Communications Blockset

For Use with Simulink®

- Modeling
- Simulation
- Implementation

User's Guide

Version 3

How to Contact The 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 Blockset User's Guide

Revision History

June 2001	Online only	New for Version 2.0.1 (Release 12.1)
July 2002	First printing	Revised for Version 2.5 (Release 13)
June 2004	Online only	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.1 (Release 14SP2)
September 2005	Online only	Revised for Version 3.2 (Release 14SP3)
March 2006	Online only	Revised for Version 3.3 (Release 2006a)
September 2006	Online only	Revised for Version 3.4 (Release 2006b)

Contents

1 Using the Libraries

2 Modeling Communication Systems

3 Data Type Support

4 Simulink and Real-Time Workshop Related Enhancements

Index

1

Using the Libraries

This chapter describes and illustrates how to model communication techniques using the blocks in the Communications Blockset. Most sections correspond to core libraries within the blockset.

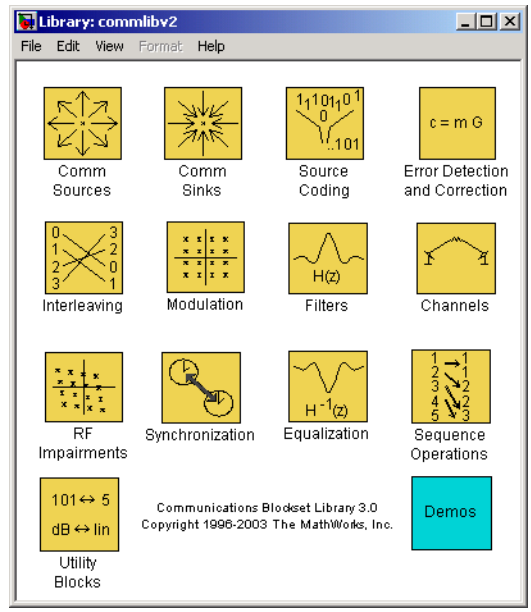
Channels (p. 1-122)	Modeling channel impairments
RF Impairments (p. 1-127)	Modeling impairments caused by the radio frequency components
Synchronization (p. 1-138)	Phase recovery methods and phase-locked loops
Equalizers (p. 1-160)	Adaptive and MLSE equalizers

Accessing the Libraries

You can access the main library of the Communications Blockset by entering

```
commlib
```

in the MATLAB® Command Window.



From the main library, you can access sublibraries by double-clicking their icons.

On Windows platforms, you can also use the Simulink® Library Browser to access libraries of the Communications Blockset. To open the Simulink Library Browser, enter `simulink` in the MATLAB Command Window.

Source code for the communications blocks can be found in `<MATLAB>\toolbox\commblks\sim\sfun`.

Signal Support

Simulink® supports matrix signals and one-dimensional arrays, and frame-based and sample-based signals. This section describes how the Communications Blockset processes certain kinds of matrix and frame-based signals. The topics are

Signal Terminology

This section defines important terms related to matrix and frame-based signals.

Matrices, Vectors, and Scalars

This document uses the unqualified words *scalar* and *vector* in ways that emphasize a signal's number of elements, not its strict dimension properties:

- A *scalar* signal is one that contains a single element. The signal could be a one-dimensional array with one element, or a matrix of size 1-by-1.
- A *vector* signal is one that contains one or more elements, arranged in a series. The signal could be a one-dimensional array, a matrix that has exactly one column, or a matrix that has exactly one row. The number of elements in a vector is called its *length* or, sometimes, its *width*.

In cases when it is important for a description or schematic to distinguish among different types of scalar signals or different types of vector signals, this document mentions the distinctions explicitly. For example, the terms *one-dimensional array*, *column vector*, and *row vector* distinguish among three types of vector signals.

The *size* of a matrix is the pair of numbers that indicate how many rows and columns the matrix has. The *orientation* of a two-dimensional vector is its status as either a row vector or column vector. A one-dimensional array has no orientation.

A matrix signal that has more than one row and more than one column is called a *full matrix* signal.

Frame-Based and Sample-Based Signals

In Simulink, each matrix signal has a *frame attribute* that declares the signal to be either frame-based or sample-based, but not both. (A one-dimensional array signal is always sample-based, by definition.) Simulink indicates the frame attribute visually by using a double connector line in the model window instead of a single connector line. In general, Simulink interprets frame-based and sample-based signals as follows:

- A frame-based signal in the shape of an M-by-1 (column) matrix represents M successive samples from a single time series.
- A frame-based signal in the shape of a 1-by-N (row) matrix represents a sample of N independent channels, taken at a single instant in time.
- A sample-based matrix signal might represent a set of bits that collectively represent an integer, or a set of symbols that collectively represent a codeword, or something else *other than* a fragment of a single time series.

Processing Matrices, Vectors, and Scalars

These rules indicate the shapes of sample-based signals that Communications Blockset blocks can process:

- Most blocks do not process matrix signals that have more than one row and more than one column.
- In their numerical computations, blocks that process scalars do not distinguish between one-dimensional scalars and one-by-one matrices. If the block produces a scalar output from a scalar input, the block preserves dimension.
- If a block can process sample-based vectors,
  - The numerical computations do not distinguish between one-dimensional arrays, M-by-1 matrices, and 1-by-N matrices.
  - The block output preserves dimension and orientation.
  - The block treats elements of the input vector as a collection that arises naturally from the block's operation (for example, a collection of symbols

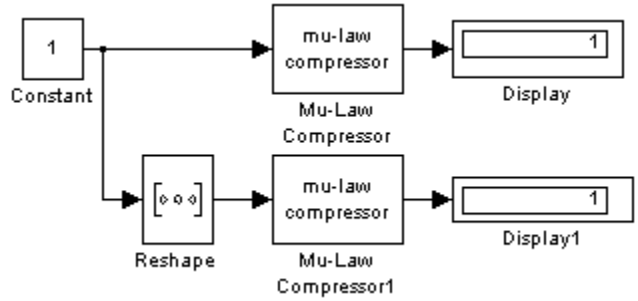
that jointly represent a codeword), or as samples from independent channels. The block does *not* assume that the elements of the input vector are successive samples from a single time series.

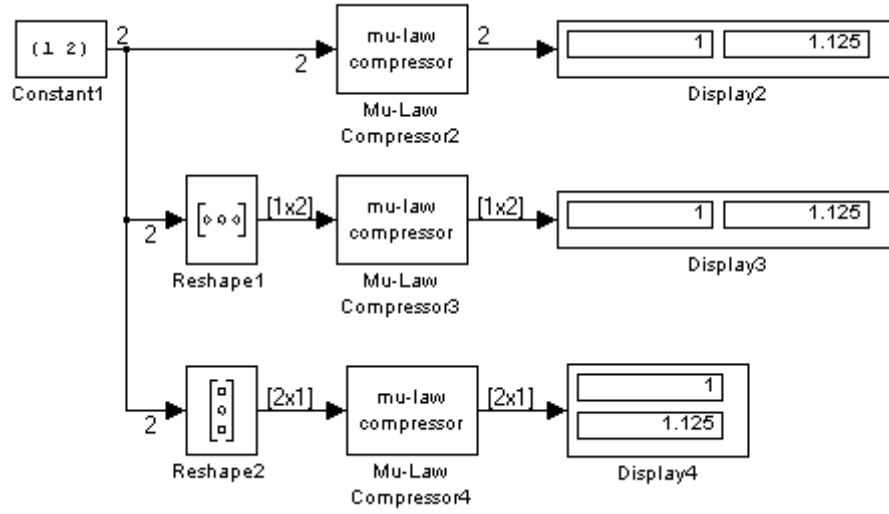
Some blocks process vectors but require them to be frame-based. For more information about processing frame-based signals, see "Processing Frame-Based and Sample-Based Signals" on page 1-7.

To find out whether a block processes scalar signals, vector signals, or both, refer to its entry in the reference section.

Illustrations of Scalar and Vector Processing

The figures below depict the preservation of dimension and orientation when a block processes scalars (without oversampling) and vectors. To display signal dimensions in your model, turn on the **Signal dimensions** option in the **Port/signal displays** submenu of the model window's **Format** menu.





Processing Frame-Based and Sample-Based Signals

All one-dimensional arrays are sample-based, but a matrix signal can be either frame-based or sample-based. A frame-based signal in the shape of an N-by-1 matrix represents a series of N successive samples from a single time series. The Communications Blockset processes some frame-based signals and is compatible with the Signal Processing Blockset. However, the Communications Blockset omits some frame-based features, and many blocks are not specifically optimized for frame-based processing.

These rules indicate how most Communications Blockset blocks handle frame-based matrix signals:

- Most blocks do not process frame-based matrix signals that have more than one row and more than one column.
- Most blocks do not process frame-based row vectors and do not support multichannel functionality.
- Blocks that process continuous-time signals do not process frame-based inputs. Such blocks include the analog modulation blocks and the analog phase-locked loop blocks.

- Blocks for which a frame-based multichannel operation makes sense, even if the blocks do not currently support such operation, reject sample-based vectors because their interpretation is ambiguous.

  Frame-based vectors, however, have an unambiguous interpretation. Blocks interpret a frame-based row vector as multiple channels at a single instant of time, and interpret a frame-based column vector as multiple samples from a single time series (that is, a single channel).

- Some blocks, such as the digital baseband modulation blocks, can produce multiple output values for each value of a scalar input signal. In such cases, a frame-based 1-by-1 matrix input results in a frame-based column vector output. By contrast, a sample-based scalar input results in a sample-based scalar output with a smaller sample time.

Communications Sources

Every communication system contains one or more sources. You can find sources in the Simulink Sources library, in the Signal Processing Sources library, and in the Communications Blockset's Comm Sources library.

You can open the Comm Sources library by double-clicking its icon in the main Communications Blockset library.

Blocks in the Comm Sources library can

- Generate random data
- Generate random noise to simulate channels
- Generate sequences that can be used for spreading or synchronization in a communication system

This section describes these capabilities, considering first random and then nonrandom signals.

Random Data Sources

Blocks in the Random Data Sources sublibrary of the Comm Sources library generate random data to simulate signal sources. You can use blocks in the Random Data Sources sublibrary to generate

- Random bits
- Random integers

In addition, you can use built-in Simulink blocks such as the Random Number block as a data source.

You can open the Random Data Sources sublibrary by double-clicking its icon (found in the Comm Sources library of the main Communications Blockset library).

Random Bits

The Bernoulli Binary Generator block generates random bits and is suitable for representing sources. The block considers each element of the signal to be an independent Bernoulli random variable. Also, different elements need not be identically distributed.

Random Integers

The Random Integer Generator and Poisson Integer Generator blocks both generate vectors containing random nonnegative integers. The Random Integer Generator block uses a uniform distribution on a bounded range that you specify in the block mask. The Poisson Integer Generator block uses a Poisson distribution to determine its output. In particular, the output can include any nonnegative integer.

Random Noise Generators

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

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

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

Sequence Generators

You can use blocks in the Sequence Generators sublibrary of the Communications Sources library to generate sequences for spreading or synchronization in a communication system. You can open the Sequence

Generators sublibrary by double-clicking its icon in the main Communications Blockset library.

Blocks in the Sequence Generators sublibrary generate

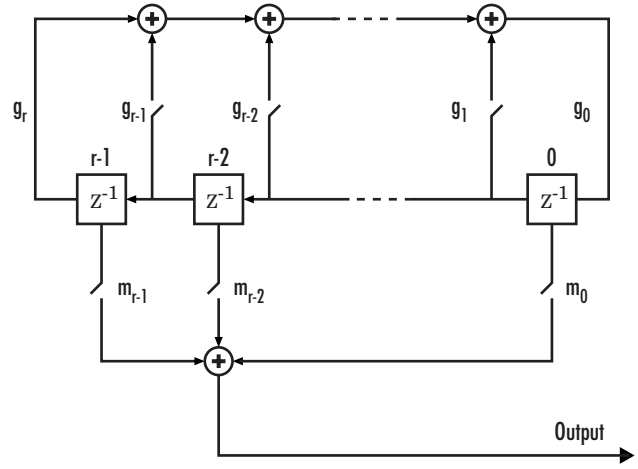
- Pseudorandom sequences
- Synchronization codes
- Orthogonal codes

Pseudorandom Sequences

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

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

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



All *r* registers in the generator update their values at each time step according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The shift register can be described by a binary polynomial in z, $g_r z^r + g_{r-1} z^{r-1} + ... + g_0$. The coefficient g_i is 1 if there is a connection from the ith shift register to the adder, and 0 otherwise.

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

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

See “Pseudorandom Sequences” on page 1-13 for an example that uses these blocks.

Synchronization Codes

The Barker Code Generator block generates Barker codes to perform synchronization. Barker codes are subsets of PN sequences. They are short codes, with a length at most 13, which are low-correlation sidelobes. A

correlation sidelobe is the correlation of a codeword with a time-shifted version of itself.

Orthogonal Codes

Orthogonal codes are used in systems in which the receiver is perfectly synchronized with the transmitter. For such systems, the despreading operation is ideal when orthogonal codes are used for the spreading. For example, they are used in the forward link of the IS-95 system, in which the base station transmits a pilot signal to help the receiver gain synchronization.

Code	Block
Hadamard codes	Hadamard Code Generator
OVSF codes	OVSF Code Generator
Walsh codes	Walsh Code Generator

See “Orthogonal Sequences” on page 1-17 for an example that uses these blocks.

Sequence Generator Examples

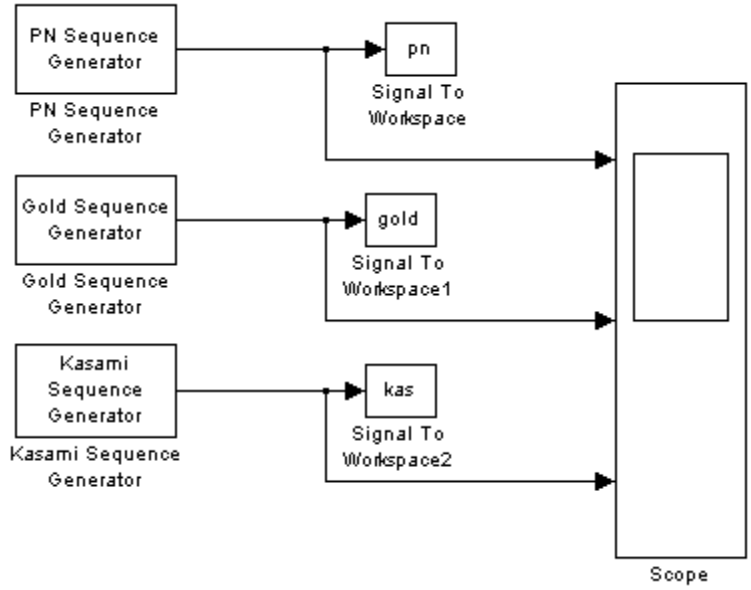
This section presents two example models that illustrate the blocks in the Sequence Generators library.

Pseudorandom Sequences

This example describes the autocorrelation properties of the pseudorandom sequences generated by the following three blocks:

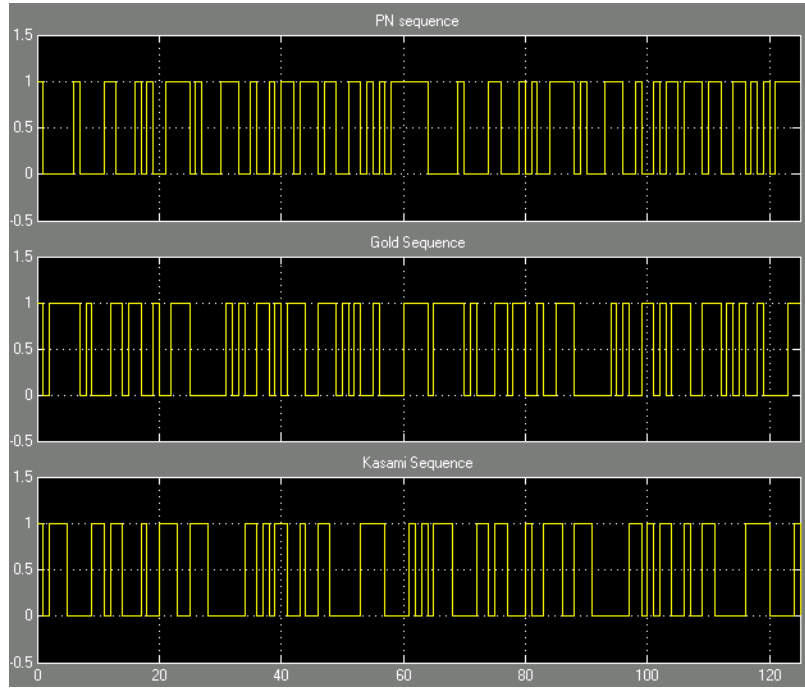
- PN Sequence Generator
- Gold Sequence Generator
- Kasami Sequence Generator

If you are reading this in the MATLAB® Help Browser, click here to open the model.



The model displays the output sequences of the three blocks in a scope. All three blocks have the same **Generator polynomial** parameter, `[1 0 0 0 0 1 1]`, whose digits are the coefficients of the polynomial $x^6 + x + 1$. Since this polynomial has degree 6, the output sequence has period $2^6 - 1 = 63$.

When you run the model, the scope displays two periods of data for each of the three signals, as in the following figure.



The model also sends the output sequences to the MATLAB workspace as the vectors `pn`, `gold`, and `kas`. You can verify the autocorrelation properties of the output of the PN Sequence Generator block by entering the following at the MATLAB prompt:

```
x = pn(1:63); % Take one period only.
x = 1 - 2.*x; % Convert to bipolar.
for i = 1:63  % Determine the cyclic autocorrelation.
    corrvec(i) = x' * [x(i:end); x(1:i-1)];
end
corrvals = unique(sort(corrvec)) % Choose the unique values.
```

This code calculates the cyclic autocorrelation of the PN sequence, by taking the inner product of one period of the sequence with each of its 63 cyclic rotations, and stores the results in a vector, `corrvec`, of length 63. The code then sorts the entries of `corrvec` and finds the unique autocorrelation values.

The result is

```
corrvals =
-1    63
```

The first entry of the vector `corrvec` is `63`, while all other values are `-1`, as you can verify by entering `corrvec` at the MATLAB prompt. This means that `63` occurs only by taking the inner product of the sequence `pn` with an unrotated copy of itself. All other inner products have the value `-1`.

You can analyze the output sequences of the Gold Sequence Generator block and the Kasami Sequence Generator block similarly by changing the first line of the preceding code to

```
x = gold(1:63);
```

and

```
x = kas(1:63);
```

respectively.

For the Gold and Kasami sequences, the autocorrelation takes on three values. For example, the values for the Gold sequence are

```
corrvals =
-17    -1    15    63
```

The values for the Kasami sequence are

```
corrvals =
-9    -1     7    63
```

Of the three types of sequences, the PN sequences are best suited for synchronization because the autocorrelation takes on just two values. However, the Gold and Kasami sequences provide a larger number of sequences with good cross-correlation properties than do the PN sequences.

The peak value of `corrvals` for the Kasami sequence is less than the peak value for the Gold sequence. In fact, the small set of Kasami sequences

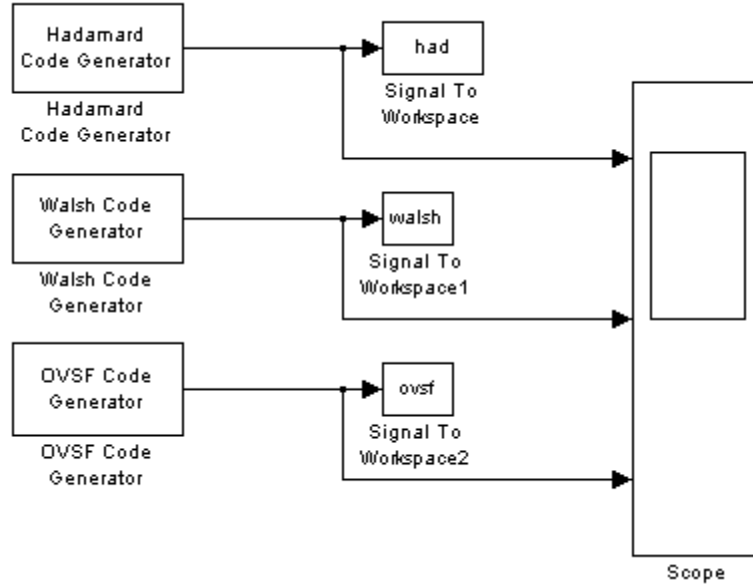
satisfies the lower bounds for correlation values, and for this reason they are also referred to as *optimal sequences*.

Orthogonal Sequences

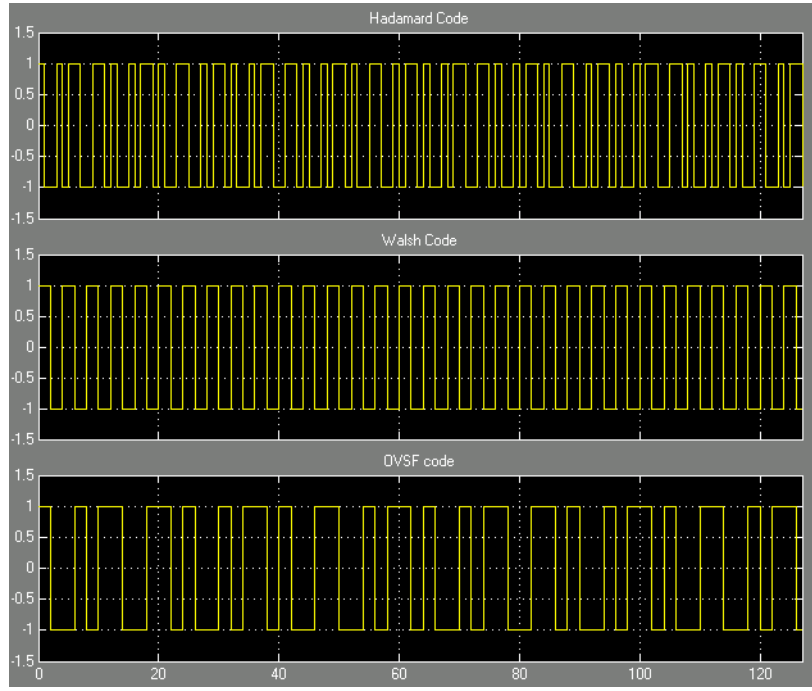
This example demonstrates the orthogonality of pairs of sequences generated using different **Code index** parameters, for each of the following three blocks:

- Hadamard Code Generator
- Walsh Code Generator
- OVSF Code Generator

If you are reading this in the MATLAB Help Browser, click here to open the model.



The model displays the output sequences of the three blocks in a scope. All three blocks output sequences of period 64, corresponding to their **Code length** parameters. When you run the model, the scope displays two periods of data for each sequence.



The following script runs the model twice, the first time with the **Code index** parameter of `60` for all three blocks, and the second time with a **Code index** of `30`. The script then calculates, for each of the three blocks, the cross-correlation between the sequence generated by the first run and the sequence generated by the second run.

```
% Simulate once.
set_param('doc_ortho/Hadamard Code Generator', 'index', '60');
set_param('doc_ortho/Walsh Code Generator', 'index', '60');
set_param('doc_ortho/OVSF Code Generator', 'index', '60');
sim('doc_ortho');

% Store the codes.
had60 = had(1:64);
walsh60 = walsh(1:64);
ovsf60 = ovsf(1:64);

% Simulate again.
```

```
set_param('doc_ortho/Hadamard Code Generator', 'index', '31');
set_param('doc_ortho/Walsh Code Generator', 'index', '31');
set_param('doc_ortho/OVSF Code Generator', 'index', '31');
sim('doc_ortho');

% Store the codes.
had31 = had(1:64);
walsh31 = walsh(1:64);
ovsf31 = ovsf(1:64);

% Calculate the cross-correlation.
hadcorr = had60(1:64)'*had31(1:64);
hadcorr
walshcorr = walsh60(1:64)'*walsh31(1:64);
walshcorr
ovsfcorr = ovsf60(1:64)'*ovsf31(1:64);
ovsfcorr
```

The results are

```
haddcorr=
0
walshcorr =
0
ovsfcorr =
0
```

The results show that for each block, the sequence generated by the first run is orthogonal to the sequence generated by the second run.

