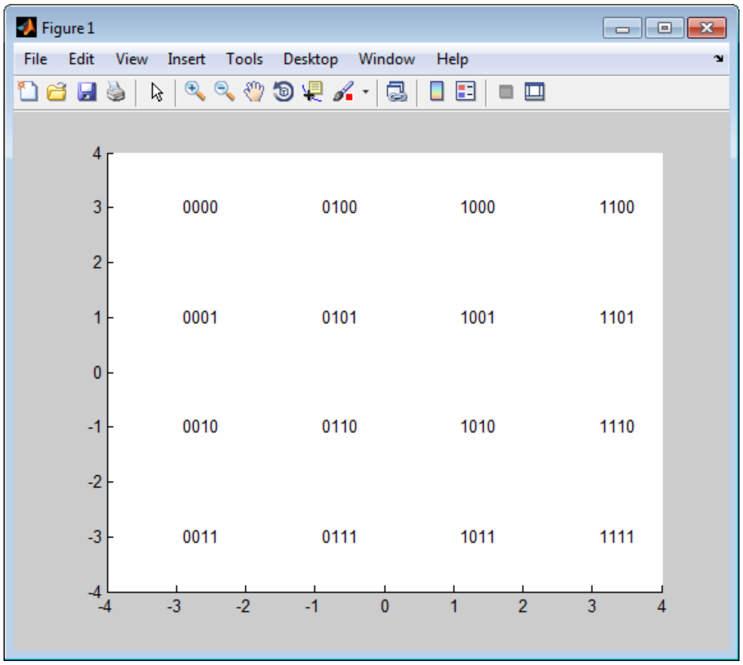
2. Plot the Signal Constellation. The `scatterplot` function plots the points in `pt`.

```
% Plot the constellation.
scatterplot(pt);
```



3. Annotate the Plot to Indicate the Mapping. To annotate the plot to show the relationship between `mapping` and `pt`, use the `text` function to place a number in the plot beside each constellation point. The coordinates of the annotation are near the real and imaginary parts of the constellation point, but slightly offset to avoid overlap. The text of the annotation comes from the binary representation of `mapping`. (The `dec2bin` function in MATLAB produces a string of digit characters, while the `de2bi` function used in the last section produces a vector of numbers.)

```
% Include text annotations that number the points.
text(real(pt)+0.1,imag(pt),dec2bin(mapping));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.
```

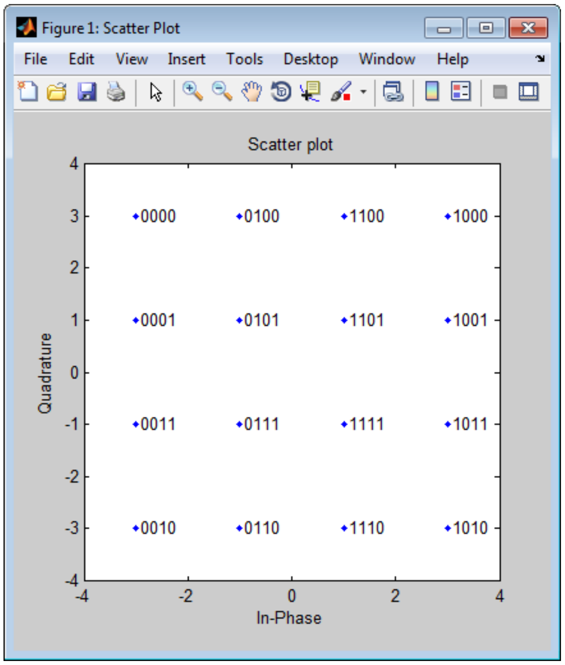


Binary-Coded 16-QAM Signal Constellation

Examine the Plot

In the plot above, notice that 0001 and 0010 correspond to adjacent constellation points on the left side of the diagram. Because these binary representations differ by two bits, the adjacency indicates that the `modem.qammod` object did *not* use a Gray-coded signal constellation. (That is, if it were a Gray-coded signal constellation, then the annotations for each pair of adjacent points would differ by one bit.)

By contrast, the constellation below is one example of a Gray-coded 16-QAM signal constellation.