Block Parameters

This section discusses the sample time parameter, seed parameter, and signal attribute parameters that are common to many random source blocks, and then discusses each category of random source.

Sample Time Parameter for Random Sources

Each of the random source blocks requires you to set a **Sample time** parameter in the block mask. If you configure the block to produce a

sample-based signal, this parameter is the time interval between successive updates of the signal. If you configure the block to produce a frame-based matrix signal, the **Sample time** parameter is the time interval between successive rows of the frame-based matrix.

If you use a Simulink Probe block to query the period of a frame-based output from a random source block in the Comm Sources library, note that the Probe block reports the period of the *entire frame*, not the period of *each sample* in a given channel of the frame. The following equation relates the quantities involved for a single-channel signal:

A seconds/frame = (B seconds/sample)*(S samples/frame)

where

- A is the number shown in the Probe block after the `Tf` notation.
- B is the random source block's **Sample time** parameter.
- S is the random source block's **Samples per frame** parameter.

Seed Parameter

The blocks in the Communication Sources library that generate random data require you to set a seed in the block mask. This is the initial seed that the random number generator uses when forming its sequence of numbers. Make sure that initial seeds in different blocks in a model have different values, so that they generate statistically independent sequences.

Some blocks in the Communication Sources library require you to choose their seeds according to the following rule, in order to obtain accurate results:

Seed rule: Set the **Initial seed** to be a prime number greater than 30.

This rule applies to the following blocks:

- Gaussian Noise Generator
- Rayleigh Noise Generator

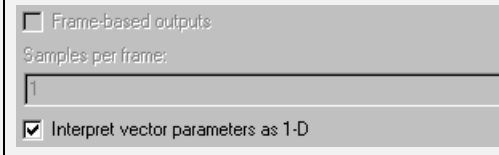
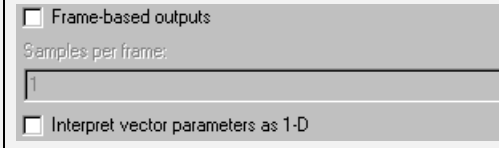
- Rician Noise Generator

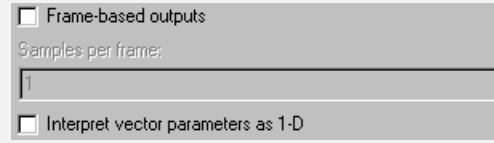
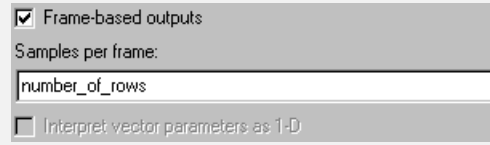
To avoid having to remember whether a block you are using is on this list, you simply apply the seed rule to all source blocks that have an **Initial seed** parameter.

You can choose integers that satisfy the seed rule with the `randseed` function. Entering `randseed` at the MATLAB prompt returns a prime number greater than 30. If you choose a constant seed such as `randseed(n)`, where n is some positive integer variable, the block produces the same noise sequence each time you start the simulation. The sequence is different from that produced with a different constant seed. If you want the noise to be different each time you start the simulation, use a varying seed such as `randseed(cputime)`.

Signal Attribute Parameters for Random Sources

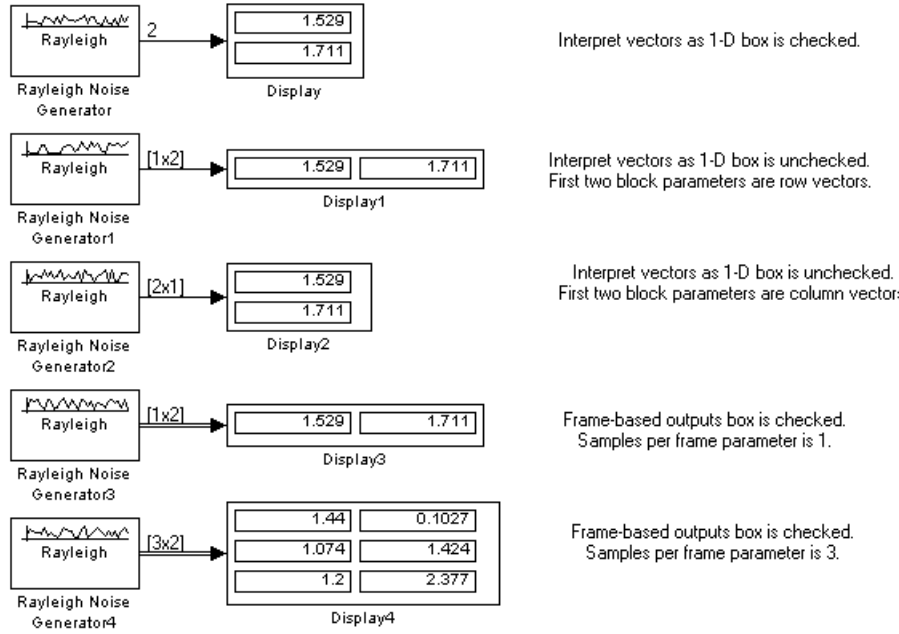
In most random source blocks, the output can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. The following table indicates how to set certain block parameters depending on the kind of signal you want to generate.

Signal Attributes	Parameter Settings
Sample-based, one-dimensional	
Sample-based row vector	 Also, any vector parameters in the block should be rows, not columns.

Signal Attributes	Parameter Settings
Sample-based column vector	 Also, any vector parameters in the block should be columns, not rows.
Frame-based	 Also, set **Samples per frame** to the number of samples in each output frame, that is, the number of rows in the signal.

The **Frame-based outputs** and **Interpret vector parameters as 1-D** check boxes are mutually exclusive, because frame-based signals and one-dimensional signals are mutually exclusive. The **Samples per frame** parameter field is active only if the **Frame-based outputs** check box is checked.

Example. The model in the following figure illustrates that one random source block can produce various kinds of signals. The annotations in the model indicate how each copy of the block is configured. Each block's configuration affects the type of connector line (single or double) and the signal dimensions that appear above each connector line. In the case of the Rayleigh Noise Generator block, the first two block parameters (**Sigma** and **Initial seed**) determine the number of channels in the output; for analogous indicators in other random source blocks, see their individual reference entries.



The particular mask parameters depend on the block. See each block's reference page for details.

Communications Sinks

The Communications Blockset provides sinks and display devices that facilitate analysis of communication system performance. You can open the Comm Sinks library by double-clicking its icon in the main Communications Blockset library.

Sink Features of the Blockset

Blocks in this library can

- Compute error statistics
- Plot an eye diagram
- Generate a scatter plot
- Plot a signal trajectory

This section describes these capabilities. Other sinks are in the Simulink Sinks library and in the Signal Processing Sinks library.

Error Statistics

The Error Rate Calculation block compares input data from a transmitter with input data from a receiver. It calculates these error statistics:

- Error rate
- Number of error events
- Total number of input events

The block reports these statistics either as final values in the workspace or as running statistics at an output port.

You can use this block either with binary inputs to compute the bit error rate, or with symbol inputs to compute the symbol error rate. You can use frame-based or sample-based data. If you use frame-based data, you can have the block consider certain samples and ignore others.

The example in the section "Example: Soft-Decision Decoding" on page 1-67 illustrates the use of the Error Rate Calculation block.

Scopes

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

- "Eye Diagrams" on page 1-25
- "Scatter Plots" on page 1-26
- "Signal Trajectories" on page 1-26

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

Block Name	Plots
Discrete-Time Eye Diagram Scope	Eye diagram of a discrete signal
Discrete-Time Scatter Plot Scope	Scatter plot of a discrete signal
Discrete-Time Signal Trajectory Scope	Signal trajectory of a discrete signal

Eye Diagrams

An eye diagram is a simple and convenient tool for studying the effects of intersymbol interference and other channel impairments in digital transmission. When this blockset constructs an eye diagram, it plots the received signal against time on a fixed-interval axis. At the end of the fixed interval, it wraps around to the beginning of the time axis. As a result, the diagram consists of many overlapping curves. One way to use an eye diagram is to look for the place where the "eye" is most widely opened, and use that point as the decision point when demapping a demodulated signal to recover a digital message.

The Discrete-Time Eye Diagram Scope block produces eye diagrams. This block processes discrete-time signals. and periodically draws a line to indicate a decision, according to a mask parameter.

Examples appear in "Example: Viewing a Sinusoid" on page 1-26 and "Example: Viewing a Modulated Signal" on page 1-29.

Scatter Plots

A scatter plot of a signal plots the signal's value at its decision points. In the best case, the decision points should be at times when the eye of the signal's eye diagram is the most widely open.

The Discrete-Time Scatter Plot Scope block produces scatter plots from discrete-time signals. An example appears in "Example: Viewing a Sinusoid" on page 1-26.

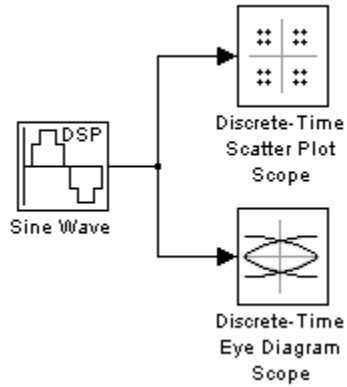
Signal Trajectories

A signal trajectory is a continuous plot of a signal over time. A signal trajectory differs from a scatter plot in that the latter displays points on the signal trajectory at discrete intervals of time.

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

Example: Viewing a Sinusoid

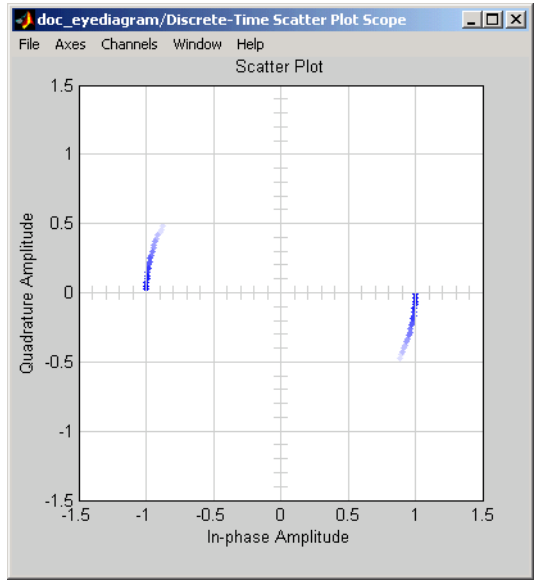
The following model produces a scatter plot and an eye diagram from a complex sinusoidal signal. Because the decision time interval is almost, but not exactly, an integer multiple of the period of the sinusoid, the eye diagram exhibits drift over time. More specifically, successive traces in the eye diagram and successive points in the scatter diagram are near each other but do not overlap.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Sine Wave, in the Signal Processing Sources library (*not* the Sine Wave block in the Simulink Sources library)
  - Set **Frequency** to `.502`.
  - Set **Output complexity** to `Complex`.
  - Set **Sample time** to `1/16`.
- Discrete-Time Scatter Plot Scope, in the Comm Sinks library
  - On the **Plotting Properties** panel, set **Samples per symbol** to `16`.
  - On the **Figure Properties** panel, set **Scope position** to `figposition([2.5 55 35 35]);`.
- Discrete-Time Eye Diagram Scope, in the Comm Sinks library
  - On the **Plotting Properties** panel, set **Samples per symbol** to `16`.
  - On the **Figure Properties** panel, set **Scope position** to `figposition([42.5 55 35 35]);`.

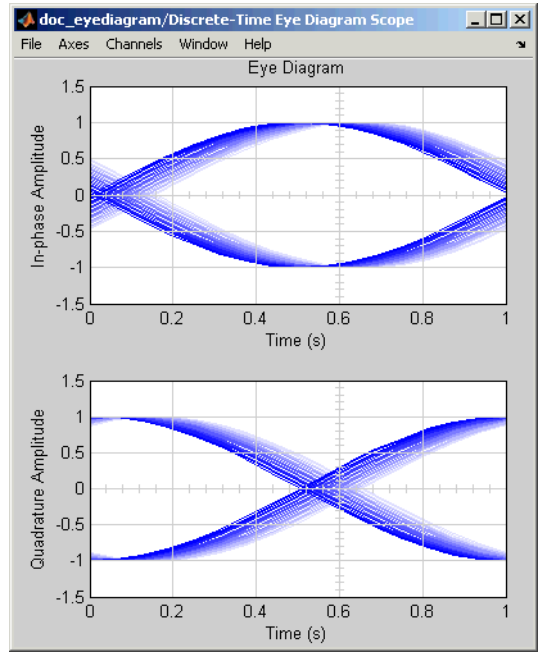
Connect the blocks as shown in the preceding figure. From the model window's **Simulation** menu, choose **Configuration parameters**; in the Configuration Parameters dialog box, set **Stop time** to `250`. Running the model produces the following scatter diagram plot.



The points of the scatter plot lie on a circle of radius 1. Note that the points fade as time passes. This is because the box next to **Color fading** is checked under **Rendering Properties**, which causes the scope to render points more dimly the more time that passes after they are plotted. If you clear this box, you see a full circle of points.

If you add the Discrete-Time Signal Trajectory Scope block to the model, it displays a circular trajectory.

In the eye diagram, the upper set of traces represents the real part of the signal and the lower set of traces represents the imaginary part of the signal.



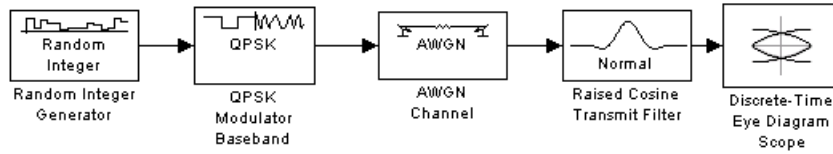
Example: Viewing a Modulated Signal

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

- "Eye Diagram of a Modulated Signal" on page 1-29
- "Scatter Plot of a Modulated Signal" on page 1-33
- "Signal Trajectory of a Modulated Signal" on page 1-34

Eye Diagram of a Modulated Signal

The following model modulates a random signal using QPSK, filters the signal with a raised cosine filter, and creates an eye diagram from the filtered signal.

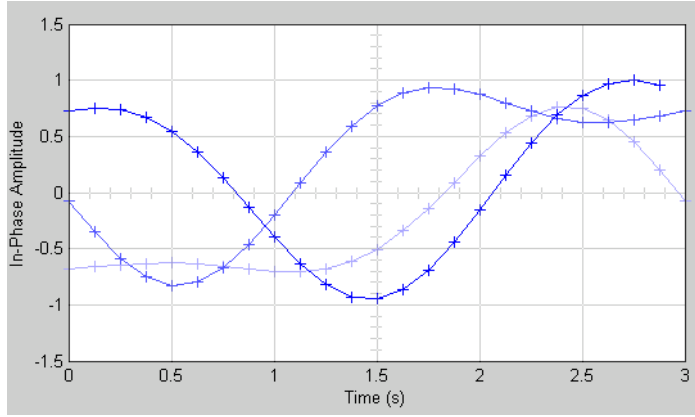


To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure the following blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `4`.
  - Set **Sample time** to to `0.01`.
- QPSK Modulator Baseband, in PM in the Digital Baseband sublibrary of the Modulation library of the Communications Blockset, with default parameters
- AWGN Channel, in the Channels library of the Communications Blockset, with the following changes to the default parameter settings:
  - Set **Mode** to `Signal-to-noise ratio (SNR)`.
  - Set **SNR (dB)** to `15`.
- Raised Cosine Transmit Filter, in the Comm Filters library
  - Set **Filter type** to `Normal`.
  - Set **Group delay** to `3`.
  - Set **Rolloff factor** to `0.5`.
  - Set **Input sampling mode** to `Sample-based`.
  - Set **Upsampling factor** to `8`.
- Discrete-Time Eye Diagram Scope, in the Comms Sinks library
  - Set **Samples per symbol** to `8`.
  - Set **Symbols per trace** to `3`. This specifies the number of symbols that are displayed in each trace of the eye diagram. A *trace* is any one of the individual lines in the eye diagram.

- Set **Traces displayed** to 3.
- Set **New traces per display** to 1. This specifies the number of new traces that appear each time the diagram is refreshed. The number of traces that remain in the diagram from one refresh to the next is **Traces displayed** minus **New traces per display**.
- On the **Rendering Properties** panel, set **Markers** to + to indicate the points plotted at each sample. The default value of **Markers** is empty, which indicates no marker.
- On the **Figure Properties** panel, set **Eye diagram to display** to `In-phase only`.

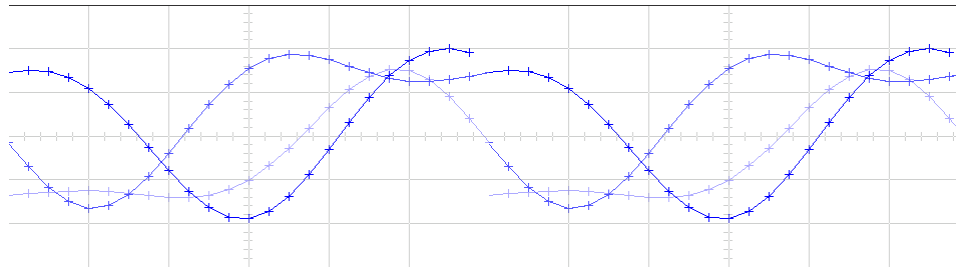
When you run the model, the Discrete-Time Eye Diagram Scope displays the following diagram. Your exact image varies depending on when you pause or stop the simulation.



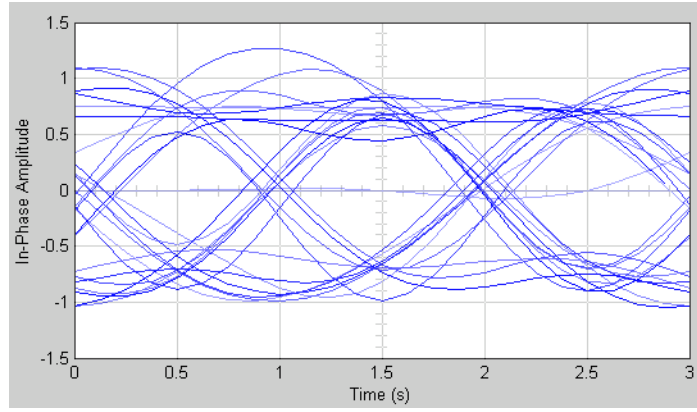
Three traces are displayed. Traces 2 and 3 are faded because the **Color fading** check box under **Rendering Properties** is selected. This causes traces to be displayed less brightly the older they are. In this picture, Trace 1 is the most recent and Trace 3 is the oldest. Because **New traces per display** is set to 1, only Trace 1 is appearing for the first time. Traces 2 and 3 also appear in the previous display.

Because **Symbols per trace** is set to 3, each trace contains three symbols, and because **Samples per trace** is set to 8, each symbol contains eight samples.

Note that trace 1 contains 24 points, which is the product of **Symbols per trace** and **Samples per symbol**. However, traces 2 and 3 contain 25 points each. The last point in trace 2, at the right border of the scope, represents the same sample as the first point in trace 1, at the left border of the scope. Similarly, the last point in trace 3 represents the same sample as the first point in trace 2. These duplicate points indicate where the traces would meet if they were displayed side by side, as illustrated in the following picture.



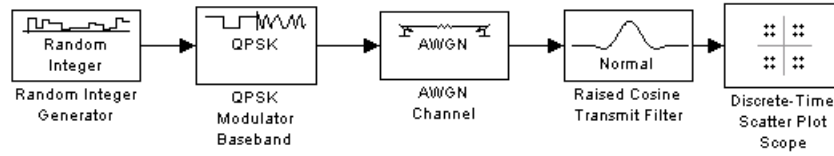
You can view a more realistic eye diagram by changing the value of **Traces displayed** to `40` and clearing the **Markers** field.



When the **Offset** parameter is set to `0`, the plotting starts at the center of the first symbol, so that the open part of the eye diagram is in the middle of the plot for most points.

Scatter Plot of a Modulated Signal

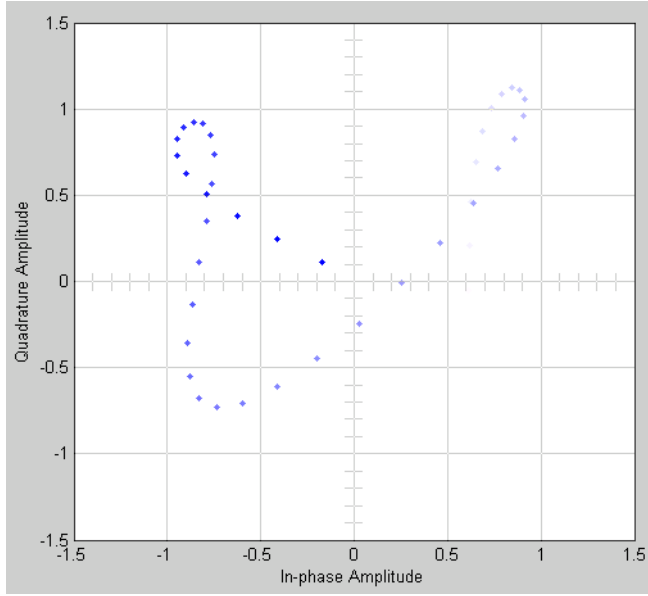
The following model creates a scatter plot of the same signal considered in “Eye Diagram of a Modulated Signal” on page 1-29.



To build the model, follow the instructions in “Eye Diagram of a Modulated Signal” on page 1-29 but replace the Discrete-Time Eye Diagram block with the following block:

- Discrete-Time Scatter Plot Scope, in the Comms Sinks library
  - Set **Samples per symbol** to `2`.
  - Set **Offset** to `0`. This specifies the number of samples to skip before plotting the first point.
  - Set **Points displayed** to `40`.
  - Set **New points per display** to `10`. This specifies the number of new points that appear each time the diagram is refreshed. The number of points that remain in the diagram from one refresh to the next is **Points displayed** minus **New points per display**.

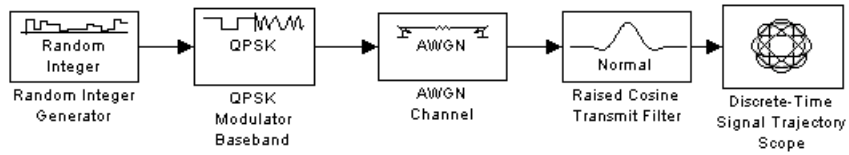
When you run the simulation, the Discrete-Time Scatter Plot Scope block displays the following plot.



The plot displays 30 points. Because the **Color fading** check box under **Rendering Properties** is selected, points are displayed less brightly the older they are.

Signal Trajectory of a Modulated Signal

The following model creates a signal trajectory plot of the same signal considered in “Eye Diagram of a Modulated Signal” on page 1-29.