Gray-Coded 16-QAM Signal Constellation

The only difference, compared to the previous example, is that you configure `modem.qammod` object to use a Gray-coded constellation.

```
%% Modified Plot, With Gray Coding
M = 16; % Number of points in constellation
h = modem.qammod('M',M,'SymbolOrder','Gray'); % Modulator object
mapping = h.SymbolMapping; % Symbol mapping vector
pt = h.Constellation; % Vector of all points in constellation

scatterplot(pt); % Plot the constellation.

% Include text annotations that number the points.
text(real(pt)+0.1,imag(pt),dec2bin(mapping));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.%% Modified Pl
M = 16; % Number of points in constellation
h = modem.qammod('M',M,'SymbolOrder','Gray'); % Modulator object
mapping = h.SymbolMapping; % Symbol mapping vector
```

```
pt = h.Constellation; % Vector of all points in constellation

scatterplot(pt); % Plot the constellation.

% Include text annotations that number the points.
text(real(pt)+0.1,imag(pt),dec2bin(mapping));
axis([-4 4 -4 4]); % Change axis so all labels fit in plot.
```

Pulse Shaping Using a Raised Cosine Filter

This section further extends the example by addressing the following problem:

Problem Modify the Gray-coded modulation example so that it uses a pair of square root raised cosine filters to perform pulse shaping and matched filtering at the transmitter and receiver, respectively.

The solution uses the `rcosine` function to design the square root raised cosine filter and the `rcosflt` function to filter the signals. Alternatively, you can use the `rcosflt` function to perform both tasks in one command; see or the `rcosdemo` demonstration for more details.

Solution of Problem

This solution modifies the code from `commdoc_gray.m`. To view the original code in an editor window, enter the following command in the MATLAB Command Window.

```
edit commdoc_gray
```

To view a completed MATLAB file for this example, enter `edit commdoc_rrc` in the MATLAB Command Window.

1. Define Filter-Related Parameters. In the `Setup` section of the example, replace the definition of the oversampling rate, `nsamp`, with the following.

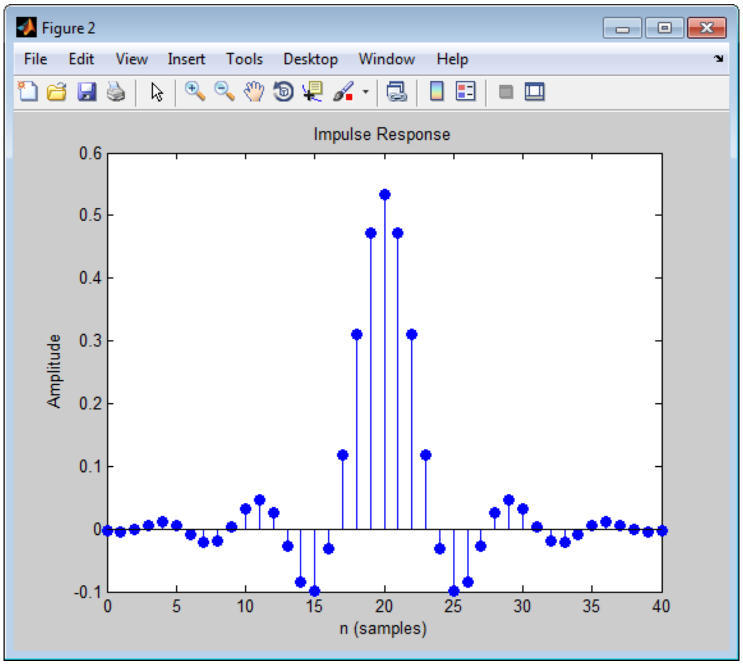
```
nsamp = 4; % Oversampling rate
```

Also, define other key parameters related to the filter by inserting the following after the `Modulation` section of the example and before the `Transmitted signal` section.

```
%% Filter Definition
% Define filter-related parameters.
filtorder = 40; % Filter order
delay = filtorder/(nsamp*2); % Group delay (# of input samples)
rolloff = 0.25; % Rolloff factor of filter
```

2. Create a Square Root Raised Cosine Filter. To design the filter and plot its impulse response, insert the following commands after the commands you added in the previous step.

```
% Create a square root raised cosine filter.
rrcfilter = rcosine(1,nsamp,'fir/sqrt',rolloff,delay);

% Plot impulse response.
figure; impz(rrcfilter,1);
```

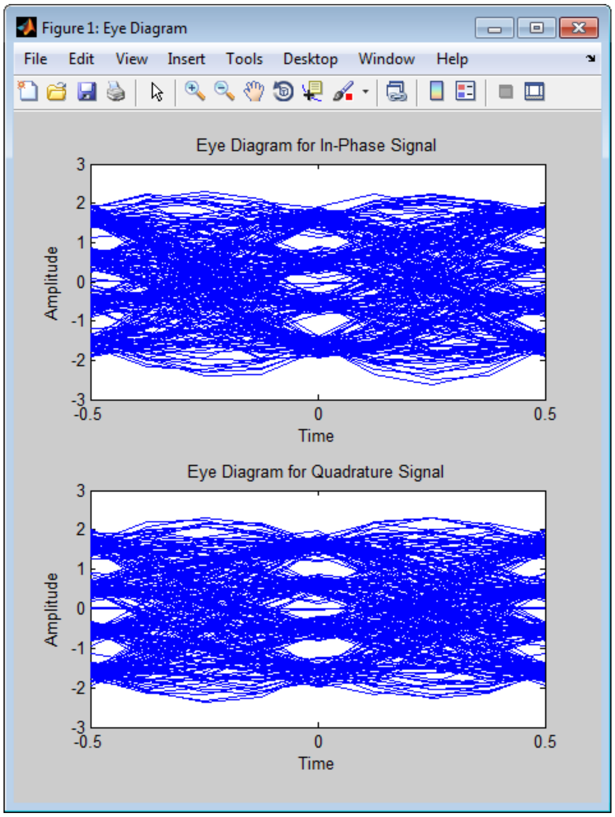


3. Filter the Modulated Signal. To filter the modulated signal, replace the `Transmitted Signal` section with following.

```
%% Transmitted Signal
% Upsample and apply square root raised cosine filter.
ytx = rcosflt(y,1,nsamp,'filter',rrcfilter);

% Create eye diagram for part of filtered signal.
eyediagram(ytx(1:2000),nsamp*2);
```

The rcosflt command internally upsamples the modulated signal, y, by a factor of nsamp, pads the upsampled signal with zeros at the end to flush the filter at the end of the filtering operation, and then applies the filter.

The eyediagram command 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.



To learn more about eyediagram, see .

4. Filter the Received Signal. To filter the received signal, replace the `Received Signal` section with the following.

```
%% Received Signal
% Filter received signal using square root raised cosine filter.
yrx = rcosflt(ynoisy,1,nsamp,'Fs/filter',rrcfilter);
yrx = downsample(yrx,nsamp); % Downsample.
yrx = yrx(2*delay+1:end-2*delay); % Account for delay.
```

These commands apply the same square root raised cosine filter that the transmitter used earlier, and then downsample the result by a factor of `nsamp`.

The last command removes the first `2*delay` symbols and the last `2*delay` symbols in the downsampled signal because they represent the cumulative delay of the two filtering operations. Now `yrx`, which is the input to the demodulator, and `y`, 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 important to compare two vectors that have the same size.

5. Adjust the Scatter Plot. For variety in this example, make the scatter plot show the received signal before and after the filtering operation. To do this, replace the `Scatter Plot` section of the example with the following.

```
%% Scatter Plot
% Create scatter plot of received signal before and
% after filtering.
h = scatterplot(sqrt(nsamp)*ynoisy(1:nsamp*5e3),nsamp,0,'g.');
hold on;
scatterplot(yrx(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.
```

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

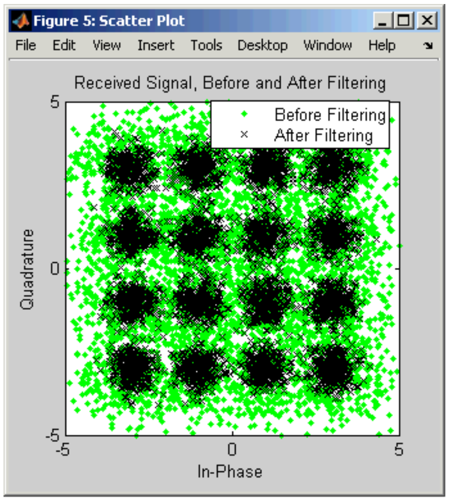


Figure 5: Scatter Plot
File Edit View Insert Tools Desktop Window Help
Received Signal, Before and After Filtering
Before Filtering
After Filtering
Quadrature
In-Phase
5
0
-5
-5
0
5

Use a Convolutional Code

This section further extends the example by addressing the following problem:

Problem Modify the previous example so that it includes convolutional coding and decoding, given the constraint lengths and generator polynomials of the convolutional code.

The solution uses the `convenc` and `vitdec` functions to perform encoding and decoding, respectively. It also uses the `poly2trellis` function to define a trellis that represents a convolutional encoder. To learn more about these functions, see .