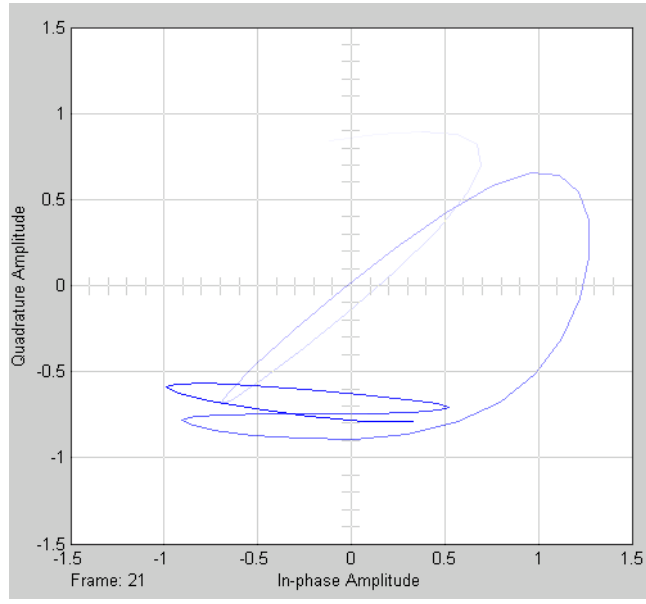


To build the model, follow the instructions in “Eye Diagram of a Modulated Signal” on page 1-29 but replace the Discrete-Time Eye Diagram block with the following block:

- Discrete-Time Signal Trajectory Scope, in the Comms Sinks library

- Set **Samples per symbol** to `8`.
- Set **Symbols displayed** to `40`. This specifies the number of symbols displayed in the signal trajectory. The total number of points displayed is the product of **Samples per symbol** and **Symbols displayed**.
- Set **New symbols per display** to `10`. This specifies the number of new symbols that appear each time the diagram is refreshed. The number of symbols that remain in the diagram from one refresh to the next is **Symbols displayed** minus **New symbols per display**.

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

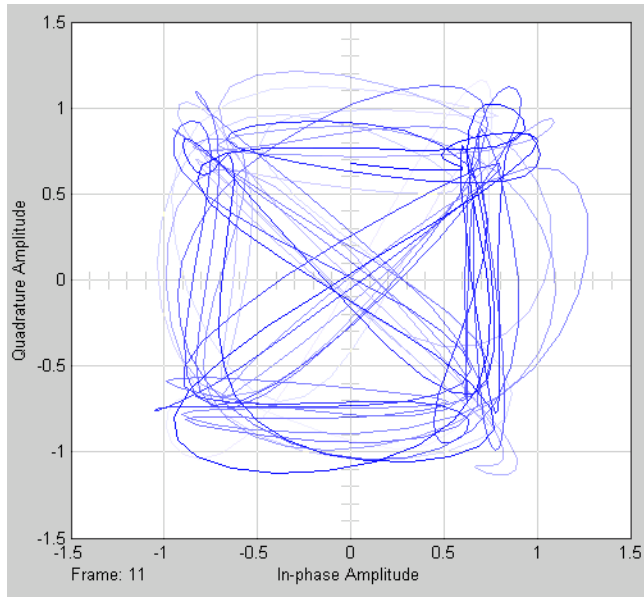


The plot displays 40 symbols. Because the **Color fading** check box under **Rendering Properties** is selected, symbols are displayed less brightly the older they are.

See “Scatter Plot of a Modulated Signal” on page 1-33 to compare the preceding signal trajectory to the scatter plot of the same signal. The

Discrete-Time Signal Trajectory Scope block connects the points displayed by the Discrete-Time Scatter Plot Scope block to display the signal trajectory.

If you increase **Symbols displayed** to 100, the model produces a signal trajectory like the one below. The total number of points displayed at any instant is 800, which is the product of the parameters **Samples per symbol** and **Symbols displayed**.



Source Coding

Source coding, also known as *quantization* or *signal formatting*, is a way of processing data to reduce redundancy or prepare it for later processing. Analog-to-digital conversion and data compression are two categories of source coding.

Source coding divides into two basic procedures: *source encoding* and *source decoding*. Source encoding converts a source signal into a digital signal using a quantization method. The symbols in the resulting signal are nonnegative integers in some finite range. Source decoding recovers the original information from the source-coded signal.

For background material on the subject of source coding, see the works listed in "Selected Bibliography for Source Coding" on page 1-45.

Source-Coding Features of the Blockset

This blockset supports scalar quantization, companders, and differential coding. It does not support vector quantization. You can open the Source Coding library by double-clicking its icon in the main Communications Blockset library.

Blocks in the Source Coding library can

- Use a partition and codebook to quantize a signal
- Compand a signal using a μ-law or A-law compressor or expander
- Encode or decode a signal using differential coding

Supporting functionality in the Communications Toolbox also allows you to optimize source-coding parameters for a set of training data. See "Optimizing Quantization Parameters" in the Communications Toolbox User's Guide for more information about such capabilities.

Representing Quantization Parameters

Scalar quantization is a process that maps all inputs within a specified range to a common value. It maps inputs in a different range of values to a different common value. In effect, scalar quantization digitizes an analog signal. Two

parameters determine a quantization: a partition and a codebook. This section describes how blocks represent these parameters.

Partitions

A quantization partition defines several contiguous, nonoverlapping ranges of values within the set of real numbers. To specify a partition as a parameter, list the distinct endpoints of the different ranges in a vector.

For example, if the partition separates the real number line into the sets

$\{x:\ x \le 0\}$
$\{x:\ 0 < x \le 1\}$
$\{x:\ 1 < x \le 3\}$
$\{x:\ 3 < x\}$

then you can represent the partition as the three-element vector

```
[0,1,3]
```

The length of the partition vector is one less than the number of partition intervals.

Codebooks

A codebook tells the quantizer which common value to assign to inputs that fall into each range of the partition. Represent a codebook as a vector whose length is the same as the number of partition intervals. For example, the vector

```
[-1,0.5,2,3]
```

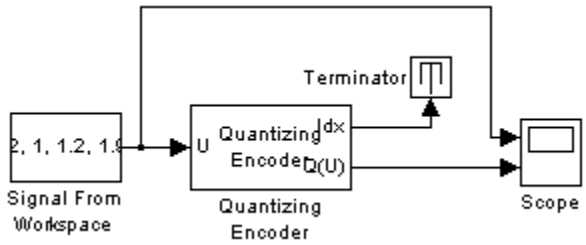
is one possible codebook for the partition `[0,1,3]`.

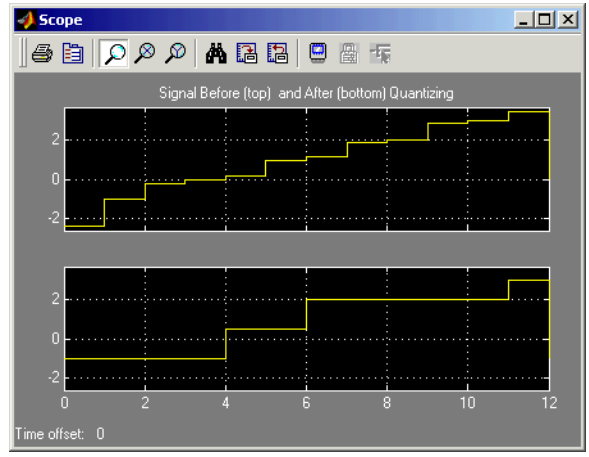
Quantizing a Signal

This section shows how the Quantizing Encoder and Quantizing Decoder blocks use the partition and codebook parameters. The examples here are analogous to “Scalar Quantization Example 1” and “Scalar Quantization Example 2” in the Communications Toolbox documentation.

Scalar Quantization Example 1

The figure below shows how the Quantizing Encoder block uses the partition and codebook as defined above to map a real vector to a new vector whose entries are either -1, 0.5, 2, or 3. In the Scope window, the bottom signal is the quantization of the (original) top signal.





To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

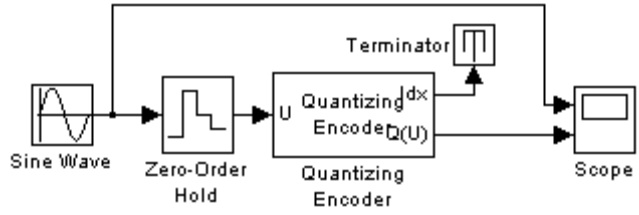
- Signal From Workspace, in the Signal Processing Sources library
  - Set **Signal** to `[-2.4,-1,-.2,0,.2,1,1.2,1.9,2,2.9,3,3.5]'`.
- Quantizing Encoder
  - Set **Quantization partition** to `[0, 1, 3]`.
  - Set **Quantization codebook** to `[-1, 0.5, 2, 3]`.

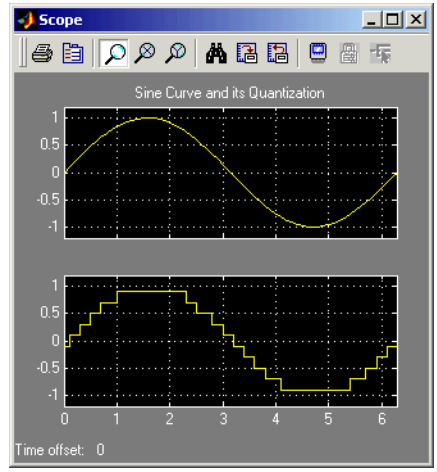
- Terminator, in the Simulink Sinks library
- Scope, in the Simulink Sinks library
  - After double-clicking the block to open it, click the **Parameters** icon and set **Number of axes** to 2.

Connect the blocks as shown in the figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to 12. Running the model produces a scope image similar to the one above. (To make the axis ranges and title exactly match those in the figure, right-click each plot area in the scope and select **Axes properties**.)

Scalar Quantization Example 2

This example, shown in the figure below, illustrates the nature of scalar quantization more clearly. It samples and quantizes a sine wave and then plots the original (top) and quantized (bottom) signals. The plot contrasts the smooth sine curve with the polygonal curve of the quantized signal. The vertical coordinate of each flat part of the polygonal curve is a value in the **Quantization codebook** vector.





To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

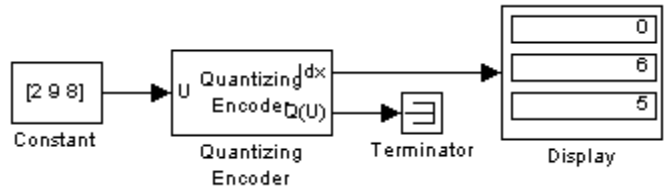
- Sine Wave, in the Simulink Sources library (*not* the Sine Wave block in the Signal Processing Sources library)
- Zero-Order Hold, in the Simulink Discrete library
  - Set **Sample time** to `0.1`.
- Quantizing Encoder
  - Set **Quantization partition** to `[-1:.2:1]`.
  - Set **Quantization codebook** to `[-1.1:.2:1.1]`.
- Terminator, in the Simulink Sinks library
- Scope, in the Simulink Sinks library
  - After double-clicking the block to open it, click the **Parameters** icon and set **Number of axes** to `2`.

Connect the blocks as shown in the figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `2*pi`. Running the model produces the scope image as shown above. (To make the axis ranges and title exactly

match those in the figure, right-click each plot area in the scope and select **Axes properties**.)

Determining Which Interval Each Input Is In

The Quantizing Encoder block also returns a signal, at the first output port, that tells which interval each input is in. For example, the model below shows that the input entries lie within the intervals labeled `0`, `6`, and `5`, respectively. Here, the 0th interval consists of real numbers less than or equal to 3; the 6th interval consists of real numbers greater than 8 but less than or equal to 9; and the 5th interval consists of real numbers greater than 7 but less than or equal to 8.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Constant, in the Simulink Sources library
  - Set **Constant value** to `[2, 9, 8]`.
- Quantizing Encoder
  - Set **Quantization partition** to `[3, 4, 5, 6, 7, 8, 9]`.
  - Set **Quantization codebook** to any vector whose length exceeds the length of **Quantization Partition** by one.
- Terminator, in the Simulink Sinks library
- Display, in the Simulink Sinks library
  - Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as shown above. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters

dialog box, set **Stop time** to 10. Running the model produces the display numbers as shown in the figure.

You can continue this example by branching the first output of the Quantizing Encoder block, connecting one branch to the input port of the Quantizing Decoder block, and connecting the output of the Quantizing Decoder block to another Display block. If the two source coding blocks' **Quantization codebook** parameters match, the output of the Quantizing Decoder block is the same as the second output of the Quantizing Encoder block. Thus the Quantizing Decoder block partially duplicates the functionality of the Quantizing Encoder block, but requires different input data and fewer parameters.

Companding a Signal

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

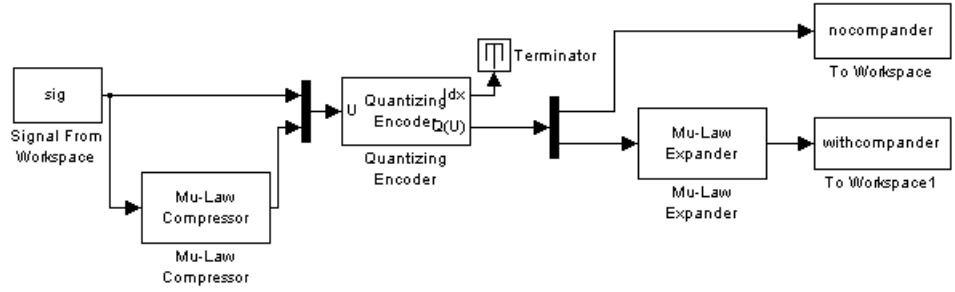
This blockset supports two kinds of companders: μ-law and A-law companders. The reference pages for the A-Law Compressor, A-Law Expander, Mu-Law Compressor, and Mu-Law Expander blocks list the relevant expander and compressor laws.

Example: Using a μ-Law Compander

This example quantizes an exponential signal in two ways and compares the resulting mean-square distortions. To create the signal in the MATLAB workspace, execute these commands:

```
sig = -4:.1:4;
sig = exp(sig'); % Exponential signal to quantize
```

The model in the following figure performs two computations. One computation uses the Quantizing Encoder block with a partition consisting of length-one intervals. The second computation uses the Mu-Law Compressor block to implement a μ-law compressor, the Quantizing Encoder block to quantize the compressed data and, finally, the Mu-Law Expander block to expand the quantized data.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Signal From Workspace, in the Signal Processing Sources library
  - Set **Signal** to `sig`.
- Mu-Law Compressor
  - Set **Peak signal magnitude** to `max(sig)`.
- Mux, in the Simulink Signal Routing library
- Quantizing Encoder, in the Source Coding library
  - Set **Quantization partition** to `0:floor(max(sig))`.
  - Set **Quantization codebook** to `0:ceil(max(sig))`.
- Terminator, in the Simulink Sinks library
- Demux, in the Simulink Signal Routing library
- Mu-Law Expander
  - Set **Peak signal magnitude** to `ceil(max(sig))`.
- Two copies of To Workspace, in the Simulink Sinks library
  - Set **Variable name** to `nocompander` and `withcompander`, respectively, in the two copies of this block.
  - Set **Save format** to `Array` in each of the two copies of this block.

Connect the blocks as shown above. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `80`. Run the model and execute these commands:

```
distor = sum((nocompander-sig).^2)/length(sig);
distor2 = sum((withcompander-sig).^2)/length(sig);
[distor distor2]

ans =

    0.5348    0.0397
```

This output shows that the distortion is smaller for the second scheme. This is because equal-length intervals are well suited to the logarithm of the data but not as well suited to the data itself.

Selected Bibliography for Source Coding

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, 6th edition, Upper Saddle River, NJ, Prentice Hall, 2001.

[2] Kondoz, A.M., *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.

[3] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice Hall, 1988.

Block Coding

Error-control coding techniques detect and possibly correct errors that occur when messages are transmitted in a digital communication system. To accomplish this, the encoder transmits not only the information symbols but also extra redundant symbols. The decoder interprets what it receives, using the redundant symbols to detect and possibly correct whatever errors occurred during transmission. You might use error-control coding if your transmission channel is very noisy or if your data is very sensitive to noise. Depending on the nature of the data or noise, you might choose a specific type of error-control coding.

Block coding is a special case of error-control coding. Block-coding techniques map a fixed number of message symbols to a fixed number of code symbols. A block coder treats each block of data independently and is a memoryless device.

Organization of this Section

These topics provide background information:

These topics describe how to simulate linear block coding:

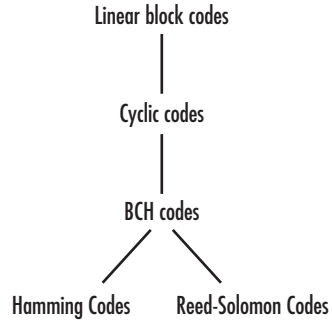
For background material on the subject of block coding, see the works listed in "Selected Bibliography for Block Coding" on page 1-59.

Accessing Block-Coding Blocks

You can open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library. Then open the Block sublibrary by double-clicking its icon in the Error Detection and Correction library.

Block-Coding Features of the Blockset

The class of block-coding techniques includes categories shown in the diagram below.



The Communications Blockset supports general linear block codes. It also includes blocks that process cyclic, BCH, Hamming, and Reed-Solomon codes (which are all special kinds of linear block codes). Blocks in the blockset can encode or decode a message using one of the techniques mentioned above. The Reed-Solomon and BCH decoders indicate how many errors they detected while decoding. The Reed-Solomon coding blocks also let you decide whether to use symbols or bits as your data.

Note The blocks in this blockset are designed for error-control codes that use an alphabet having 2 or 2^m symbols.

Communications Toolbox Support Functions

Functions in the Communications Toolbox can support the Communications Blockset simulation blocks by

- Determining characteristics of a technique, such as error-correction capability or possible message lengths
- Performing lower-level computations associated with a technique, such as
  - Computing a truth table
  - Computing a generator or parity-check matrix
  - Converting between generator and parity-check matrices
  - Computing a generator polynomial

For more information about error-control coding capabilities of the Communications Toolbox, see “Block Coding” in the Communications Toolbox User’s Guide.

Channel-Coding Terminology

Throughout this section, the information to be encoded consists of *message* symbols and the code that is produced consists of *codewords*.

Each block of K message symbols is encoded into a codeword that consists of N message symbols. K is called the message length, N is called the codeword length, and the code is called an [N,K] code.

Data Formats for Block Coding

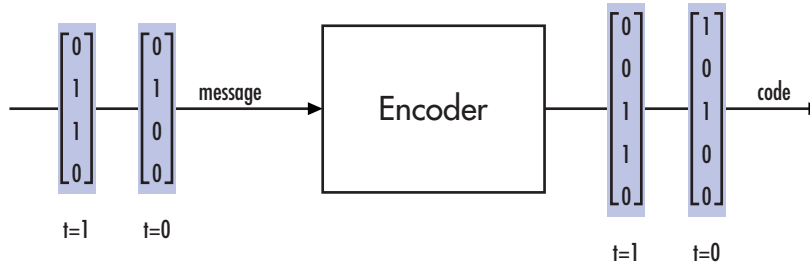
Each message or codeword is an ordered grouping of symbols. Each block in the Block Coding sublibrary processes one word in each time step, as described in the following section, “Binary Format (All Coding Methods)” on page 1-48. Reed-Solomon coding blocks also let you choose between binary and integer data, as described in “Integer Format (Reed-Solomon Only)” on page 1-50.

Binary Format (All Coding Methods)

You can structure messages and codewords as binary *vector* signals, where each vector represents a message word or a codeword. At a given time, the

encoder receives an entire message word, encodes it, and outputs the entire codeword. The message and code signals share the same sample time.

The figure below illustrates this situation. In this example, the encoder receives a four-bit message and produces a five-bit codeword at time 0. It repeats this process with a new message at time 1.



For all coding techniques *except* Reed-Solomon using binary input, the message vector must have length K and the corresponding code vector has length N. For Reed-Solomon codes with binary input, the symbols for the code are binary sequences of length M, corresponding to elements of the Galois field GF(2^M). In this case, the message vector must have length M*K and the corresponding code vector has length M*N. The Binary-Input RS Encoder block and the Binary-Output RS Decoder block use this format for messages and codewords.

If the input to a block-coding block is a frame-based vector, it must be a column vector instead of a row vector.

To produce sample-based messages in the binary format, you can configure the Bernoulli Binary Generator block so that its **Probability of a zero** parameter is a vector whose length is that of the signal you want to create. To produce frame-based messages in the binary format, you can configure the same block so that its **Probability of a zero** parameter is a scalar and its **Samples per frame** parameter is the length of the signal you want to create.

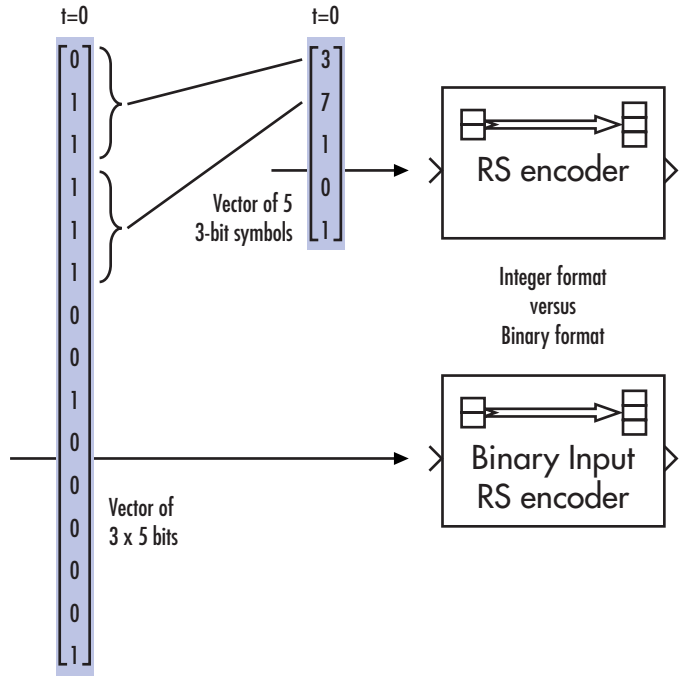
Using Serial Signals. If you prefer to structure messages and codewords as scalar signals, where several samples jointly form a message word or codeword, you can use the Buffer and Unbuffer blocks in the Signal Processing Blockset. Be aware that buffering involves latency and multirate processing. See the reference page for the Buffer block for more details. If your model computes error rates, the initial delay in the coding-buffering combination influences the **Receive delay** parameter in the Error Rate Calculation block. If you are unsure about the sample times of signals in your model, selecting **Sample time colors** from the **Port/signal displays** submenu of the model's **Format** menu, or attaching Probe blocks (from the Simulink Signal Attributes library) to connector lines might help.

Integer Format (Reed-Solomon Only)

A message word for an [N,K] Reed-Solomon code consists of M*K bits, which you can interpret as K symbols between 0 and 2^M. The symbols are binary sequences of length M, corresponding to elements of the Galois field GF(2^M), in descending order of powers. The integer format for Reed-Solomon codes lets you structure messages and codewords as *integer* signals instead of binary signals. (The input must be a frame-based column vector.)

Note In this context, Simulink expects the *first* bit to be the most significant bit in the symbol. "First" means the smallest index in a vector or the smallest time for a series of scalars.

The following figure illustrates the equivalence between binary and integer signals for a Reed-Solomon encoder. The case for the decoder is similar.



To produce sample-based messages in the integer format, you can configure the Random Integer Generator block so that **M-ary number** and **Initial seed** parameters are vectors of the desired length and all entries of the **M-ary number** vector are 2^M. To produce frame-based messages in the integer format, you can configure the same block so that its **M-ary number** and **Initial seed** parameters are scalars and its **Samples per frame** parameter is the length of the signal you want to create.

Using Block Encoders and Decoders Within a Model

Once you have configured the coding blocks, a few tips can help you place them correctly within your model:

- If a block has multiple outputs, the first one is always the stream of coding data.

  The Reed-Solomon and BCH blocks have an error counter as a second output.

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

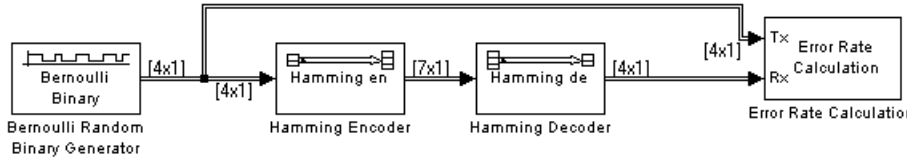
  If you are unsure about the size of signals in your model, selecting **Signal dimensions** from the **Port/signal displays** submenu of the model's **Format** menu might help.

Examples of Block Coding

This section presents two example models. The first example processes a Hamming code using the binary format, and the second example processes a Reed-Solomon code using the integer format.

Example: Hamming Code in Binary Format

This example shows very simply how to use an encoder and decoder. It illustrates the appropriate vector lengths of the code and message signals for the coding blocks. Because the Error Rate Calculation block accepts only scalars or frame-based column vectors as the transmitted and received signals, this example uses frame-based column vectors throughout. (It thus avoids having to change signal attributes using a block such as Convert 1-D to 2-D.)



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

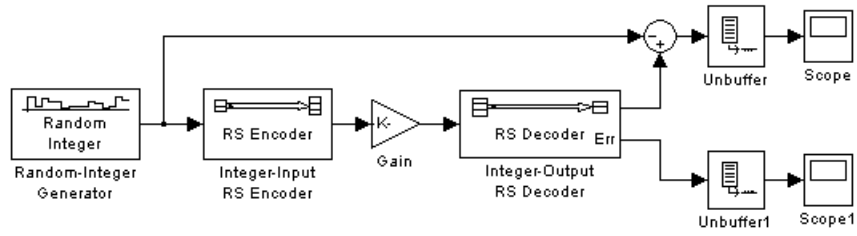
- Bernoulli Binary Generator, in the Comm Sources library
  - Set **Probability of a zero** to `.5`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
  - Check the **Frame-based outputs** check box.
  - Set **Samples per frame** to 4.

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

Connect the blocks as in the preceding figure. Use the **Signal dimensions** feature from the **Port/signal displays** submenu of the model window's **Format** menu. After updating the diagram if necessary (**Update diagram** from the **Edit** menu), the connector lines show relevant signal attributes. The connector lines are double lines to indicate frame-based signals, and the annotations next to the lines show that the signals are column vectors of appropriate sizes.

Example: Reed-Solomon Code in Integer Format

This example uses a Reed-Solomon code in integer format. It illustrates the appropriate vector lengths of the code and message signals for the coding blocks. It also exhibits error correction, using a very simple way of introducing errors into each codeword.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

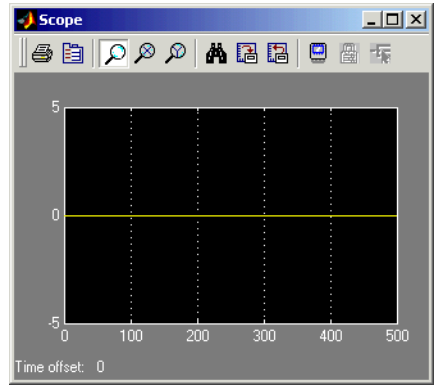
- Random Integer Generator, in the Comm Sources library
  - Set **M-ary number** to 15.
  - Set **Initial seed** to any prime number greater than 30, preferably the output of the `randseed` function.
  - Check the **Frame-based outputs** check box.

  - Set **Samples per frame** to `5`.
- Integer-Input RS Encoder
  - Set **Codeword length N** to `15`.
  - Set **Message length K** to `5`.
- Gain, in the Simulink Math Operations library
  - Set **Gain** to `[0; 0; 0; 0; 0; ones(10,1)]`.
- Integer-Output RS Decoder
  - Set **Codeword length N** to `15`.
  - Set **Message length K** to `5`.
- Scope, in the Simulink Sinks library. Get two copies of this block.
- Sum, in the Simulink Math Operations library
  - Set **List of signs** to `|-+`

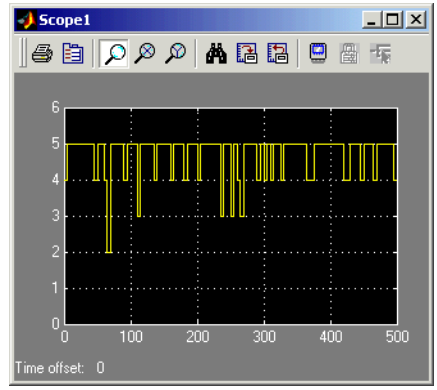
Connect the blocks as in the preceding figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `500`.

The vector length numbers appear on the connecting lines only if you select **Signal dimensions** from the **Port/signal displays** submenu of the model's **Format** menu. The encoder accepts a vector of length 5 (which is K in this case) and produces a vector of length 15 (which is N in this case). The decoder does the opposite. The **Initial seed** parameter in the Random Integer Generator block is a vector of length 5 because it must generate a message word of length 5.

Running the model produces the following scope images. Your plot of the error counts might differ somewhat, depending on your **Initial seed** value in the Random Integer Generator block. (To make the axis range exactly match that of the first scope, right-click the plot area in the scope and select **Axes properties**.)



Difference Between Original Message and Recovered Message



Number of Errors Before Correction

The second plot is the number of errors that the decoder detected while trying to recover the message. Often the number is five because the Gain block replaces the first five symbols in each codeword with zeros. However, the number of errors is less than five whenever a correct codeword contains one or more zeros in the first five places.

The first plot is the difference between the original message and the recovered message; since the decoder was able to correct all errors that occurred, each of the five data streams in the plot is zero.

Notes on Specific Block-Coding Techniques

Although the Block Coding sublibrary is somewhat uniform in its look and feel, the various coding techniques are not identical. This section describes special options and restrictions that apply to parameters and signals for the coding technique categories in this sublibrary. Read the part that applies to the coding technique you want to use: generic linear block code, cyclic code, Hamming code, BCH code, or Reed-Solomon code.

Generic Linear Block Codes

Encoding a message using a generic linear block code requires a generator matrix. Decoding the code requires the generator matrix and possibly a truth table. In order to use the Binary Linear Encoder and Binary Linear Decoder blocks, you must understand the **Generator matrix** and **Error-correction truth table** parameters.

Generator Matrix. The process of encoding a message into an [N,K] linear block code is determined by a K-by-N generator matrix G. Specifically, a 1-by-K message vector v is encoded into the 1-by-N codeword vector vG. If G has the form $[I_k, P]$ or $[P, I_k]$, where P is some K-by-(N-K) matrix and I_k is the K-by-K identity matrix, G is said to be in *standard form*. (Some authors, such as Clark and Cain [1], use the first standard form, while others, such as Lin and Costello [2], use the second.) The linear block-coding blocks in this blockset require the **Generator matrix** mask parameter to be in standard form.

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

The Binary Linear Decoder block allows you to specify a decoding table in the **Error-correction truth table** parameter. Represent a decoding table as a matrix with N columns and 2^{N-K} rows. Each row gives a correction vector for one received codeword vector.

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

Cyclic Codes

For cyclic codes, the codeword length N must have the form 2^M-1, where M is an integer greater than or equal to 3.

Generator Polynomials. Cyclic codes have special algebraic properties that allow a polynomial to determine the coding process completely. This so-called generator polynomial is a degree-(N-K) divisor of the polynomial x^N-1. Van Lint [4] explains how a generator polynomial determines a cyclic code.

The Binary Cyclic Encoder and Binary Cyclic Decoder blocks allow you to specify a generator polynomial as the second mask parameter, instead of specifying K there. The blocks represent a generator polynomial using a vector that lists the polynomial's coefficients in order of *ascending* powers of the variable. You can find generator polynomials for cyclic codes using the `cyclpoly` function in the Communications Toolbox.

If you do not want to specify a generator polynomial, set the second mask parameter to the value of K.

Hamming Codes

For Hamming codes, the codeword length N must have the form 2^M-1, where M is an integer greater than or equal to 3. The message length K must equal N-M.

Primitive Polynomials. Hamming codes rely on algebraic fields that have 2^M elements (or, more generally, p^M elements for a prime number p). Elements of such fields are named *relative to* a distinguished element of the field that is called a *primitive element*. The minimal polynomial of a primitive element is called a *primitive polynomial*. The Hamming Encoder and Hamming Decoder blocks allow you to specify a primitive polynomial for the finite field that they use for computations. If you want to specify this polynomial, do so in the second mask parameter field. The blocks represent a primitive polynomial using a vector that lists the polynomial's coefficients in order of *ascending* powers of the variable. You can find generator polynomials for Galois fields using the `gfprimfd` function in the Communications Toolbox.

If you do not want to specify a primitive polynomial, set the second mask parameter to the value of K.

BCH Codes

For BCH codes, the codeword length N must have the form 2^M-1, where M is an integer between 3 and 9. The message length K is restricted to particular values that depend on N. To see which values of K are valid for a given N, see the reference page for the `bchenc` function in the Communications Toolbox. No known analytic formula describes the relationship among the codeword length, message length, and error-correction capability for BCH codes.

Error Information. The BCH Decoder block can also return error-related information during the simulation. The optional second output signal indicates the number of errors that the block detected in the input codeword. A negative integer in the second output indicates that the block detected more errors than it could correct using the coding scheme. If you do not want the block to create a second output signal, clear **Show number of errors** in the block dialog box.

Reed-Solomon Codes

Reed-Solomon codes are useful for correcting errors that occur in bursts. In the simplest case, the length of codewords in a Reed-Solomon code is of the form N= 2^M-1, where the 2^M is the number of symbols for the code. The error-correction capability of a Reed-Solomon code is `floor((N-K)/2)`, where K is the length of message words. The difference N-K must be even.

It is sometimes convenient to use a shortened Reed-Solomon code in which N is less than 2^M-1. In this case, the encoder appends 2^M-1-N zero symbols to each message word and codeword. The error-correction capability of a shortened Reed-Solomon code is also `floor((N-K)/2))`. The Communications Blockset Reed-Solomon blocks can implement shortened Reed-Solomon codes.

Effect of Nonbinary Symbols. One difference between Reed-Solomon codes and the other codes supported in this blockset is that Reed-Solomon codes process symbols in GF(2^M) instead of GF(2). Each such symbol is specified by M bits. The nonbinary nature of the Reed-Solomon code symbols causes the Reed-Solomon blocks to differ from other coding blocks in these ways:

- You can use the integer format, via the Integer-Input RS Encoder and Integer-Output RS Decoder blocks.

- The binary format expects the vector lengths to be an integer multiple of M*K (not K) for messages and the same integer M*N (not N) for codewords.

Error Information. The Reed-Solomon decoding blocks (Binary-Output RS Decoder and Integer-Output RS Decoder) return error-related information during the simulation. The second output signal indicates the number of errors that the block detected in the input codeword. A -1 in the second output indicates that the block detected more errors than it could correct using the coding scheme.

Selected Bibliography for Block Coding

[1] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

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

[3] Peterson, W. Wesley, and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed., Cambridge, MA, MIT Press, 1972.

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

Convolutional Coding

Convolutional coding is a special case of error-control coding. Unlike a block coder, a convolutional coder is not a memoryless device. Even though a convolutional coder accepts a fixed number of message symbols and produces a fixed number of code symbols, its computations depend not only on the current set of input symbols but on some of the previous input symbols.

The following topics are covered:

Organization of this Section

- How to access the Convolutional sublibrary of Error Detection and Correction
- The software's capabilities
- Parameters for convolutional coding (polynomial description and trellis description)
- The use of the coding blocks for a rate 2/3 code
- How to implement a systematic encoder
- Soft-decision decoding

For background material on the subject of convolutional coding, see the works listed in "Selected Bibliography for Convolutional Coding" on page 1-74.

Accessing Convolutional-Coding Blocks

Open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library. Open the Convolutional sublibrary by double-clicking its icon in the Error Detection and Correction library.

Convolutional-Coding Features of the Blockset

The Communications Blockset supports feedforward or feedback binary convolutional codes that can be described by a trellis structure or a set of generator polynomials. It uses the Viterbi algorithm to implement hard-decision and soft-decision decoding.

The blockset also includes an *a posteriori* probability decoder, which can be used for soft output decoding of convolutional codes.

Parameters for Convolutional Coding

To process convolutional codes, use the Convolutional Encoder, Viterbi Decoder, and/or APP Decoder blocks in the Convolutional sublibrary. If a mask parameter is required in both the encoder and the decoder, use the same value in both blocks.

The blocks in the Convolutional sublibrary assume that you use one of two different representations of a convolutional encoder:

- If you design your encoder using a diagram with shift registers and modulo-2 adders, you can compute the code generator polynomial matrix and subsequently use the `poly2trellis` function (in the Communications Toolbox) to generate the corresponding trellis structure mask parameter automatically. For an example, see "Example: A Rate 2/3 Feedforward Encoder" on page 1-62.
- If you design your encoder using a trellis diagram, you can construct the trellis structure in MATLAB and use it as the mask parameter.

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

Using the Polynomial Description in Blocks

To use the polynomial description with the Convolutional Encoder, Viterbi Decoder, or APP Decoder blocks, use the utility function `poly2trellis` from the Communications Toolbox. This function accepts a polynomial description and converts it into a trellis description. For example, the following command computes the trellis description of an encoder whose constraint length is 5 and whose generator polynomials are 35 and 31:

```
trellis = poly2trellis(5,[35 31]);
```

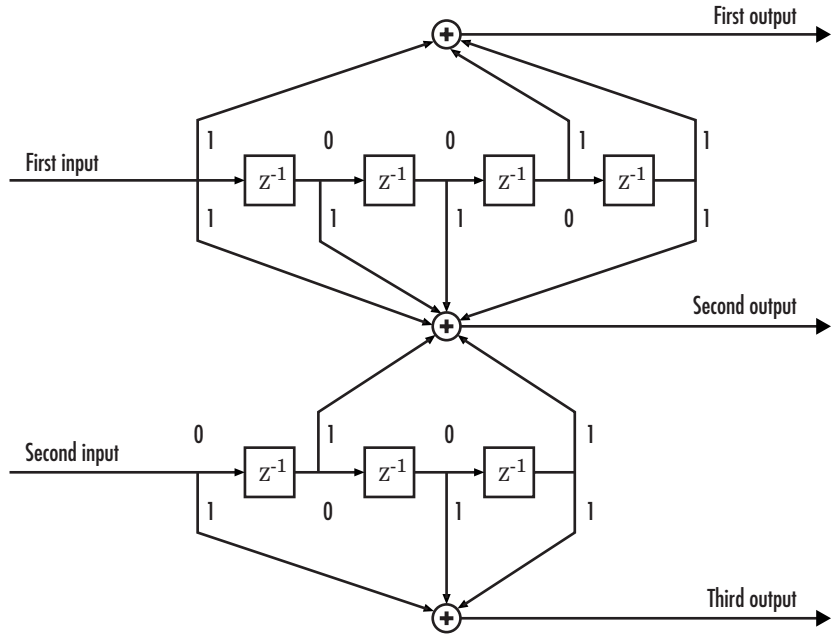
To use this encoder with one of the convolutional-coding blocks, simply place a `poly2trellis` command such as

```
poly2trellis(5,[35 31]);
```

in the **Trellis structure** parameter field.

Example: A Rate 2/3 Feedforward Encoder

This example uses the rate 2/3 feedforward convolutional encoder depicted in the following figure. The description explains how to determine the coding blocks' parameters from a schematic of a rate 2/3 feedforward encoder. This example also illustrates the use of the Error Rate Calculation block with a receive delay.



How to Determine Coding Parameters

The Convolutional Encoder and Viterbi Decoder blocks can implement this code if their parameters have the appropriate values.

The encoder's constraint length is a vector of length 2 since the encoder has two inputs. The elements of this vector indicate the number of bits stored in

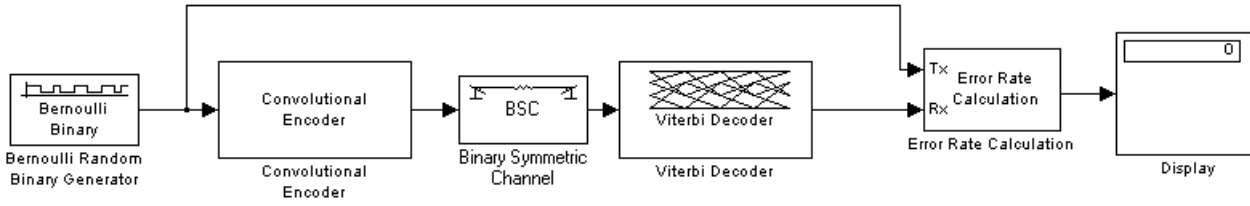
each shift register, including the current input bits. Counting memory spaces in each shift register in the diagram and adding one for the current inputs leads to a constraint length of [5 4].

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, notice that the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13. The full value of the code generator matrix is [27 33 0; 0 5 13].

To use the constraint length and code generator parameters in the Convolutional Encoder and Viterbi Decoder blocks, use the `poly2trellis` function to convert those parameters into a trellis structure.

How to Simulate the Encoder

The following model simulates this encoder.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Comm Sources library
  - Set **Probability of a zero** to `.5`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
  - Set **Sample time** to `.5`.

  - Check the **Frame-based outputs** check box.
  - Set **Samples per frame** to `2`.
- Convolutional Encoder
  - Set **Trellis structure** to `poly2trellis([5 4],[23 35 0; 0 5 13])`.
- Binary Symmetric Channel, in the Channels library
  - Set **Error probability** to `0.02`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
  - Clear the **Output error vector** check box.
- Viterbi Decoder
  - Set **Trellis structure** to `poly2trellis([5 4],[23 35 0; 0 5 13])`.
  - Set **Decision type** to `Hard Decision`.
- Error Rate Calculation, in the Comm Sinks library
  - Set **Receive delay** to `68`.
  - Set **Output data** to `Port`.
  - Check the **Stop simulation** check box.
  - Set **Target number of errors** to `100`.
- Display, in the Simulink Sinks library
  - Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as in the figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `inf`.

Notes on the Model

The matrix size annotations appear on the connecting lines only if you select **Signal Dimensions** from the **Port/signal displays** submenu of the model's **Format** menu. The encoder accepts a 2-by-1 frame-based vector and produces a 3-by-1 frame-based vector, while the decoder does the opposite.

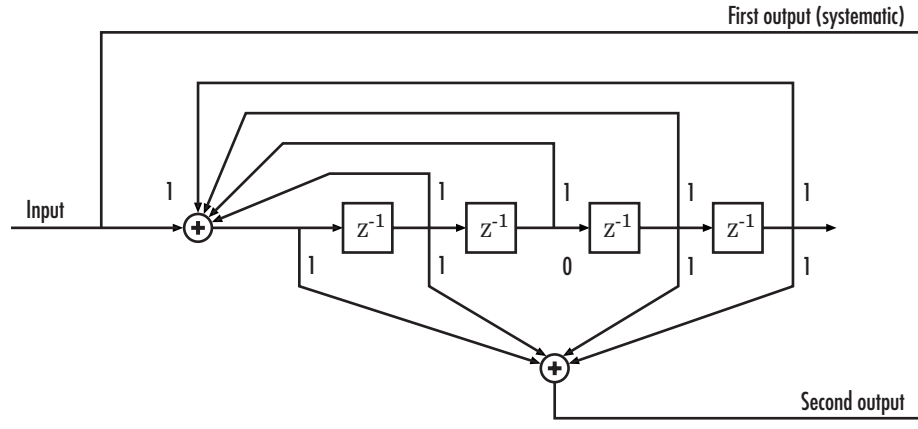
The **Samples per frame** parameter in the Bernoulli Binary Generator block is 2 because the block must generate a message word of length 2.

The **Receive delay** parameter in the Error Rate Calculation block is 68, which is the vector length (2) of the recovered message times the **Traceback depth** value (34) in the Viterbi Decoder block. If you examine the transmitted and received signals as matrices in the MATLAB workspace, you see that the first 34 rows of the recovered message consist of zeros, while subsequent rows are the decoded messages. Thus the delay in the received signal is 34 vectors of length 2, or 68 samples.

Running the model produces display output consisting of three numbers: the error rate, the total number of errors, and the total number of comparisons that the Error Rate Calculation block makes during the simulation. (The first two numbers vary depending on your **Initial seed** values in the Bernoulli Binary Generator and Binary Symmetric Channel blocks.) The simulation stops after 100 errors occur, because **Target number of errors** is set to `100` in the Error Rate Calculation block. The error rate is much less than `0.02`, the **Error probability** in the Binary Symmetric Channel block.

Implementing a Systematic Encoder with Feedback

This section explains how to use the Convolutional Encoder block to implement a systematic encoder with feedback. A code is *systematic* if the actual message words appear as part of the codewords. The following diagram shows an example of a systematic encoder.



To implement this encoder, set the **Trellis structure** parameter in the Convolutional Encoder block to `poly2trellis(5, [37 33], 37)`. This setting corresponds to

- Constraint length: `5`
- Generator polynomial pair: `[37 33]`
- Feedback polynomial: `37`

The feedback polynomial is represented by the binary vector [1 1 1 1 1], corresponding to the upper row of binary digits. These digits indicate connections from the outputs of the registers to the adder. The initial 1 corresponds to the input bit. The octal representation of the binary number 11111 is 37.

To implement a systematic code, set the first generator polynomial to be the same as the feedback polynomial in the **Trellis structure** parameter of the Convolutional Encoder block. In this example, both polynomials have the octal representation 37.

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

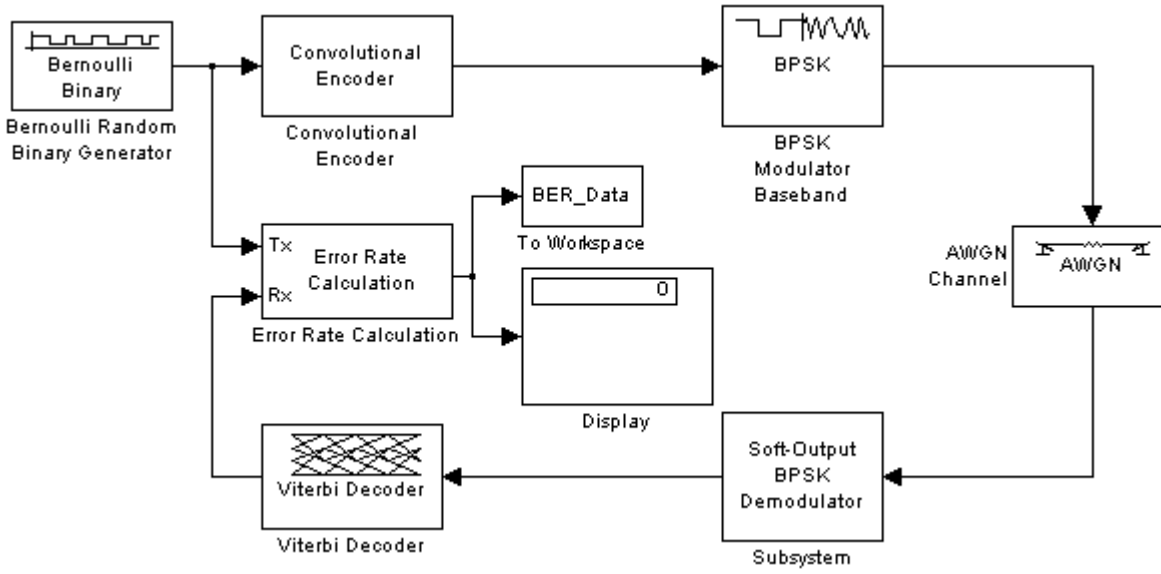
For more information on setting the mask parameters for the Convolutional Encoder block, see “Polynomial Description of a Convolutional Encoder” in the Communications Toolbox documentation.