See also `vitsimdemo` for an example of convolutional coding and decoding.

Solution of Problem

This solution modifies the code from "Pulse Shaping Using a Raised Cosine Filter" on page 2-24. To view the original code in an editor window, enter the following command in the MATLAB Command Window.

```
edit commdoc_rrc
```

To view a completed MATLAB file for this example, enter `edit commdoc_code` in the MATLAB Command Window.

1. Increase the Number of Symbols. Convolutional coding at this value of `EbNo` reduces the BER markedly. As a result, accumulating enough errors to compute a reliable BER requires you to process more symbols. In the `Setup` section, replace the definition of the number of bits, `n`, with the following.

```
n = 5e5; % Number of bits to process
```

Note The larger number of bits in this example causes it to take a noticeably longer time to run compared to the examples in previous sections.

---

2. Encode the Binary Data. To encode the binary data before mapping it to integers for modulation, insert the following after the `Signal Source` section of the example and before the `Bit-to-Symbol Mapping` section.

```
%% Encoder
% Define a convolutional coding trellis and use it
% to encode the binary data.
t = poly2trellis([5 4],[23 35 0; 0 5 13]); % Trellis
code = convenc(x,t); % Encode.
coderate = 2/3;
```

The `poly2trellis` command defines the trellis that represents the convolutional code that `convenc` uses for encoding the binary vector, `x`. The two input arguments in the `poly2trellis` command indicate the constraint length and generator polynomials, respectively, of the code. A diagram showing this encoder is in .

3. Apply the Bit-to-Symbol Mapping to the Encoded Signal. The bit-to-symbol mapping must apply to the encoded signal, `code`, not the original uncoded data. Replace the first definition of `xsym` (within the `Bit-to-Symbol Mapping` section) with the following.

```
% B. Do ordinary binary-to-decimal mapping.
xsym = bi2de(reshape(code,k,length(code)/k).','left-msb');
```

Recall that `k` is 4, the number of bits per symbol in 16-QAM.

4. Account for Code Rate When Defining SNR. Converting from E_b/N_0 to the signal-to-noise ratio requires you to account for the number of information bits per symbol. Previously, each symbol corresponded to `k` bits. Now, each symbol corresponds to `k*coderate` information bits. More concretely, three symbols correspond to 12 coded bits in 16-QAM, which correspond to 8 uncoded (information) bits, so the ratio of symbols to information bits is `8/3 = 4*(2/3) = k*coderate`.

Therefore, change the definition of `snr` (within the `Channel` section) to the following.

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

5. Decode the Convolutional Code. To decode the convolutional code before computing the error rate, insert the following after the entire `Symbol-to-Bit Mapping` section and just before the `BER Computation` section.

```
%% Decoder
% Decode the convolutional code.
tb = 16; % Traceback length for decoding
z = vitdec(z,t,tb,'cont','hard'); % Decode.
```

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 you invoke the function repeatedly (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. The delay is discussed further in “More About Delays” on page 2-32.

6. Account for Delay When Computing BER. The continuous operation mode of the Viterbi decoder incurs a delay whose duration in bits equals the traceback length, `tb`, times the number of input streams to the *encoder*. For this rate 2/3 code, the encoder has two input streams, so the delay is `2*tb` bits.

As a result, the first `2*tb` bits in the decoded vector, `z`, are just zeros. When computing the bit error rate, you should ignore the first `2*tb` bits in `z` and the last `2*tb` bits in the original vector, `x`. If you do not compensate for the delay, then the BER computation is meaningless because it compares two vectors that do not truly correspond to each other.

Therefore, replace the `BER Computation` section with the following.

```
%% BER Computation
% Compare x and z to obtain the number of errors and
% the bit error rate. Take the decoding delay into account.
decdelay = 2*tb; % Decoder delay, in bits
[number_of_errors,bit_error_rate] = ...
   biterr(x(1:end-decdelay),z(decdelay+1:end))
```

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"

System Objects

What Are System Objects?

In this section...

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

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

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. System objects support fixed-point arithmetic and C-code generation from MATLAB and Simulink. With System objects, you can optionally generate code to target the desktop or external hardware. You can use System objects in Simulink® models via the MATLAB Function block.

Note System objects do not support sparse matrices.