Example: Soft-Decision Decoding

This example creates a rate 1/2 convolutional code. It uses a quantizer and the Viterbi Decoder block to perform soft-decision decoding. This description covers these topics:

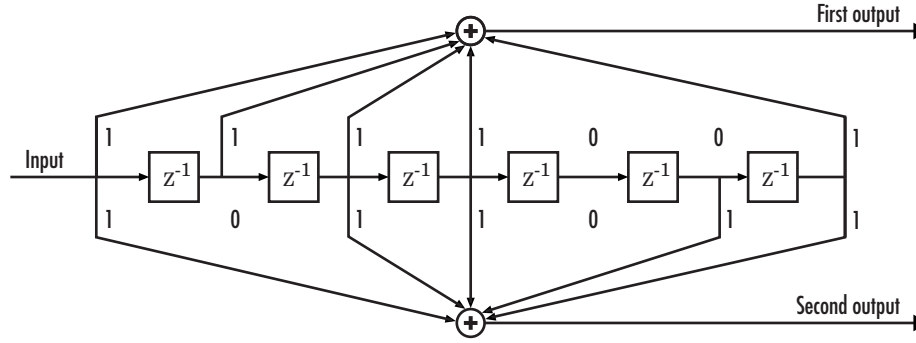
Overview of the Simulation

The model is in the following figure. To open the model, click here in the MATLAB Help browser. The simulation creates a random binary message signal, encodes the message into a convolutional code, modulates the code using the binary phase shift keying (BPSK) technique, and adds white Gaussian noise to the modulated data in order to simulate a noisy channel. Then, the simulation prepares the received data for the decoding block and decodes. Finally, the simulation compares the decoded information with the original message signal in order to compute the bit error rate. The simulation ends after processing 100 bit errors or 10^7 message bits, whichever comes first.



Defining the Convolutional Code

The feedforward convolutional encoder in this example is depicted below.



The encoder's constraint length is a scalar since the encoder has one input. The value of the constraint length is the number of bits stored in the shift register, including the current input. There are six memory registers, and the current input is one bit. Thus the constraint length of the code is 7.

The code generator is a 1-by-2 matrix of octal numbers because the encoder has one input and two outputs. The first element in the matrix indicates which input values contribute to the first output, and the second element in the matrix indicates which input values contribute to the second output.

For example, the first output in the encoder diagram is the modulo-2 sum of the rightmost and the four leftmost elements in the diagram's array of input values. The seven-digit binary number 1111001 captures this information, and is equivalent to the octal number 171. The octal number 171 thus becomes the first entry of the code generator matrix. Here, each triplet of bits uses the leftmost bit as the most significant bit. The second output corresponds to the binary number 1011011, which is equivalent to the octal number 133. The code generator is therefore [171 133].

The **Trellis structure** parameter in the Convolutional Encoder block tells the block which code to use when processing data. In this case, the `poly2trellis` function, in the Communications Toolbox, converts the constraint length and the pair of octal numbers into a valid trellis structure.

While the message data entering the Convolutional Encoder block is a scalar bit stream, the encoded data leaving the block is a stream of binary vectors of length 2.

Mapping the Received Data

The received data, that is, the output of the AWGN Channel block, consists of complex numbers that are close to -1 and 1. In order to reconstruct the original binary message, the receiver part of the model must decode the convolutional code. The Viterbi Decoder block in this model expects its input data to be integers between 0 and 7. The demodulator, a custom subsystem in this model, transforms the received data into a format that the Viterbi Decoder block can interpret properly. More specifically, the demodulator subsystem

- Converts the received data signal to a real signal by removing its imaginary part. It is reasonable to assume that the imaginary part of the received data does not contain essential information, because the imaginary part of the transmitted data is zero (ignoring small roundoff errors) and because the channel noise is not very powerful.
- Normalizes the received data by dividing by its running standard deviation and then multiplying by -1.

- Quantizes the normalized data using three bits.

The combination of this mapping and the Viterbi Decoder block's decision mapping reverses the BPSK modulation that the BPSK Modulator Baseband block performs on the transmitting side of this model. To examine the demodulator subsystem in more detail, double-click the icon labeled Soft-Output BPSK Demodulator.

Decoding the Convolutional Code

After the received data is properly mapped to length-2 vectors of 3-bit decision values, the Viterbi Decoder block decodes it. The block uses a soft-decision algorithm with 2^3 different input values because the **Decision type** parameter is `Soft Decision` and the **Number of soft decision bits** parameter is `3`.

Soft-Decision Interpretation of Data. When the **Decision type** parameter is set to `Soft Decision`, the Viterbi Decoder block requires input values between 0 and 2^b-1, where b is the **Number of soft decision bits** parameter. The block interprets 0 as the most confident decision that the codeword bit is a 0 and interprets 2^b-1 as the most confident decision that the codeword bit is a 1. The values in between these extremes represent less confident decisions. The following table lists the interpretations of the eight possible input values for this example.

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

Traceback and Decoding Delay. The **Traceback depth** parameter in the Viterbi Decoder block represents the length of the decoding delay. Typical values for a traceback depth are about five or six times the constraint length, which would be 35 or 42 in this example. However, some hardware implementations offer options of 48 and 96. This example chooses 48 because that is closer to the targets (35 and 42) than 96 is.

Delay in Received Data

The Error Rate Calculation block's **Receive delay** parameter is nonzero because a given message bit and its corresponding recovered bit are separated in time by a nonzero amount of simulation time. The **Receive delay** parameter tells the block which elements of its input signals to compare when checking for errors.

In this case, the **Receive delay** value is 49 samples, which is one more than the **Traceback depth** value (48) in the Viterbi Decoder block. The extra one-sample delay comes from the initial delay in the Buffer block. Because the Buffer block must collect two scalar samples before it can output one vector, its first meaningful output occurs at time 1 second, not time 0.

Comparing Simulation Results with Theoretical Results

This section describes how to compare the bit error rate in this simulation with the bit error rate that would theoretically result from unquantized decoding. The process includes a few steps, described in these sections:

- "Computing Theoretical Bounds for the Bit Error Rate" on page 1-71
- "Simulating Multiple Times to Collect Bit Error Rates" on page 1-73

Computing Theoretical Bounds for the Bit Error Rate. To calculate theoretical bounds for the bit error rate P_b of the convolutional code in this model, you can use this estimate based on unquantized-decision decoding:

$$P_b < \sum_{d=f}^{\infty} c_d P_d$$

In this estimate, c_d is the sum of bit errors for error events of distance d, and f is the free distance of the code. The quantity P_d is the pairwise error probability, given by

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

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

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

Values for the coefficients c_d and the free distance f are in published articles such as [4]. The free distance for this code is $f = 10$.

The following commands calculate the values of P_b for E_b/N_0 values in the range from 1 to 3.5, in increments of 0.5:

```
EbNoVec = [1:0.5:4.0];
R = 1/2;
% Errs is the vector of sums of bit errors for
% error events at distance d, for d from 10 to 29.
Errs = [36 0 211 0 1404 0 11633 0 77433 0 502690 0,...
        3322763 0 21292910 0 134365911 0 843425871 0];
% P is the matrix of pairwise error probilities, for
% Eb/No values in EbNoVec and d from 10 to 29.
P = zeros(20,7); % Initialize.
for d = 10:29
   P(d-9,:) = (1/2)*erfc(sqrt(d*R*10.^(EbNoVec/10)));
end
% Bounds is the vector of upper bounds for the bit error
% rate, for Eb/No values in EbNoVec.
Bounds = Errs*P;
```

Simulating Multiple Times to Collect Bit Error Rates. You can efficiently vary the simulation parameters by using the `sim` function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB, in increments of 0.5 dB. It collects all bit error rates from these simulations in the matrix `BERVec`. It also plots the bit error rates in a figure window along with the theoretical bounds computed in the preceding code fragment.

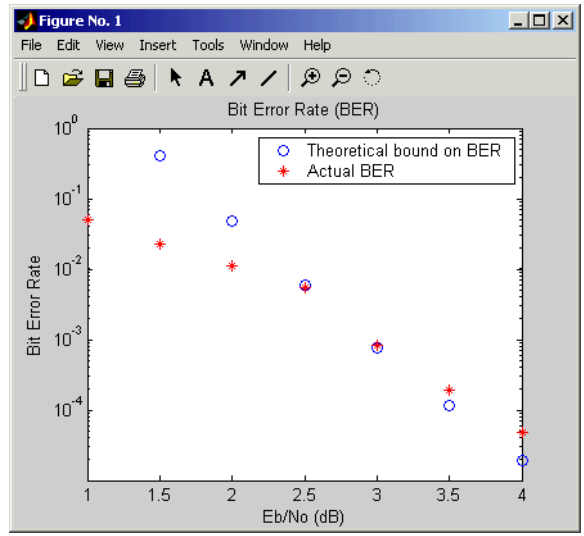
Note First open the model by clicking here in the MATLAB Help browser. Then execute these commands, which might take a few minutes.

```
% Plot theoretical bounds and set up figure.
figure;
semilogy(EbNoVec,Bounds,'bo',1,NaN,'r*');
xlabel('Eb/No (dB)'); ylabel('Bit Error Rate');
title('Bit Error Rate (BER)');
legend('Theoretical bound on BER','Actual BER');
axis([1 4 1e-5 1]);
hold on;

BERVec = [];
opts = simset('SrcWorkspace','Current',...
   'DstWorkspace','Current');
% Make the noise level variable.
set_param('doc_softdecision/AWGN Channel',...
    'EsNodB','EbNodB+10*log10(1/2)');
% Simulate multiple times.
for n = 1:length(EbNoVec)
    EbNodB = EbNoVec(n);
    sim('doc_softdecision',5000000,opts);
    BERVec(n,:) = BER_Data;
    semilogy(EbNoVec(n),BERVec(n,1),'r*'); % Plot point.
    drawnow;
end
hold off;
```

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

The plot of bit error rate against signal-to-noise ratio follows. The locations of your actual BER points might vary because the simulation involves random numbers.



Selected Bibliography for Convolutional Coding

[1] Benedetto, Sergio, and Guido Montorsi, "Performance of Continuous and Blockwise Decoded Turbo Codes," *IEEE Communications Letters*, Vol. 1, May 1997, pp.77–79.

[2] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara, "A Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes," *JPL TDA Progress Report*, Vol. 42-127, November 1996.

[3] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[4] Frenger, P., P. Orten, and T. Ottosson, "Convolution Codes with Optimum Distance Spectrum," *IEEE Communications Letters*, Vol. 3, November 1999, pp. 317–319.

[5] Gitlin, Richard D., Jeremiah F. Hayes, and Stephen B. Weinstein, *Data Communications Principles*, New York, Plenum, 1992.

[6] Heller, Jerrold A., and Irwin Mark Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Transactions on Communication Technology*, Vol. COM-19, October 1971, pp. 835–848.

[7] Viterbi, Andrew J., "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, Vol. 16, February 1998, pp. 260–264.

Cyclic Redundancy Check Coding

Cyclic redundancy check (CRC) coding is an error-control coding technique for detecting errors that occur when a message is transmitted. Unlike block or convolutional codes, CRC codes do not have a built-in error-correction capability. Instead, when an error is detected in a received message word, the receiver requests the sender to retransmit the message word.

In CRC coding, the transmitter applies a rule to each message word to create extra bits, called the *checksum,* or *syndrome*, and then appends the checksum to the message word. After receiving a transmitted word, the receiver applies the same rule to the received word. If the resulting checksum is nonzero, an error has occurred, and the transmitter should resend the message word.

This section covers the following topics:

Accessing CRC Blocks

Open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library. Open the CRC sublibrary by double-clicking on its icon in the Error Detection and Correction library.

CRC-Coding Features of the Blockset

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

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

The General CRC Generator block computes a checksum for each input frame, appends it to the message word, and transmits the result. The General CRC

Syndrome Detector block receives a transmitted word and calculates its checksum. The block has two outputs. The first is the message word without the transmitted checksum. The second output is a binary error flag, which is 0 if the checksum computed for the received word is zero, and 1 otherwise.

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

CRC Algorithm

The CRC algorithm accepts a binary data frame, corresponding to a polynomial M, and appends a checksum of r bits, corresponding to a polynomial C. The concatenation of the input frame and the checksum then corresponds to the polynomial $T = M*x^r + C$, since multiplying by x^r corresponds to shifting the input frame r bits to the left. The algorithm chooses the checksum C so that T is divisible by a predefined polynomial P of degree r, called the *generator polynomial*.

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

The General CRC Generator block and the CRC-N Generator block, which implement the transmission phase of the CRC algorithm, do the following:

1. Left shift the input data frame by r bits and divide the corresponding polynomial by P.
2. Set the checksum equal to the binary vector of length r, corresponding to the remainder from step 1.
3. Append the checksum to the input data frame. The result is the output frame.

The General CRC Syndrome Detector block and the CRC-N Syndrome Detector block implement the detection phase of the CRC algorithm. Each of

these blocks computes the checksum for its entire input frame, as described above. The block's second output port issues a 0 if the checksum computation yields a zero value, and a 1 otherwise.

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

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

Example

Suppose the input frame is `[1 1 0 0 1 1 0]'`, corresponding to the polynomial M = $x^6 + x^5 + x^2 + x$, and the generator polynomial is P = $x^3 + x^2 + 1$, of degree r = 3. By polynomial division, M*x^3 = ($x^6 + x^3 + x$)*P + x. The remainder is R = x, so that the checksum is then `[0 1 0]'`. An extra 0 is added on the left to make the checksum have length 3.

Selected Bibliography for CRC Coding

[1] Sklar, Bernard., *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice Hall, 1988.

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

Interleaving

An interleaver permutes symbols according to a mapping. A corresponding deinterleaver uses the inverse mapping to restore the original sequence of symbols. Interleaving and deinterleaving can be useful for reducing errors caused by burst errors in a communication system.

Open the Interleaving library by double-clicking its icon in the main Communications Blockset library. Then open the interleaving sublibraries by double-clicking their icons in the Interleaving library.

Interleaving Features of the Blockset

This blockset provides interleavers in two broad categories:

- Block interleavers. This category includes matrix, random, algebraic, and helical scan interleavers as special cases.
- Convolutional interleavers. This category includes a helical interleaver as a special case, as well as a general multiplexed interleaver.

In typical usage of all interleaver/deinterleaver pairs in this blockset, the parameters of the deinterleaver match those of the interleaver.

For background information about interleavers, see the works listed in “Selected Bibliography for Interleaving” on page 1-87.

Block Interleavers

A block interleaver accepts a set of symbols and rearranges them, without repeating or omitting any of the symbols in the set. The number of symbols in each set is fixed for a given interleaver. The interleaver’s operation on a set of symbols is independent of its operation on all other sets of symbols.

Types of Block Interleavers

The set of block interleavers in this library includes a general interleaver/deinterleaver pair as well as several special cases. Each special-case block uses the same computational code that its more general counterpart uses, but provides an interface that is more suitable for the special case.

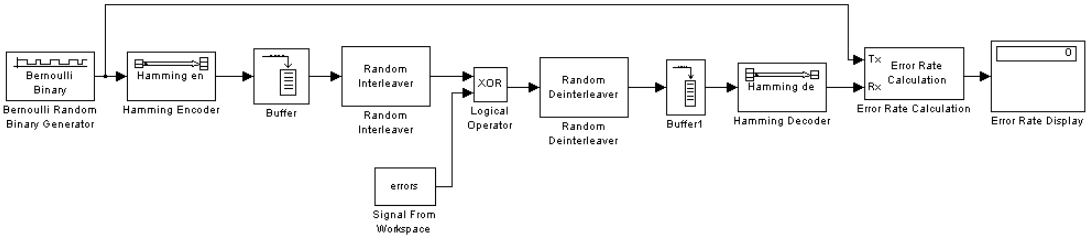
The Matrix Interleaver block accomplishes block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port column by column. For example, if the interleaver uses a 2-by-3 matrix to do its internal computations, then for an input of `[1 2 3 4 5 6]`, the block produces an output of `[1 4 2 5 3 6]`.

The Random Interleaver block chooses a permutation table randomly using the **Initial seed** parameter that you provide in the block mask. By using the same **Initial seed** value in the corresponding Random Deinterleaver block, you can restore the permuted symbols to their original ordering.

The Algebraic Interleaver block uses a permutation table that is algebraically derived. It supports Takeshita-Costello interleavers and Welch-Costas interleavers. These interleavers are described in [4].

Example: Block Interleavers

The following example shows how to use an interleaver to improve the error rate when the channel produces bursts of errors.



Before running the model, you must create a binary vector that simulates bursts of errors, as described in "Creating the Vector of Errors" on page 1-82. The Signal From Workspace block imports this vector from the MATLAB workspace into the model, where the Logical Operator block performs an XOR of the vector with the signal.

To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Random Data Sources sublibrary of the Comm Sources library

  - Check the box next to **Frame-based outputs**.
  - Set **Samples per frame** to `4`.
- Hamming Encoder, in the Block sublibrary of the Error Detection and Correction library. Use default parameters
- Buffer, in the Buffers sublibrary of the Signal Management library in the Signal Processing Blockset
  - Set **Output buffer size (per channel)** to `84`.
- Random Interleaver, in the Block sublibrary of the Interleaving library in the Communications Blockset
  - Set **Number of elements** to `84`.
- Logical Operator, in the Simulink Math Operations library
  - Set **Operator** to `XOR`.
- Signal From Workspace, in the Signal Processing Sources library
  - Set **Signal** to `errors`.
  - Set **Sample time** to `4/7`.
  - Set **Samples per frame** to `84`.
- Random Deinterleaver, in the Block sublibrary of the Interleaving library in the Communications Blockset
  - Set **Number of elements** to `84`.
- Buffer, in the Buffers sublibrary of the Signal Management library in the Signal Processing Blockset
  - Set **Output buffer size (per channel)** to `7`.
- Hamming Decoder, in the Block sublibrary of the Error Detection and Correction library. Use default parameters.
- Error Rate Calculation, in the Comm Sinks library
  - Set **Receive delay** to `(4/7)*84`.
  - Set **Computation delay** to `100`.
  - Set **Output data** to `Port`.
- Display, in the Simulink Sinks library. Use default parameters.

Select **Configuration parameters** from the model's **Simulation** menu and set **Stop time** to `length(errors)`.

Creating the Vector of Errors. Before running the model, use the following code to create a binary vector in the MATLAB workspace. The model uses this vector to simulate bursts of errors. The vector contains blocks of three 1s, representing bursts of errors, at random intervals. The distance between two consecutive blocks of 1s is a random integer between 1 and 80.

```
errors=zeros(1,10^4);
n=1;
while n<10^4-80;
n=n+floor(79*rand(1))+3;
errors(n:n+2)=[1 1 1];
end
```

To determine the ratio of the number of 1s to the total number of symbols in the vector `errors`, enter

```
sum(errors)/length(errors)
```

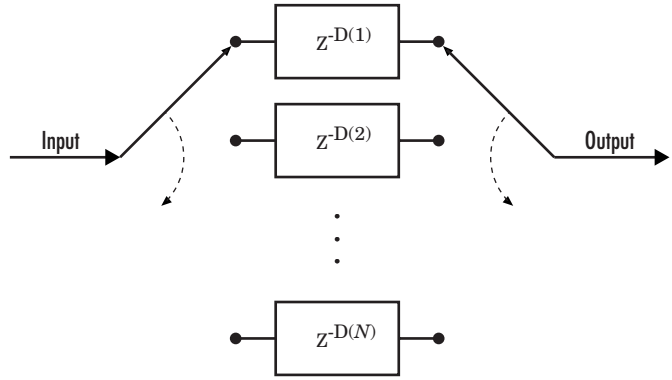
Your answer should be approximately 3/43, or .0698, since after each sequence of three 1s, the expected distance to the next sequence of 1s is 40. Consequently, you expect to see three 1s in 43 terms of the sequence. If there were no error correction in the model, the bit error rate would be approximately .0698.

When you run a simulation with the model, the error rate is approximately .019, which shows the improvement due to error correction and interleaving. You can see the effect of interleaving by deleting the Random Interleaver and Random Deinterleaver blocks from the model, connecting the lines, and running another simulation. The bit error rate is higher without interleaving because the Hamming code can only correct one error in each codeword.

Convolutional Interleavers

A convolutional interleaver consists of a set of shift registers, each with a fixed delay. In a typical convolutional interleaver, the delays are nonnegative integer multiples of a fixed integer (although a general multiplexed interleaver allows arbitrary delay values). Each new symbol from the input

signal feeds into the next shift register and the oldest symbol in that register becomes part of the output signal. The schematic below depicts the structure of a convolutional interleaver by showing the set of shift registers and their delay values D(1), D(2),..., D(N). The kth shift register holds D(k) symbols, where k = 1, 2,..., N. The blocks in this library have mask parameters that indicate the delay for each shift register. The delay is measured in samples.



This section discusses

- The types of convolutional interleavers included in the library
- The delay between the original sequence and the restored sequence
- An example that uses a convolutional interleaver

Types of Convolutional Interleavers

The set of convolutional interleavers in this library includes a general interleaver/deinterleaver pair as well as several special cases. Each special-case block uses the same computational code that its more general counterpart uses, but provides an interface that is more suitable for the special case.

The most general block in this library is the General Multiplexed Interleaver block, which allows arbitrary delay values for the set of shift registers. To implement the preceding schematic using this block, use an **Interleaver delay** parameter of [D(1); D(2); ...; D(N)].

More specific is the Convolutional Interleaver block, in which the delay value for the kth shift register is (k-1) times the block's **Register length step** parameter. The number of shift registers in this block is the value of the **Rows of shift registers** parameter.

Finally, the Helical Interleaver block supports a special case of convolutional interleaving that fills an array with symbols in a helical fashion and empties the array row by row. To configure this interleaver, use the **Number of columns of helical array** parameter to set the width of the array, and use the **Group size** and **Helical array step size** parameters to determine how symbols are placed in the array. See the reference page for the Helical Interleaver block for more details and an example.

Delays of Convolutional Interleavers

After a sequence of symbols passes through a convolutional interleaver and a corresponding convolutional deinterleaver, the restored sequence lags behind the original sequence. The delay, measured in symbols, between the original and restored sequences is

(Number of shift registers) * (Maximum delay among all shift registers)

for the most general multiplexed interleaver. If your model incurs an additional delay between the interleaver output and the deinterleaver input, the restored sequence lags behind the original sequence by the sum of the additional delay and the amount in the preceding formula.

Note For proper synchronization, the delay in your model between the interleaver output and the deinterleaver input must be an integer multiple of the number of shift registers. You can use the Delay block in the Signal Processing Blockset to adjust delays manually, if necessary.

Convolutional Interleaver block. In the special case implemented by the Convolutional Interleaver/Convolutional Deinterleaver pair, the number of shift registers is the **Rows of shift registers** parameter, while the maximum delay among all shift registers is

B * (N-1)

where B is the **Register length step** parameter and N is the **Rows of shift registers** parameter.

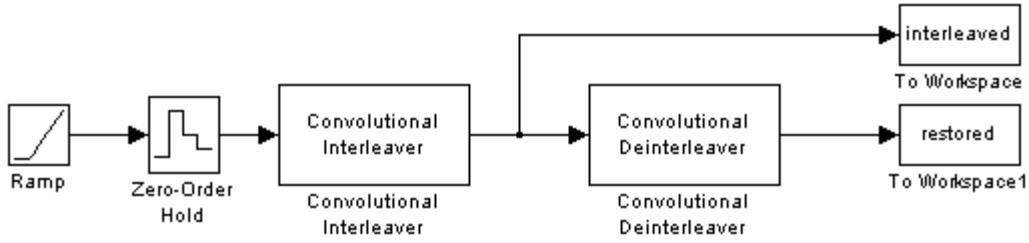
Helical Interleaver block. In the special case implemented by the Helical Interleaver/Helical Deinterleaver pair, the delay between the restored sequence and the original sequence is

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

where C is the **Number of columns in helical array** parameter, N is the **Group size** parameter, and s is the **Helical array step size** parameter.

Example: Convolutional Interleavers

The example below illustrates convolutional interleaving and deinterleaving using a sequence of consecutive integers. It also illustrates the inherent delay and the effect of the interleaving blocks' initial conditions.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Ramp, in the Simulink Sources library. Use default parameters.
- Zero-Order Hold, in the Simulink Discrete library. Use default parameters.
- Convolutional Interleaver
  - Set **Rows of shift registers** to 3.

  - Set **Initial conditions** to `[-1 -2 -3]'`.
- Convolutional Deinterleaver
  - Set **Rows of shift registers** to `3`.
  - Set **Initial conditions** to `[-1 -2 -3]'`.
- Two copies of To Workspace, in the Simulink Sinks library
  - Set **Variable name** to `interleaved` and `restored`, respectively, in the two copies of this block.
  - Set **Save format** to `Array` in each of the two copies of this block.