Create a System Object

To use System objects, you must first create an object. For example,

```
H = dsp.FFT        % Create default FFT object, H
```

```
% Create input data
Fs = 1000;           % Sampling frequency
T = 1/Fs;            % Sample time
L = 1024;            % Length of signal
t = (0:L-1)*T        % Time vector

% Sum of two sinusoids
X = 0.7*sin(2*pi*50*t.') + sin(2*pi*120*t.');
```

Change a System Object Property

To change the value of a property, use this format,

```
H.Normalize = true  % Set the Normalize property
```

The property values of the FFT object, `H`, are displayed. In general, you should set the object properties before you use the `step` method to run data through the object.

Run a System Object

To execute a system object, use the `step` method.

```
Y = step(H,X);       % Process input data, X
```

The output data from the `step` method is stored in `Y`, which, in this case, is the FFT of X.

Display Available System Objects

To see a list of all the System objects for a particular package, type `help <packagename>`. For example,

```
% DSP System Toolbox System objects
help dsp

 % Computer Vision System Toolbox  System objects
help vision

% Communications System Toolbox System objects
```

```
help comm
```

To display help for specific objects, properties, or methods, see “Find Help and Demos for System Objects” on page 3-14 .

Set Up a System Object

Create a New System Object

You must create a System object before using it. You can create the object at the MATLAB command line or within a program file. Your command-line code and programs can pass MATLAB variables into and out of System objects.

For general information about working with MATLAB objects, see *Object-Oriented Programming* in the MATLAB user documentation.

Syntax for Creating a System Object

The syntax for creating a System object, in this case, a digital filter object, with default property values is:

```
H = dsp.DigitalFilter
```

where

- `H` is the handle to the object.

  System objects are handle objects and follow handle semantics (e.g., when you call a method using the handle, it affects the original object, not a copy of that object). See "The Handle Superclass" for information on handle objects. See "Value or Handle Class — Which to Use" in the MATLAB user documentation for information on object handles.

- `dsp` is the package name for objects in the DSP System Toolbox product. *Packages* are libraries of System objects.

  Other package names are `vision`, which is in the Computer Vision System Toolbox product, and `comm`, which is in the Communications System Toolbox product.

- `DigitalFilter` is the object name.

Create Arrays of System Objects

You can create arrays that contain the same or different classes of System objects. This is convenient for running methods on multiple objects simultaneously. You can run only these methods on arrays of System objects.

- `clone`
- `getNumInputs`
- `getNumOutputs`
- `isLocked`
- `release`
- `reset`

Note You cannot run the `step` method on an array of System objects.

This example shows the syntax you use to create an array of three System objects.

```
hFilts = [dsp.CICDecimator,dsp.FIRDecimator,dsp.DigitalFilter];
```

Retreive System Object Property Values

System objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *Name-Value pair*. You can display the list of relevant property names and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handle>.<Name>`.

Example. This example retrieves and displays the `TransferFunction` property value for the previously created `DigitalFilter` object:

```
H.TransferFunction
```

Set System Object Property Values

You set the property values of a System object to model the desired algorithm.

Note When you use Name-Value pair syntax, the object sets property values in the order you list them. If you specify a dependent property value before its parent property, an error or warning may occur.

Set Properties for a New System Object. To set a property when you first create the object, use Name-Value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
H1 = dsp.DigitalFilter('CoefficientsSource','Input port')
```

where

- `H1` is the handle to the object
- `dsp` is the package name
- `DigitalFilter` is the object name
- `'CoefficientsSource'` is the property name
- `'Input port'` is the property value

Set Properties for an Existing System Object. To set a property after you have created an object, use either of the following syntaxes:

```
H1.CoefficientsSource = 'Property'
```

or

```
set(H1,'CoefficientsSource','Property')
```

Use Value-Only Inputs. Some object properties have no useful default values or must be specified every time you create an object. For these properties, you can specify only the value without specifying the corresponding property name. If you use value-only inputs, those inputs must be in a specific order, which is the same as the order in which the properties are displayed. Refer to the object reference page for details. For example,

```
H2 = dsp.FIRDecimator(3,[1 .5 1])
```

specifies the `DecimationFactor` as `3` and the `Numerator` as `[1 .5 1]`.

Process Data with System Objects

In this section...

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)`.

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 3-9.

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
`step`	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 `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. See "Understand System Object Modes" on page 3-12.
`clone`	Creates another object with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked. See "Understand System Object Modes" on page 3-12.
`reset`	Resets the internal states of the object to the initial values for that object
`isDone`	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 `false`.
`info`	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.
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.

Advantages of Using Methods

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 instances of an object, 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.

What are System Object Locking and Property Tunability?

In this section...

Understand System Object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an instance of an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

The object initializes and locks when it begins processing data. The typical way in which an object becomes locked is when the step method is called on that object. To determine if an object is locked, use the `isLocked` method. To unlock an object, use the `release` method. When the object is locked, you cannot change any of the following:

- Number of inputs or outputs
- Data type
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data, where the input size can vary. See "What Is Variable-Size Data?" for more information.)
- Value of any nontunable property
- Complexity of inputs from real to complex. (You can, however, change input complexity from complex to real without unlocking the object.)

These restrictions allow the object to maintain states and allocate memory appropriately.

Change Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular

System object property is tunable, see the corresponding reference page or use a command of this form:

```
help dsp.FFT.Normalize
```

where

- `dsp` is the package name.
- `FFT` is the object name.
- `Normalize` is the property name.

For information on locked and unlocked modes, see "Understand System Object Modes" on page 3-12.

Change System Object Input Complexity or Dimensions

During simulations you can change an input's complexity from complex to real, but not from real to complex. You cannot change any input complexity during code generation.

For objects that do not support variable-size input, if you change the input dimensions while the object is in locked mode, the object produces a warning and unlocks. The object then reinitializes the next time you call the step method. See the object's reference page for more information. You can change the value of a tunable property and the input size without a warning or error being produced. For all other changes at runtime, an error occurs.

Find Help and Demos for System Objects

Use Help Commands

Refer to the following resources for more information about System objects.

- Package help – `help dsp`, where `dsp` is a product package name
- Object help – `help dsp.FFT`, where `FFT` is the object name
- Documentation reference pages for an object – `doc dsp.FFT`
- Property help — `help dsp.FFT.Normalize`, where `Normalize` is the property name.
- Fixed-point property help – `dsp.FFT.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any object.
- Method help – `help dsp.FFT.step`, where `step` is the method name.

Find Demos

To view demos, go to the Help contents for the associated product. Under `Demos`, select `MATLAB demos`.

Use System Objects for Code Generation from MATLAB

In this section...

Considerations for Using System Objects in Generated Code

You can use System objects in code generated from MATLAB. To generate code, you must also have the MATLAB® Coder™ product. Using this product with System objects, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. System objects also support code generation using the MATLAB Function block in Simulink and the MATLAB Coder `codegen` function.

For general information on generating code, see

- *MATLAB Coder Getting Started Guide.*
- *Simulink® Coder™ User's Guide*
- *Embedded Coder™ Getting Started Guide*

The following example, which uses System objects, shows the key factors to consider, such as using persistent variables, passing property values, and extrinsic functions, when you make MATLAB code suitable for code generation.

```
function lmssystemidentification
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.

    persistent hlms hfilt;
```

```
        % 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);

            % Create system (an FIR filter) to be identified.

            hfilt = dsp.DigitalFilter(...
                        'TransferFunction', 'FIR (all zeros)', ...
                        'Numerator', fir1(10, .25));
        end

        x = randn(1000,1);                          % Input signal
        d = step(hfilt, x) + 0.01*randn(1000,1);    % Desired signal
        [~,~,w] = step(hlms, x, d);                 % Filter weights

        % Declare functions called into MATLAB that do not generate
        % code as extrinsic.

        coder.extrinsic('stem');

        stem([get(hfilt, 'Numerator').', w]);
    end

    % To compile this function use codegen lmssystemidentification.
    % This produces a mex file with the same name in the current
    % directory.
```

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

Usage Rules for System Objects in Generated MATLAB Code

- Assign System objects to `persistent` variables.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a compiler options object. For example,

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

  Then, include `'-config f'` in your `codegen` command.
- Initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Call the constructor exactly once for any instance of a System object.
- Set arguments to System object constructors as compile-time constants.
- Use the object constructor to set System object properties because you cannot use dot notation for code generation. You can use the `get` method to display properties.
- Test your code in simulation before generating code.

Limitations on Using System Objects in Generated MATLAB Code

- Ensure that size, type and complexity of inputs do not change.
- Ensure that the value assigned to a nontunable or public property is a constant and that there is at most one assignment to that property (including the assignment in the constructor). Do not set any properties during code generation.
- The only System object methods supported in code generation are
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `reset`
  - `step`

- Do not set System objects to become outputs from the 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 (i.e., functions called in interpreted mode) using the `coder.extrinsic` function. Do not return System objects from any extrinsic functions.

Use System Objects with 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 the *MATLAB Coder User's Guide* for more information.

Use System Objects with the MATLAB Function Block

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

Index