Connect the blocks as shown in the preceding diagram. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `20`. Run the simulation and execute the following command:

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

comparison =

     0     0    -1
     1    -2    -2
     2    -3    -3
     3     3    -1
     4    -2    -2
     5    -3    -3
     6     6    -1
     7     1    -2
     8    -3    -3
     9     9    -1
    10     4    -2
    11    -3    -3
    12    12     0
    13     7     1
    14     2     2
    15    15     3
    16    10     4
    17     5     5
    18    18     6
```

```
19     13      7
20      8      8
```

In this output, the first column contains the original symbol sequence. The second column contains the interleaved sequence, while the third column contains the restored sequence.

The negative numbers in the interleaved and restored sequences come from the interleaving blocks' initial conditions, not from the original data. The first of the original symbols appears in the restored sequence only after a delay of 12 symbols. The delay of the interleaver-deinterleaver combination is the product of the number of shift registers (3) and the maximum delay among all shift registers (4).

For a similar example that also indicates the contents of the shift registers at each step of the process, see "Example: Convolutional Interleavers" in the Communications Toolbox documentation set.

Selected Bibliography for Interleaving

[1] Berlekamp, E.R., and P. Tong, "Improved Interleavers for Algebraic Block Codes," U. S. Patent 4559625, Dec. 17, 1985.

[2] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[3] Forney, G.D., Jr., "Burst-Correcting Codes for the Classic Bursty Channel," *IEEE Transactions on Communications*, Vol. COM-19, October 1971, pp. 772–781.

[4] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

[5] Ramsey, J.L, "Realization of Optimum Interleavers," *IEEE Transactions on Information Theory*, IT-16 (3), May 1970, pp. 338–345.

[6] Takeshita, O.Y., and Costello, D.J., Jr., "New Classes of Algebraic Interleavers for Turbo-Codes," *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, August, 1998, pp. 419.

Analog Modulation

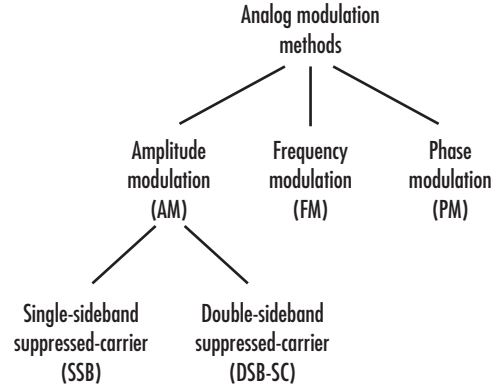
In most media for communication, only a fixed range of frequencies is available for transmission. One way to communicate a message signal whose frequency spectrum does not fall within that fixed frequency range, or one that is otherwise unsuitable for the channel, is to alter a transmittable signal according to the information in your message signal. This alteration is called *modulation*, and it is the modulated signal that you transmit. The receiver then recovers the original signal through a process called *demodulation*. This section describes how to modulate and demodulate analog signals with the Communications Blockset. The topics are as follows:

Accessing Analog Modulation Blocks

Open the Modulation library by double-clicking its icon in the main Communications Blockset library. Then open the Analog Passband sublibrary by double-clicking its icon in the Modulation library.

Analog Modulation Features of the Blockset

The following figure shows the modulation techniques that the Communications Blockset supports for analog signals. As the figure suggests, some categories of techniques include named special cases.



For a given modulation technique, two ways to simulate modulation techniques are called *baseband* and *passband*. This blockset supports passband simulation for analog modulation.

The modulation and demodulation blocks also let you control such features as the initial phase of the modulated signal and post-demodulation filtering.

Representing Signals for Analog Modulation

Analog modulation blocks in this blockset process only sample-based scalar signals. The input and output of the analog modulator and demodulator are all real signals.

All analog demodulators in this blockset produce discrete-time, not continuous-time, output.

Sampling Issues in Analog Modulation

The proper simulation of analog modulation requires that the Nyquist criterion be satisfied, taking into account the signal bandwidth.

Specifically, the sample rate of the system must be greater than twice the sum of the carrier frequency and the signal bandwidth.

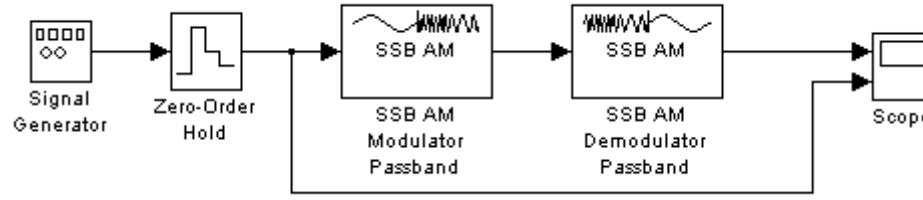
Filter Design Issues

After demodulating, you might want to filter out the carrier signal. The particular filter used, such as `butter`, `cheby1`, `cheby2`, and `ellip`, can be selected on the mask of the demodulator block. Different filtering methods have different properties, and you might need to test your application with several filters before deciding which is most suitable.

Example: Varying the Filter's Cutoff Frequency

In many situations, a suitable cutoff frequency is half the carrier frequency. Since the carrier frequency must be higher than the bandwidth of the message signal, a cutoff frequency chosen in this way properly filters out unwanted frequency components. If the cutoff frequency is too high, those components may not be filtered out. If the cutoff frequency is too low, it might narrow the bandwidth of the message signal.

The following example modulates a sawtooth message signal, demodulates the resulting signal using a Butterworth filter, and plots the original and recovered signals. The Butterworth filter is implemented within the SSB AM Demodulator Passband block.

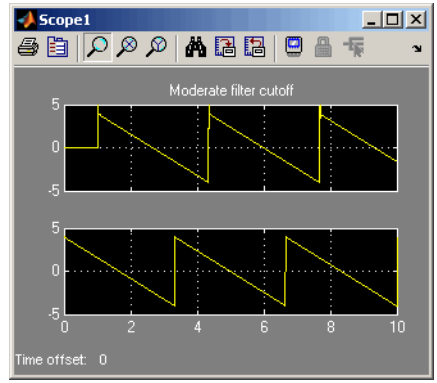


To build the model, gather and configure these blocks:

- Signal Generator, in the Simulink Sources library
  - Set **Wave form** to `Sawtooth`.
  - Set **Amplitude** to `4`.
  - Set **Frequency** to `.3`.
- Zero-Order Hold, in the Simulink Discrete library
  - Set **Sample time** to `.01`.

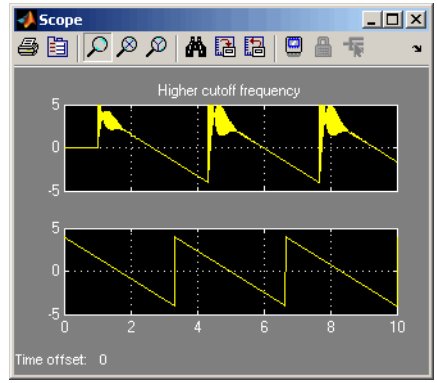
- SSB AM Modulator Passband, in the Analog Passband sublibrary of the Modulation library
  - Set **Carrier frequency** to `25`.
  - Set **Initial phase** to `0`.
  - Set **Sideband to modulate** to `Upper`.
  - Set **Hilbert transform filter order** to `200`.
- SSB AM Demodulator Passband, in the Analog Passband sublibrary of the Modulation library
  - Set **Carrier frequency** to `25`.
  - Set **Initial phase** to `0`.
  - Set **Lowpass filter design method** to `Butterworth`.
  - Set **Filter order** to `2`.
  - Set **Cutoff frequency** to `30`.
- Scope, in the Simulink Sinks library
  - After double-clicking the block to open it, click the **Parameters** icon and set **Number of axes** to `2`.

Connect the blocks as in the figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `10`. Running the model produces the following scope image. The image reflects the original and recovered signals, with a moderate filter cutoff.

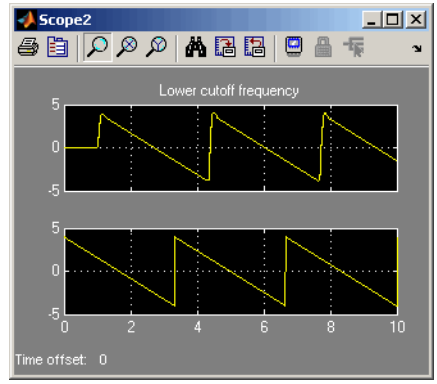


There is invariably a delay between a demodulated signal and the original received signal. Both the filter order and the filter parameters directly affect the length of this delay.

Other Filter Cutoffs. To see the effect of a lowpass filter with a *higher* cutoff frequency, set the **Cutoff frequency** of the SSB AM Demodulator Passband block to `49`, and run the simulation again. The new result is shown below. The higher cutoff frequency allows the carrier signal to interfere with the demodulated signal.



To see the effect of a lowpass filter with a *lower* cutoff frequency, set the **Cutoff frequency** of the SSB AM Demodulator Passband block to `4`, and run the simulation again. The new result is shown in the following figure. The lower cutoff frequency narrows the bandwidth of the demodulated signal.


Scope2
Lower cutoff frequency
5
0
-5
5
0
-5
0
2
4
6
8
10
Time offset: 0

Digital Modulation

Like analog modulation, digital modulation alters a transmittable signal according to the information in a message signal. However, in this case, the message signal is a discrete-time signal that can assume finitely many values. This section describes how to modulate and demodulate digital signals with the Communications Blockset. The topics in this sections are as follows:

For background material on the subject of digital modulation, see the works listed in "Selected Bibliography for Digital Modulation" on page 1-114.

Accessing Digital Modulation Blocks

Open the Modulation library by double-clicking the icon in the main Communications Blockset library. Then open the Digital Baseband sublibrary by double-clicking its icon in the Modulation library.

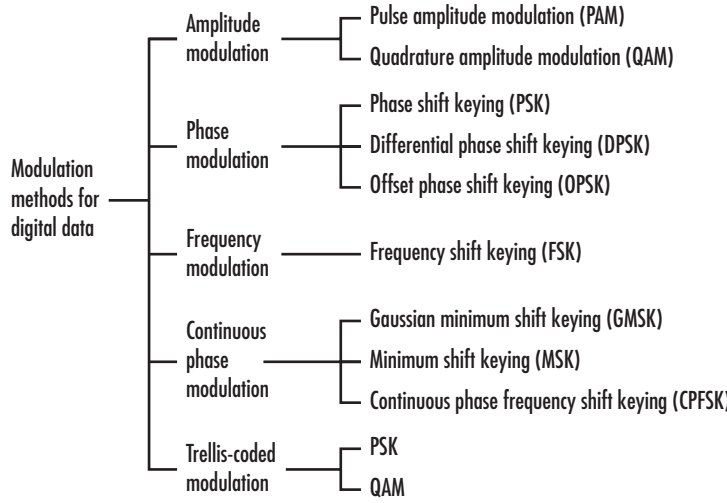
The Digital Baseband library has sublibraries of its own. Open each of these sublibraries by double-clicking the icon listed in the table below.

Kind of Modulation	Icon in Digital Baseband Library
Amplitude modulation	AM
Phase modulation	PM
Frequency modulation	FM

Kind of Modulation	Icon in Digital Baseband Library
Continuous phase modulation	CPM
Trellis-coded modulation	TCM

Digital Modulation Features of the Blockset

The figure below shows the modulation techniques that the Communications Blockset supports for digital data. All the methods at the far right are implemented in library blocks.



General and Specific Modulation Methods

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

General and Specific Blocks

General Modulator	Specific Modulator	Specific Conditions
General QAM Modulator Baseband	Rectangular QAM Modulator Baseband	Predefined constellation containing 2^K points on a rectangular lattice
M-PSK Modulator Baseband	BPSK Modulator Baseband	**M-ary number** parameter is 2.
	QPSK Modulator Baseband	**M-ary number** parameter is 4.
M-DPSK Modulator Baseband	DBPSK Modulator Baseband	**M-ary number** parameter is 2.
	DQPSK Modulator Baseband	**M-ary number** parameter is 4.
CPM Modulator Baseband	GMSK Modulator Baseband	**M-ary number** parameter is 2, **Frequency pulse shape** parameter is `Gaussian`.
	MSK Modulator Baseband	**M-ary number** parameter is 2, **Frequency pulse shape** parameter is `Rectangular`, **Pulse length** parameter is 1.
	CPFSK Modulator Baseband	**Frequency pulse shape** parameter is `Rectangular`, **Pulse length** parameter is 1.

General and Specific Blocks (Continued)

General Modulator	Specific Modulator	Specific Conditions
General TCM Encoder	Rectangular QAM TCM Encoder	Predefined signal constellation containing 2^K points on a rectangular lattice
	M-PSK TCM Encoder	Predefined signal constellation containing 2^K points on a circle

Furthermore, the CPFSK Modulator Baseband block is similar to the M-FSK Modulator Baseband block, when the M-FSK block uses continuous phase transitions. However, the M-FSK features of this blockset differ from the CPFSK features in their mask interfaces and in the demodulator implementations.

Baseband Modulated Signals

For a given modulation technique, two ways to simulate modulation techniques are called *baseband* and *passband*. This blockset supports baseband simulation for digital modulation. Baseband simulation, also known as the *lowpass equivalent method*, requires less computation compared to passband simulation. This is because modeling a high-frequency carrier signal is computationally intensive.

For the mathematical expressions that define baseband signals, see the baseband signal section in the Communications Toolbox documentation.

Representing Signals for Digital Modulation

All digital modulation blocks process only discrete-time signals and use the baseband representation. The data types of inputs and outputs are depicted in the following figure.

Note If you want to separate the in-phase and quadrature components of the complex modulated signal, use the Complex to Real-Imag block in the Simulink Math Operations library.

Binary-Valued and Integer-Valued Signals

Some digital modulation blocks can accept either integers or binary representations of such integers. The corresponding demodulation blocks can output either integers or their binary representations. This section describes how modulation blocks process binary inputs; the case for demodulation blocks is the reverse.

If a modulator block's **Input type** parameter is set to `Bit`, the block accepts binary representations of integers between 0 and M-1. It modulates each group of K bits, called a binary *word*. Also, the input vector length must be an integer multiple of K. If the input is frame-based, then it must be a column vector.

In binary input mode, the **Constellation ordering** (or **Symbol set ordering**, depending on the type of modulation) parameter indicates how the block maps a group of K input bits to a corresponding integer. If this parameter is set to `Binary`, the block maps [$u(1)$ $u(2)$ ... $u(\mathrm{K})$] to the integer

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

and subsequently behaves as if this integer were the input value. $u(1)$ is the most significant bit.

For example, if M = 8, **Constellation ordering** (or **Symbol set ordering**) is set to `Binary`, and the binary input word is [1 1 0], the block internally

converts [1 1 0] to the integer 6. The block produces the same output as in the case when the input is 6 and the **Input type** parameter is `Integer`.

If **Constellation ordering** (or **Symbol set ordering**) is set to `Gray`, the block uses a Gray-coded arrangement. The explicit mapping is described in the algorithm section on the reference page for the M-PSK Modulator Baseband block.

Delays in Digital Modulation

Digital modulation and demodulation blocks sometimes incur delays between their inputs and outputs, depending on their configuration and on properties of their signals. The following table lists sources of delay and the situations in which they occur.

Delays Resulting from Digital Modulation or Demodulation

Modulation or Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
All demodulators in AM, PM, and FM sublibraries except OQPSK	Sample-based input	One output period
All demodulators in CPM sublibrary	Sample-based input, D = **Traceback length** parameter	D+1 output periods
	Frame-based input, D = **Traceback length** parameter	D output periods

Delays Resulting from Digital Modulation or Demodulation (Continued)

Modulation or Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
OQPSK modulator-demodulator *pair*	Frame-based input	One output period
	Sample-based input	Two output periods
	Sample-based input, and the model uses a fixed-step solver with **Mode** parameter set to `Auto` or `MultiTasking`.	Two output periods
	Sample-based input, and the model uses a variable-step solver or the **Mode** parameter is not set to `Auto` or `MultiTasking`.	One output period
All demodulators in TCM sublibrary	**Operation mode** set to `Continuous`, Tr = **Traceback depth** parameter, and code rate k/n	Tr*k output bits

As a result of delays, data that enters a modulation or demodulation block at time T appears in the output at time T+delay. In particular, if your simulation computes error statistics or compares transmitted with received data, it must take the delay into account when performing such computations or comparisons.

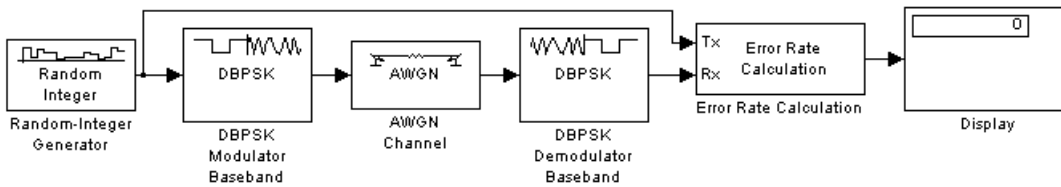
First Output Sample in DPSK Demodulation

In addition to the delays mentioned above, the DPSK, DQPSK, and DBPSK demodulators produce output whose first sample is unrelated to the input. This is related to the differential modulation technique, not the particular implementation of it.

Example: Delays from Demodulation

Demodulation in the model below causes the demodulated signal to lag, compared to the unmodulated signal. This delay is typical for sample-based data that the modulator upsamples. When computing error statistics, the model accounts for the delay by setting the Error Rate Calculation block's **Receive delay** parameter to 1. If the **Receive delay** parameter had a different value, then the error rate showing at the top of the Display block would be close to 1/2.

Note If this model used the OQPSK method instead of DBPSK, the proper **Receive delay** parameter would be 2 instead of 1.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to 2.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
- DBPSK Modulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Phase rotation** to 0.
- AWGN Channel, in the Channels library
  - Set **Es/No** to 4.
- DBPSK Demodulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation

  - Set **Phase rotation** to `0`.
- Error Rate Calculation, in the Comm Sinks library
  - Set **Receive delay** to `0`.
  - Set **Computation delay** to `0`.
  - Set **Output data** to `Port`.
- Display, in the Simulink Sinks library
  - Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as shown above. From the model window's **Simulation**, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `100`. Then run the model and observe the error rate at the top of the Display block's icon. Your error rate will vary depending on your **Initial seed** value in the Random Integer Generator block.

Upsampled Signals and Rate Changes

Some digital modulation blocks can output an upsampled version of the modulated signal, while their corresponding digital demodulation blocks can accept an upsampled version of the modulated signal as input. Each block's **Samples per symbol** parameter, S, is the upsampling factor in both cases. It must be a positive integer. Depending on whether the signal is frame-based or sample-based, the block either changes the signal's vector size or its sample time, as the table below indicates. Only the OQPSK blocks deviate from the information in the table, in that S is replaced by 2S in the scaling factors.

Processing of Upsampled Modulated Data (Except OQPSK Method)

Computation Type	Input Frame Status	Result
Modulation	Frame-based	Output vector length is S times the number of integers or binary words in the input vector. Output sample time equals the input sample time.

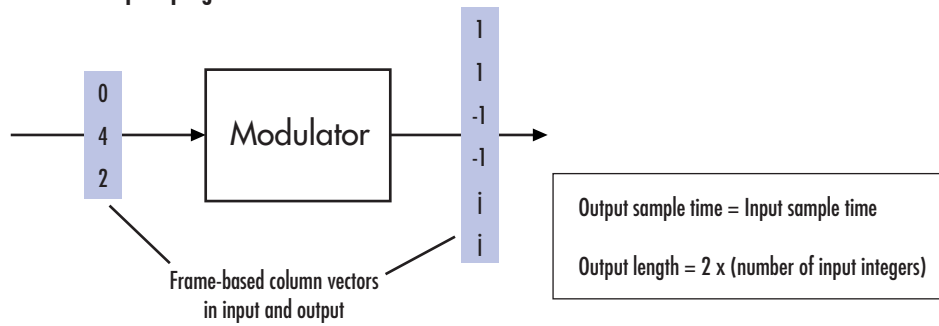
Processing of Upsampled Modulated Data (Except OQPSK Method) (Continued)

Computation Type	Input Frame Status	Result
Modulation	Sample-based	Output vector is a scalar. Output sample time is 1/S times the input sample time.
Demodulation	Frame-based	Number of integers or binary words in the output vector is 1/S times the number of samples in the input vector. Output sample time equals the input sample time.
Demodulation	Sample-based	Output signal contains one integer or one binary word. Output sample time is S times the input sample time. Furthermore, if S > 1 and the demodulator is from the AM, PM, or FM sublibrary, the demodulated signal is delayed by one output sample period. There is no delay if S = 1 or if the demodulator is from the CPM sublibrary.

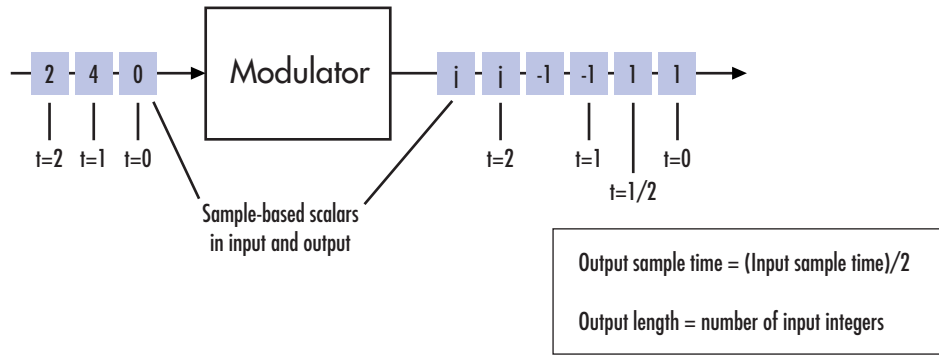
Illustrations of Size or Rate Changes

The following schematics illustrate how a modulator (other than OQPSK) upsamples a triplet of frame-based and sample-based integers. In both cases, the **Samples per symbol** parameter is 2.

Frame-Based Upsampling

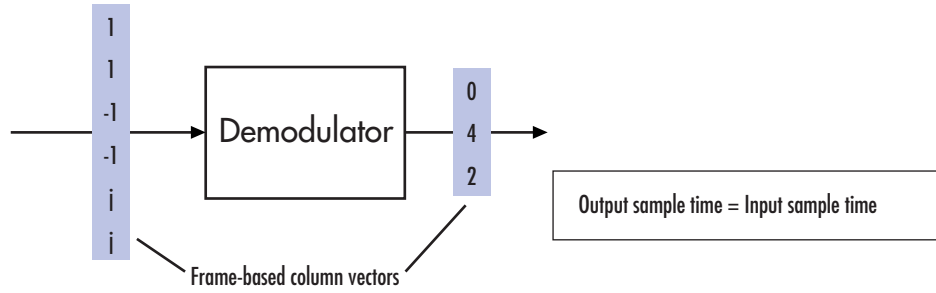


Sample-Based Upsampling

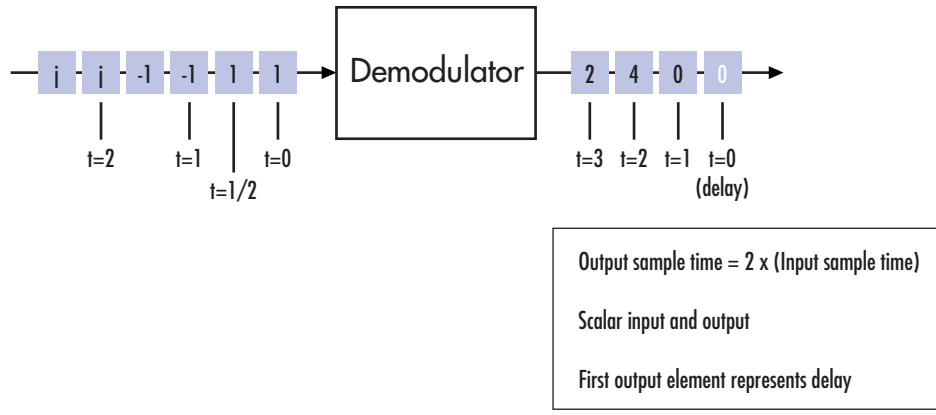


The following schematics illustrate how a demodulator (other than OQPSK or one from the CPM sublibrary) processes three doubly sampled symbols using both frame-based and sample-based inputs. In both cases, the **Samples per symbol** parameter is `2`. The sample-based schematic includes an output delay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

Frame-Based Upsampled Input



Sample-Based Upsampled Input

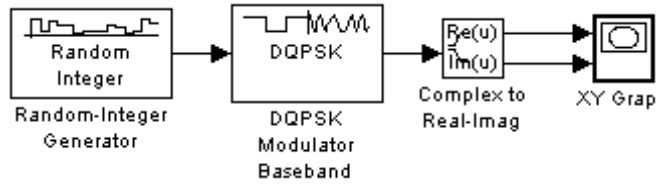


Examples of Digital Modulation

This section builds a few simple example models to illustrate the modulation methods and how the Communications Blockset allows you to implement them. The examples are

DQPSK Signal Constellation Points and Transitions

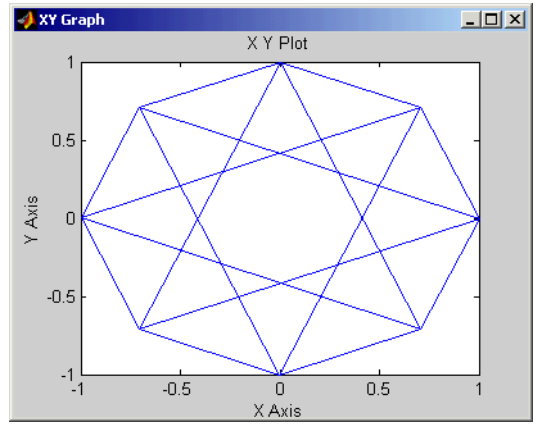
The model below plots the output of the DQPSK Modulator Baseband block. The image shows the possible transitions from each symbol in the DQPSK signal constellation to the next symbol.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `4`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
  - Set **Sample time** to `.01`.
- DQPSK Modulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- Complex to Real-Imag, in the Simulink Math Operations library
- XY Graph, in the Simulink Sinks library

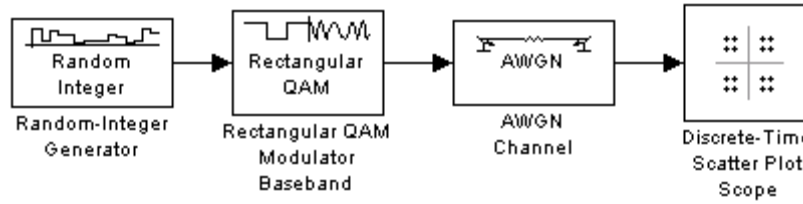
Use the blocks' default parameters unless otherwise instructed. Connect the blocks as in the figure above. Running the model produces the following plot. The plot reflects the transitions among the eight DQPSK constellation points.



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

Rectangular QAM Modulation and Scatter Diagram

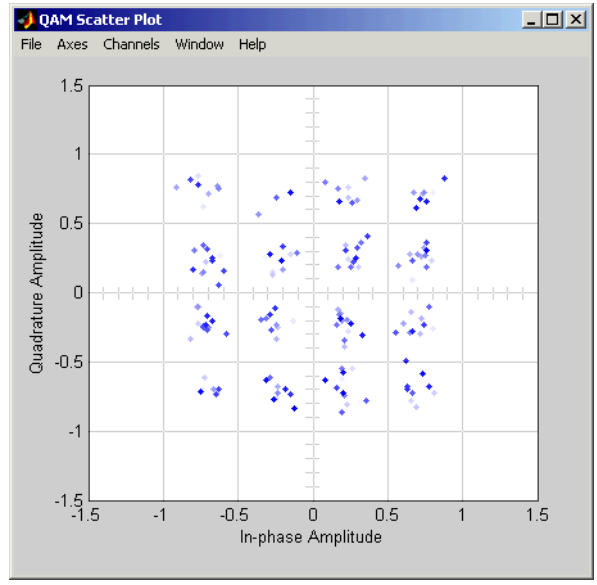
The model below uses the M-QAM Modulator Baseband block to modulate random data. After passing the symbols through a noisy channel, the model produces a scatter diagram of the noisy data. The diagram suggests what the underlying signal constellation looks like and shows that the noise distorts the modulated signal from the constellation.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `16`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randseed` function.
  - Set **Sample time** to `.1`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Peak Power`.
- AWGN Channel, in the Channels library
  - Set **Es/No** to `20`.
  - Set **Symbol period** to `.1`.
- Discrete-Time Scatter Plot Scope, in the Comm Sinks library
  - Set **Points displayed** to `160`.
  - Set **New points per display** to `80`.
  - On the **Figure Properties** panel, set **Scope position** to `figposition([2.5 55 35 35]);`.
  - On the same panel, set **Figure name** to `QAM Scatter Plot`.

Connect the blocks as in the figure. From the model window's **Simulation** menu, select **Configuration parameters**. In the Configuration Parameters dialog box, set **Stop time** to `250`. Running the model produces a scatter diagram like the following one. Your plot might look somewhat different, depending on your **Initial seed** value in the Random Integer Generator block. Because the modulation technique is 16-QAM, the plot shows 16 clusters of points. If there were no noise, the plot would show the 16 exact constellation points instead of clusters around the constellation points.

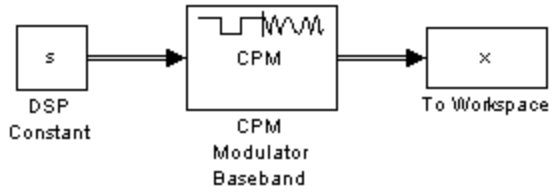


Phase Tree for Continuous Phase Modulation

This example plots a phase tree associated with a continuous phase modulation scheme. A phase tree is a diagram that superimposes many curves, each of which plots the phase of a modulated signal over time. The distinct curves result from different inputs to the modulator.

This example uses the CPM Modulator Baseband block for its numerical computations. The block is configured so that it uses a raised cosine filter pulse shape. The example also illustrates how you can use Simulink and MATLAB together. The example uses MATLAB commands to run a series of simulations with different input signals, to collect the simulation results, and to plot the full data set.

Note In contrast to this example's approach using both MATLAB and Simulink, the `cpmphasetree` demo produces a phase tree using a Simulink model without additional lines of MATLAB code.



The first step of this example is to build the model. To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- DSP Constant, in the Signal Processing Sources library
  - Set **Constant value** to `s` (which will appear in the MATLAB workspace).
  - Set **Output** to `Frame-based`.
  - Set **Frame period** to `1`.
- CPM Modulator Baseband
  - Set **M-ary number** to `2`.
  - Set **Modulation index** to `2/3`.
  - Set **Frequency pulse shape** to `Raised Cosine`.
  - Set **Pulse length** to `2`.
- To Workspace, in the Simulink Sinks library
  - Set **Variable name** to `x`.
  - Set **Save format** to `Array`.

Do not run the model, because the variable `s` is not yet defined in the MATLAB workspace. Instead, save the model to a directory on your MATLAB path, using the filename `doc_phasetree`.

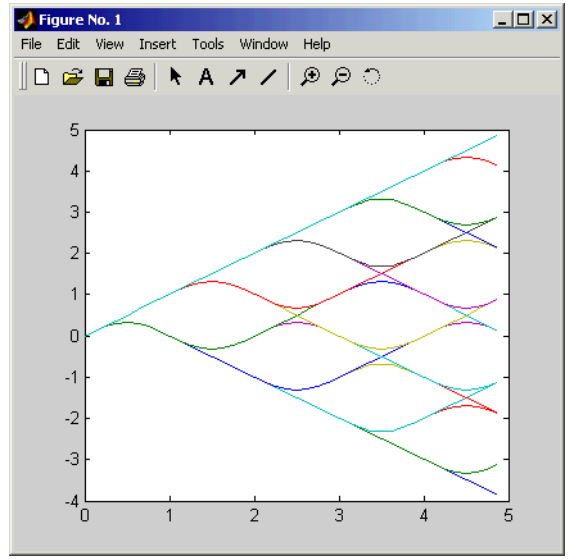
The second step of this example is to execute these commands in MATLAB:

```
% Parameters from the CPM Modulator Baseband block
M_ary_number = 2;
modulation_index = 2/3;
pulse_length = 2;
```

```
samples_per_symbol = 8;
opts = simset('SrcWorkspace','Current',...
   'DstWorkspace','Current');

L = 5;  % Symbols to display
pmat = [];
for ip_sig = 0:(M_ary_number^L)-1
    s = de2bi(ip_sig,L,M_ary_number,'left-msb');
    % Apply the mapping of the input symbol to the CPM
    % symbol 0 -> -(M-1), 1 -> -(M-2), etc.
    s = 2*s'+1-M_ary_number;
    sim('doc_phasetree', .9, opts); % Run model to generate x.
    pmat(:,ip_sig+1) = unwrap(angle(x(:))); % Next column of pmat
end;
pmat = pmat/(pi*modulation_index);
t = (0:L*samples_per_symbol-1)'/samples_per_symbol;
plot(t,pmat); figure(gcf); % Plot phase tree.
```

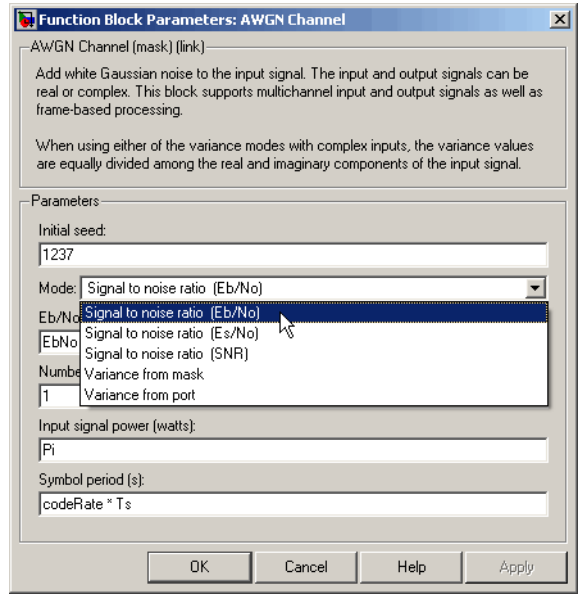
The resulting plot follows. Each curve represents a different instance of simulating the CPM Modulator Baseband block with a distinct (constant) input signal.



Setting Noise Variance for Computing LLRs

Some digital demodulation blocks can compute bitwise log-likelihood ratio (LLR) and approximate LLR values that can be used by some decoders (such as the Viterbi Decoder block) to achieve better BER performance. To compute LLR or approximate LLR, these blocks need, along with other parameters, the variance of the noise in the input signal.

If you are adding noise to the transmitted signal using the AWGN Channel block, there are various methods for computing the noise variance at the input of a demodulator block based on the **Mode** parameter of the AWGN Channel block, as shown in the following figure.



Eb/No Mode

The example model `doc_noisevariance_ebno` shows the case where the AWGN Channel block's **Mode** parameter is set to `Signal to noise ratio (Eb/No)`.

To open the model, change your directory to `<MATLAB>/help/toolbox/commblks/commblks_examples` and

type `doc_noisevariance_ebno` at the MATLAB command line, or click here, if you are viewing this in the MATLAB help browser.

The BPSK Demodulator Baseband block's **Decision type** is set to `Log-likelihood ratio`, and the **Noise variance source** is set to `Dialog`. The `Dialog` setting enables a field, **Noise variance**, which is then set to `Pi/(10^(EbNo/10))`, where `Pi` is the input signal power, in watts, and `EbNo` is the ratio of bit energy to noise power spectral density, in decibels.

The AWGN Channel block requires both `Eb/No` and `Input signal power` parameters to define the noise that it adds to the signal. Variables such as `Pi` and `EbNo` are defined by the example model in the base workspace.

Es/No Mode

The example `doc_noisevariance_esno` (also found in `<MATLAB>/help/toolbox/commblks/commblks_examples`) shows how to calculate noise variance for the MPSK Demodulator Baseband block, computing approximate LLR when the AWGN Channel block's **Mode** parameter is set to `Signal to noise ratio (Es/No)`.

The method of computing noise variance remains the same, whether the computation is using LLR or approximate LLR. The noise variance is given as `Pi/(10^(EsNo/10))`, where `Pi` is the input signal power, in watts, and `EsNo` is the ratio of signal energy to noise power spectral density, in decibels.

The AWGN Channel block uses the two parameters, `Es/No` and `Input signal power`, along with others, to compute the noise it adds to the signal.

SNR Mode

`doc_noisevariance_snr` (found in the same directory) demonstrates the use of the `Signal to noise ratio (SNR)` mode of the AWGN Channel block.

In this example, the Rectangular QAM Demodulator Baseband block, which computes the LLR, is given the noise variance of its input signal as `Pi/(10^(SNR/10))`, where `Pi` is the input signal power, in watts, and `SNR` is the ratio of signal power to noise power, in decibels.

The noise variance is computed using the parameters needed by the AWGN Channel block to define the noise added to the signal.

Selected Bibliography for Digital Modulation

[1] Anderson, J. B., T. Aulin, and C.-E. Sundberg, *Digital Phase Modulation*, New York, Plenum Press, 1986.

[2] Biglieri, E., D. Divsalar, P.J. McLane, and M.K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[3] Jeruchim, M. C., P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[4] Pawula, R.F., "On M-ary DPSK Transmission Over Terrestrial and Satellite Channels," *IEEE Transactions on Communications*, Vol. COM-32, July 1984, pp. 752–761.

[5] Smith, J. G., "Odd-Bit Quadrature Amplitude-Shift Keying," *IEEE Transactions on Communications*, Vol. COM-23, March 1975, pp. 385–389.

Communications Filters

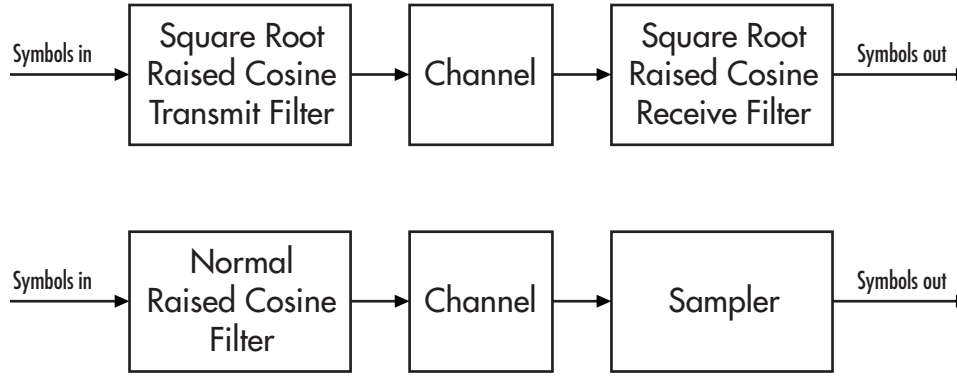
The Comm Filters library includes several blocks that you can use for filtering or pulse shaping (that is, either transmit filtering or receive filtering). These operations are necessary to control bandwidth, intersymbol interference, and signal-to-noise ratio.

The topics in this section are as follows:

Filter Features of the Blockset

Filtering tasks supported in the Communications Blockset include

- Filtering using a raised cosine filter. Raised cosine filters are very commonly used for pulse shaping and matched filtering. The schematic below illustrates two typical uses of raised cosine filters.



- Filtering using a Gaussian filter.
- Shaping a signal using ideal rectangular pulses.

- Implementing an integrate-and-dump operation or a windowed integrator. An integrate-and-dump operation is often used in a receiver model when the system's transmitter uses an ideal rectangular-pulse model. Integrate-and-dump can also be used in fiber optics and in spread-spectrum communication systems such as CDMA (code division multiple access) applications.

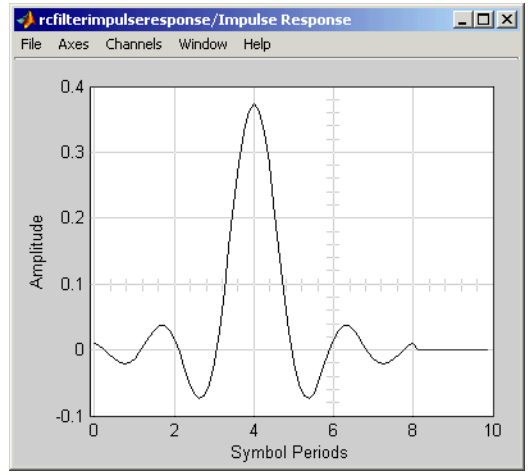
Other filtering capabilities are in the Signal Processing Blockset, in the Filter Designs and Multirate Filters libraries.

For more background information about filters and pulse shaping, see the works listed in "Selected Bibliography for Communications Filters" on page 1-121.

Group Delay of a Filter

The raised cosine and Gaussian filter blocks in this library implement realizable filters by delaying the peak response. This delay, known as the filter's *group delay*, is the length of time between the filter's initial response and its peak response. The filter blocks in this library have a **Group delay** parameter that is an integer representing the number of symbol periods.

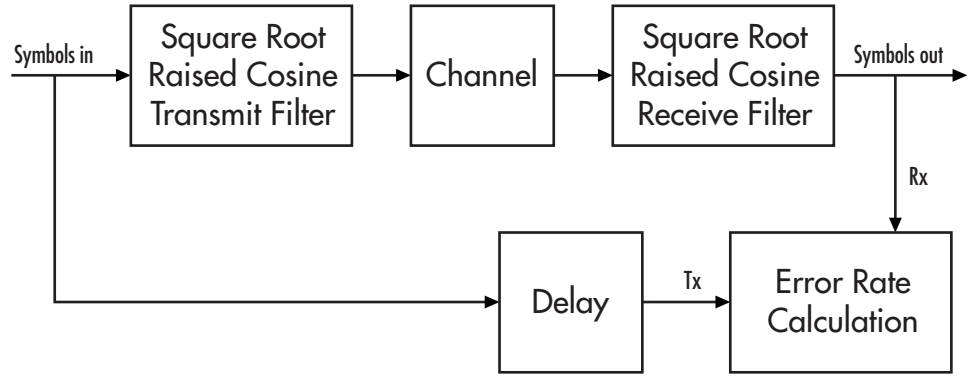
For example, the square root raised cosine filter whose impulse response shown in the following figure uses a **Group delay** parameter of `4` in the filter block. In the figure, the initial impulse response is small and the peak impulse response occurs at the fourth symbol.



Implications of Delay for Simulations

A filter block's **Group delay** parameter value has implications for other parts of your model. For example, suppose you compare the symbol streams marked Symbols In and Symbols Out in the schematics in "Filter Features of the Blockset" on page 1-115 by plotting or computing an error rate. Use one of these methods to make sure you are comparing symbols that truly correspond to each other:

- Use the Delay block in the Signal Processing Blockset to delay the Symbols In signal, thus aligning it with the Symbols Out signal. Set the **Delay** parameter equal to the filter's **Group delay** parameter (or the sum of both values, if your model uses a pair of square root raised cosine filter blocks). This usage is illustrated in the following figure for the case of a pair of square root raised cosine filters.



- Use the Align Signals block to align the two signals.
- When using the Error Rate Calculation block to compare the two signals, increase the **Receive delay** parameter by the **Group delay** parameter value (or the sum of both values, if your model uses a pair of square root raised cosine filter blocks). The **Receive delay** parameter might include other delays as well, depending on the contents of your model.

For more information about how to manage delays in a model, see "Computing Delays" on page 2-2 and "Manipulating Delays" on page 2-14.

Filtering with Raised Cosine Filter Blocks

The Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks are designed for raised cosine filtering. Each block can apply a square root raised cosine filter or a normal raised cosine filter to a signal. You can vary the rolloff factor and group delay of the filter.

The Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks are tailored for use at the transmitter and receiver, respectively. In particular, the transmit filter outputs an upsampled signal, while the receive filter expects its input signal to be upsampled already. Also, the receive filter lets you choose whether to have the block downsample the filtered signal before sending it to the output port.

Both raised cosine filter blocks incur a propagation delay, described in "Group Delay of a Filter" on page 1-116.

Combining Two Square-Root Raised Cosine Filters

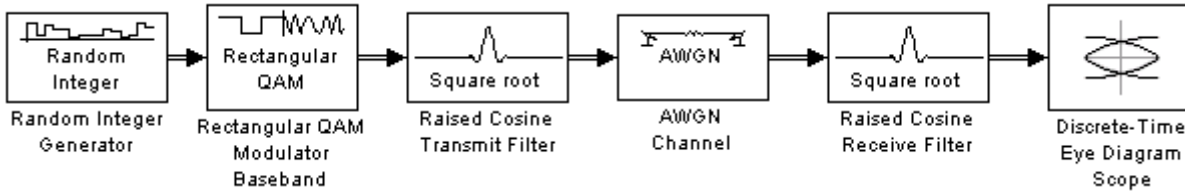
To split the filtering equally between the transmitter's filter and the receiver's filter, use a pair of square root raised cosine filters:

- Use a Raised Cosine Transmit Filter block at the transmitter, setting the **Filter type** parameter to `Square root`.
- Use a Raised Cosine Receive Filter block at the receiver, setting the **Filter type** parameter to `Square root`. In most cases, it is appropriate to set the **Input samples per symbol** parameter to match the transmit filter's **Upsampling factor** parameter.

In theory, the cascade of two square root raised cosine filters is equivalent to a single normal raised cosine filter. However, the limited impulse response of practical square root raised cosine filters causes a slight difference between the response of two cascaded square root raised cosine filters and the response of one raised cosine filter.

Example: Using Raised Cosine Filters

This example illustrates a typical setup in which a transmitter uses a square root raised cosine filter to perform pulse shaping and the corresponding receiver uses a square root raised cosine filter as a matched filter. The example plots an eye diagram from the filtered received signal.

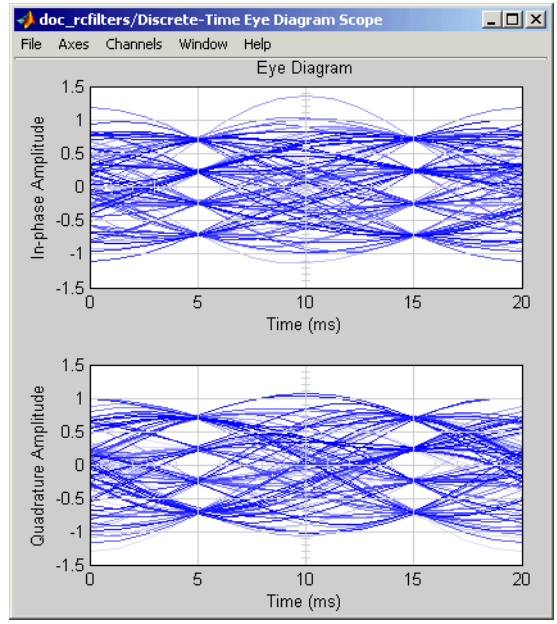


To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to 16.
  - Set **Sample time** to 1/100.

  - Select **Frame-based outputs**.
  - Set **Samples per frame** to `100`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Peak Power`.
  - Set **Peak power** to `1`.
- Raised Cosine Transmit Filter, in the Comm Filters library
  - Set **Group delay** to `4`.
- AWGN Channel, in the Channels library
  - Set **Mode** to `Signal to noise ratio (SNR)`.
  - Set **SNR** to `40`.
- Raised Cosine Receive Filter, in the Comm Filters library
  - Set **Group delay** to `4`.
  - Set **Rolloff factor** to `0.5`.
  - Set **Output mode** to `None`.
- Discrete-Time Eye Diagram Scope, in the Comm Sinks library
  - Set **Symbols per trace** to `2`.
  - Set **Traces displayed** to `100`.

Connect the blocks as in the figure. Running the simulation produces the following eye diagram. The eye diagram has two widely opened “eyes” that indicate appropriate instants at which to sample the filtered signal before demodulating. This illustrates the absence of intersymbol interference at the sampling instants of the received waveform.



The large signal-to-noise ratio in this example produces a low-noise eye diagram, while the model still illustrates where the raised cosine filter blocks typically belong in relation to a channel block. If you decrease the **SNR** parameter in the AWGN Channel block, the eyes in the diagram are less open.

Selected Bibliography for Communications Filters

[1] Proakis, John G., *Digital Communications*, 3rd ed., New York, McGraw-Hill, 1995.

[2] Rappaport, Theodore S., *Wireless Communications: Principles and Practice*, Upper Saddle River, NJ, Prentice Hall, 1996.

[3] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice Hall, 1988.

Channels

Communication channels introduce noise, fading, interference, and other distortions into the signals that they transmit. Simulating a communication system involves modeling a channel based on mathematical descriptions of the channel. Different transmission media have different properties and are modeled differently. In a simulation, the channel model usually fits directly between the transmitter and receiver, as shown below.

Channel Features of the Blockset

This blockset provides several channel models for binary, real, and complex signals. Open the Channels library by double-clicking its icon in the main Communications Blockset library.

This section describes the capabilities of the Channels library's blocks, by considering these channels:

- Additive white Gaussian noise (AWGN) channel
- Rayleigh and Rician fading channels that model real-world mobile communication effects
- Binary symmetric channel (BSC)

AWGN Channel

An AWGN channel adds white Gaussian noise to the signal that passes through it. Gaussian noise is discussed on the reference page for the Gaussian Noise Generator block. The AWGN Channel block can process either sample-based or frame-based data, and it lets you specify the variance of the noise in one of four ways:

- Directly as a mask parameter
- Directly as an input signal
- Indirectly via a signal-to-noise ratio parameter

- Indirectly via an E_s/N_o parameter

Fading Channels

The Channels library includes Rayleigh and Rician fading blocks that can simulate real-world phenomena in mobile communications. These phenomena include multipath scattering effects, as well as Doppler shifts that arise from relative motion between the transmitter and receiver. This section discusses

- How to use other blocks in the model to compensate for fading
- How to choose and configure a fading channel block
- How to plot Rayleigh channel characteristics using the channel visualization tool

For more information about fading channels in general, see "Overview of Fading Channels" in the Communications Toolbox documentation.

Note To model a channel that involves both fading and additive white Gaussian noise, use a fading channel block connected in series with the AWGN Channel block, where the fading channel block comes first.

Compensating for Fading

A communication system involving a fading channel usually requires component(s) that compensate for the fading. Here are some typical approaches:

- Differential modulation or a one-tap equalizer can help compensate for a frequency-flat fading channel.
- An equalizer with multiple taps can help compensate for a frequency-selective fading channel.

See "Equalizers" on page 1-160 to learn how to implement equalizers in this blockset. See the reference page for the M-DPSK Modulator Baseband block to learn how to implement differential modulation.

For an example that can help you visualize why compensating for a fading channel is necessary, see the plots in the `equalizer_eml` demo.

Choosing and Configuring a Fading Channel Block

The table below indicates the situations in which each fading channel block is appropriate.

Signal Path	Channel Block
Direct line-of-sight path from transmitter to receiver	Multipath Rician Fading Channel
One or more major reflected paths from transmitter to receiver	Multipath Rayleigh Fading Channel

In the case of multiple major reflected paths, a single instance of the Multipath Rayleigh Fading Channel block can model all of them simultaneously. The number of paths that the block uses is the length of either the **Delay vector** or the **Gain vector** parameter, whichever length is larger. (If both of these parameters are vectors, they must have the same length; if exactly one of these parameters is a scalar, the block expands it into a vector whose size matches that of the other **vector** parameter.)

Choosing appropriate block parameters for your situation is important. For more details about the parameters of fading channel blocks, see

- The reference pages for the Multipath Rayleigh Fading Channel block and the Multipath Rician Fading Channel block
- The “Choosing Realistic Channel Property Values” section under “Configuring Channel Objects” in the Communications Toolbox documentation
- The works listed in “Selected Bibliography for Channels” on page 1-126

Visualizing a Multipath Rayleigh Fading Channel

A multipath Rayleigh fading channel’s characteristics can be plotted using the channel visualization tool. There are two ways to do this for a model that contains a Multipath Rayleigh Fading Channel block.

One method is to double-click the block during a simulation. The second method is to select the **Open channel visualization at start of simulation** check box in the block dialog box. In subsequent simulations, the channel visualization tool appears. The tool is also used if it was left open from a previous simulation.

For the Communications Blockset, this channel visualization feature is currently available only for the Multipath Rayleigh Fading Channel block.

For details, see "Using the Channel Visualization Tool" in the Communications Toolbox User's Guide.

Examples Using Fading Channels

The following demonstration models provide examples of the use of fading channels:

- Rayleigh Fading Channel, which illustrates the channel's effect on a QPSK modulated signal
- Adaptive Equalization Using Embedded MATLAB
- IEEE 802.11a WLAN Physical Layer
- cdma2000 Physical Layer
- Defense Communications: US MIL-STD-188-110B
- WCDMA End-to-End Physical Layer

Binary Symmetric Channel

Binary error channels process binary signals by adding noise modulo 2. This library contains the Binary Symmetric Channel block, which either preserves or perturbs each vector element independently. It requires a probability that applies independently to each noise element. An example using the Binary Symmetric Channel block is in the section "Example: A Rate 2/3 Feedforward Encoder" on page 1-62.

Selected Bibliography for Channels

[1] Fechtel, Stefan A., "A Novel Approach to Modeling and Efficient Simulation of Frequency-Selective Fading Radio Channels," *IEEE Journal on Selected Areas in Communications*, Vol. 11, April 1993, pp. 422–431.

[2] Jakes, William C., ed., *Microwave Mobile Communications*, New York, IEEE Press, 1974.

[3] Lee, William C. Y., *Mobile Communications Design Fundamentals*, 2nd ed., New York, John Wiley & Sons, 1993.

[4] Proakis, John G., *Digital Communications*, 3rd ed., New York, McGraw-Hill, 1995.

RF Impairments

The RF Impairments library contains blocks that model impairments to a baseband signal caused by the radio frequency (RF) components in the receiver. This section describes the blocks in the library, covering the following topics:

Types of RF Impairments that the Blocks Model

The blocks in the RF Impairments library can simulate the following types of signal impairments:

- Nonlinearity and I/Q imbalances
- Phase/frequency offsets and phase noise
- Receiver thermal noise and free space path loss

Nonlinearity and I/Q Imbalance

The following two blocks model signal impairments due to nonlinear devices or imbalances between the in-phase and quadrature components of a modulated signal:

- The Memoryless Nonlinearity block models the AM-to-AM and AM-to-PM distortion in nonlinear amplifiers.
- The I/Q Imbalance block models imbalances between the in-phase and quadrature components of a signal caused by differences in the physical channels carrying the separate components.

These blocks distort both the phase and amplitude of the signal.

Phase/Frequency Offsets and Phase Noise

The RF Impairments library contains two blocks that simulate phase/frequency offsets and phase noise:

- The Phase/Frequency Offset block applies phase and frequency offsets to a signal.
- The Phase Noise block applies phase noise to a signal.

The Phase/Frequency Offset block and the Phase Noise block alter only the phase and frequency of the signal.

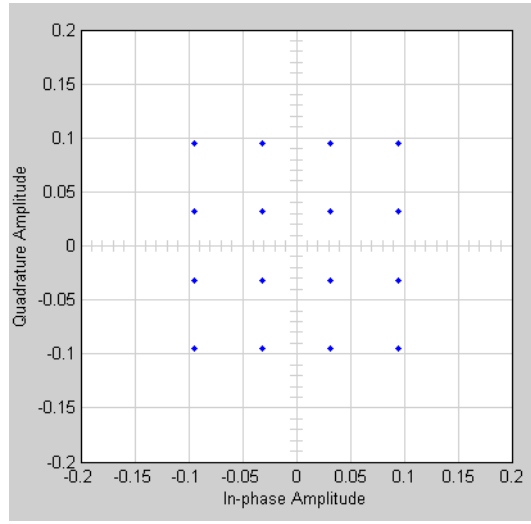
Receiver Thermal Noise and Free Space Path Loss

The RF Impairments Library contains two blocks that simulate signal impairments due to thermal noise and signal attenuation due to the distance from the transmitter to the receiver:

- The Receiver Thermal Noise block simulates the effects of thermal noise on a complex baseband signal.
- The Free Space Path Loss block simulates the loss of signal power due to the distance from the transmitter and signal frequency.

Scatter Plot Examples

This section presents scatter plots that illustrate how the blocks in the RF Impairments library distort a signal modulated by 16-ary quadrature amplitude modulation (QAM). The usual 16-ary QAM constellation without distortion is shown in the following figure.

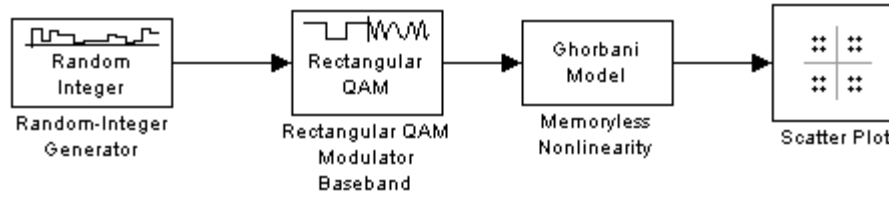


The scatter plots illustrate the effects of the following blocks:

- "Memoryless Nonlinearity Block" on page 1-130
- "I/Q Imbalance Block" on page 1-131
- "Phase/Frequency Offset Block" on page 1-132
- "Phase Noise Block" on page 1-133

As the scatter plots show, the first two blocks distort both the magnitude and angle of points in the constellation, while the last two alter just the angle.

You can create these scatter plots with models similar to the following, which produces the scatter plot for the Memoryless Nonlinearity block:



16-ary QAM Model

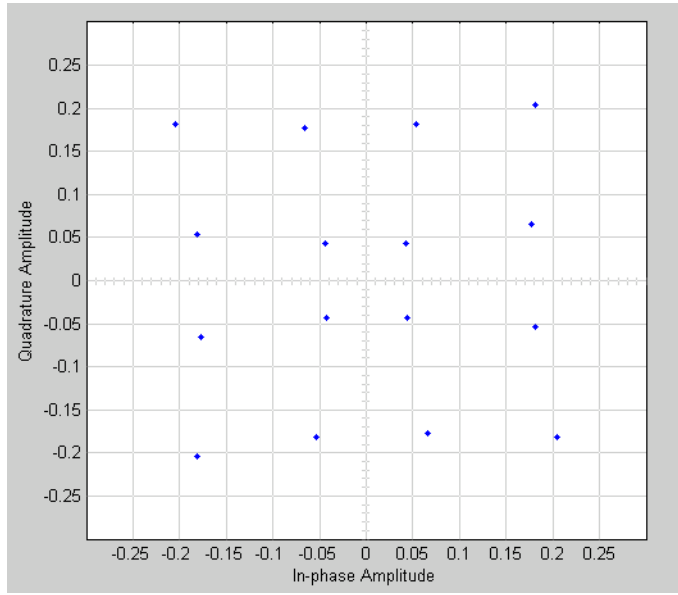
The model uses the Rectangular QAM Modulator Baseband block, from AM in the Digital Baseband Modulation sublibrary of the Modulation library. You control the power of the block's output signal with the **Normalization method** parameter.

Memoryless Nonlinearity Block

The Memoryless Nonlinearity block applies a nonlinear distortion to the input signal. This distortion models the AM-to-AM and AM-to-PM conversions in nonlinear amplifiers. The block provides several methods, which you specify by the **Method** parameter, for modeling the nonlinear characteristics of amplifiers:

- Cubic polynomial
- Hyperbolic tangent
- Saleh model
- Ghorbani model
- Rapp model

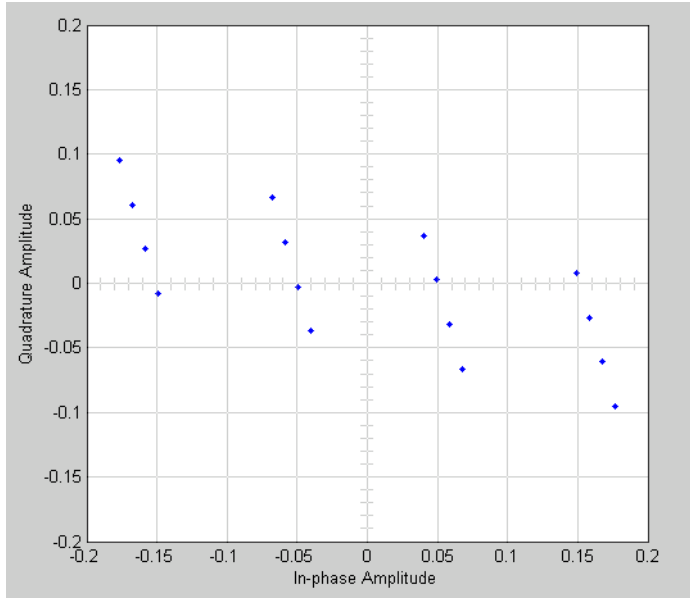
In the model shown in the preceding figure, the **Method** parameter is set to `Ghorbani model`. The following figure shows the scatter plot the model generates.



For another example of a scatter plot produced using this block, see the reference page for the Memoryless Nonlinearity block.

I/Q Imbalance Block

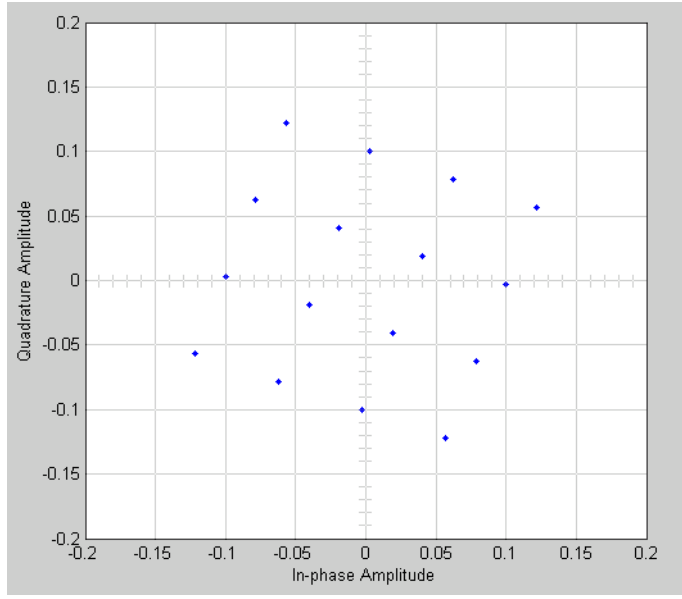
You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model on page 1-129 with the I/Q Imbalance block. Set the block's **I/Q amplitude imbalance (dB)** parameter to 10 and the **I/Q phase imbalance (deg)** parameter to 30.



For more examples of scatter plots produced using this block, see the reference page for the I/Q Imbalance block.

Phase/Frequency Offset Block

You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model on page 1-129 with the Phase/Frequency Offset block. Set the block's **Frequency offset (Hz)** parameter to `0` and the **Phase offset (deg)** parameter to `70`.

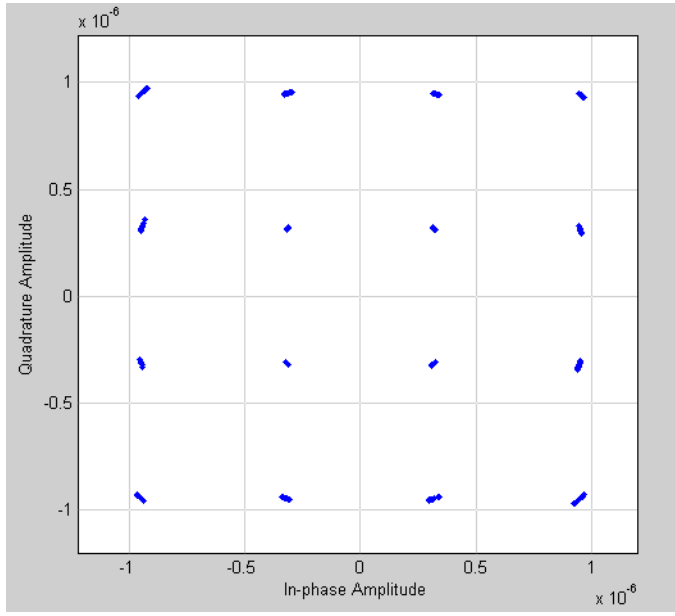


The **Frequency offset (Hz)** parameter adds a constant to the phase of the signal. The scatter plot corresponds to the standard constellation rotated by a fixed angle of 70 degrees.

The **Frequency offset (Hz)** parameter determines the rate of change of the signal's phase. In this example, **Frequency offset (Hz)** is set to `0`, so the scatter plot always falls on the grid shown in the preceding figure. If you set **Frequency offset (Hz)** to a positive number, the points on the scatter plot fall on a rotating grid, corresponding to the standard constellation, which revolves at a constant rate in the counterclockwise direction. For an example, see the reference page for the Phase/Frequency Offset block.

Phase Noise Block

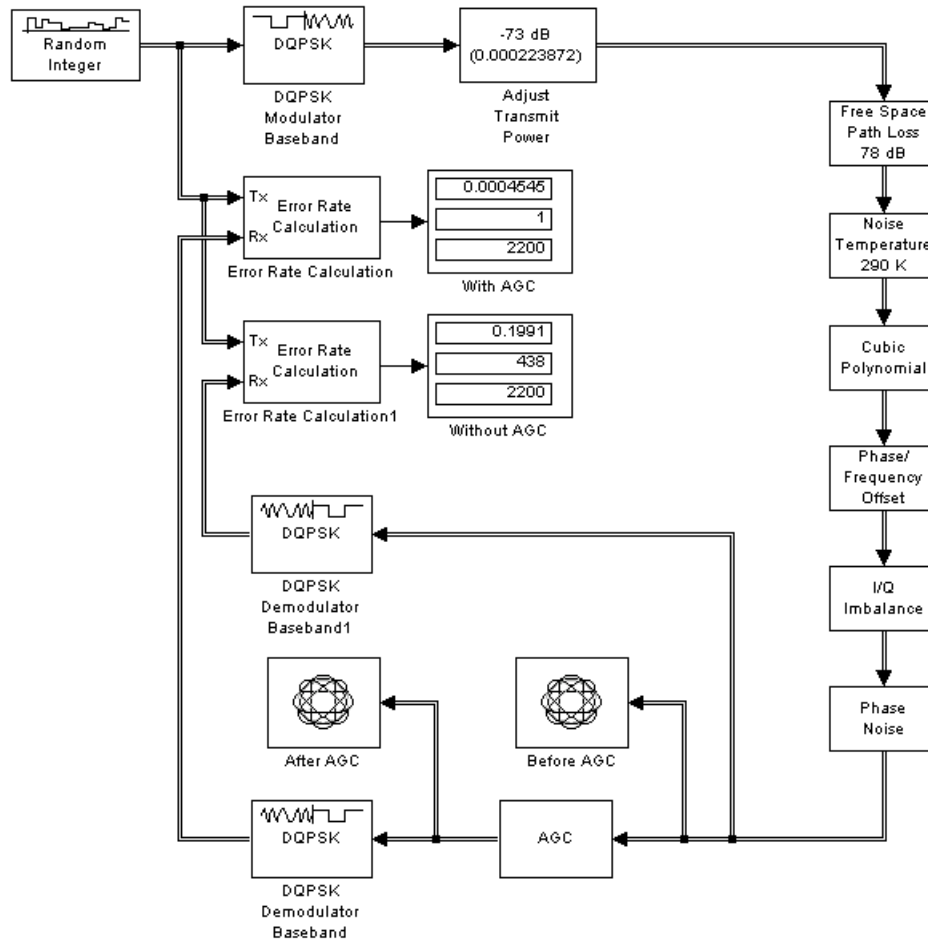
You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model on page 1-129 with the Phase Noise block. Set the **Phase noise level (dBc/Hz)** parameter to `-60` and the **Frequency offset (Hz)** parameter to `100`.



The phase noise adds a random error to the signal's phase, so that the points in the scatter plot are spread in a radial pattern around the constellation points.

Example Using the RF Impairments Library Blocks

The model shown in the following figure simulates RF impairments to a signal modulated by differential quaternary phase shift keying (DQPSK).



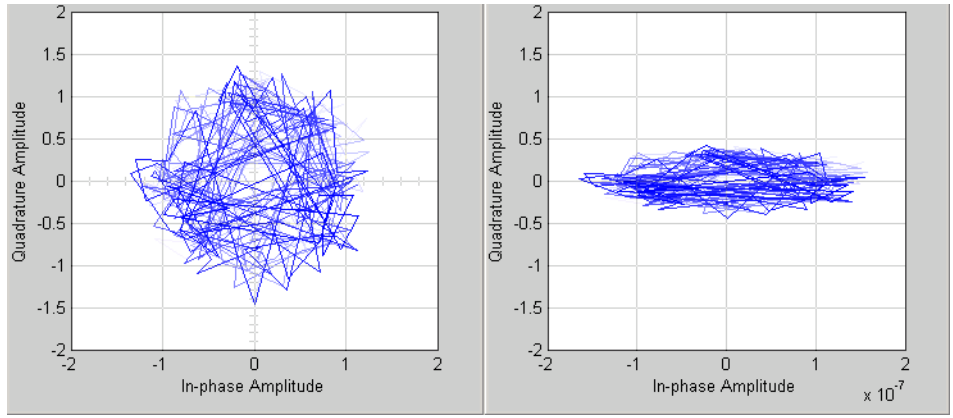
Overview of the Model

The model does the following:

- Modulates a random signal using DQPSK modulation.
- Applies impairments to the signal using the blocks from the RF Impairments library.

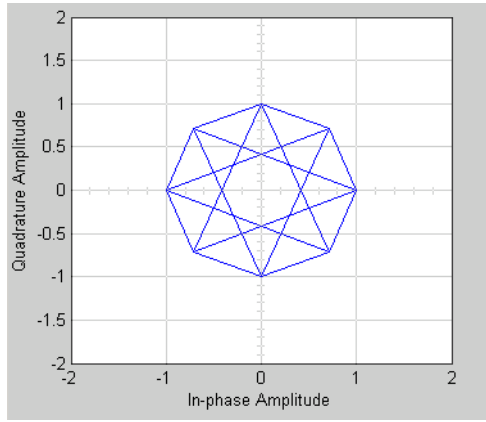
- Forks the signal into two paths, and processes one path with an automatic gain control (AGC) to compensate for the free space path loss and the I/Q imbalance.
- Displays the trajectory of the signal with AGC and the trajectory of the signal without AGC.
- Demodulates both signals and calculates their error rates.

You can see the effect of the automatic gain by comparing the trajectories of the signals with and without AGC, as shown in the following figure.



Signal With (Left) and Without (Right) AGC

The trajectory of the signal with AGC more closely matches the undistorted trajectory for DQPSK, shown in the following figure, than does than the signal without AGC. Consequently, the error rate for the signal with AGC is much lower than the error rate for the signal without AGC.



In this example, the error rate for the demodulated signal without AGC is primarily caused by free space path loss and I/Q imbalance. The QPSK modulation minimizes the effects of the other impairments.

Synchronization

In order to interpret information correctly, a communication receiver must be synchronized with the corresponding transmitter. This can be achieved in both analog and digital domains. A digital receiver must sample the signal at an appropriate instant within the symbol period, and must estimate the carrier phase. Alternatively, analog components such as voltage-controlled oscillators (VCOs) and phase-locked loops (PLLs) can enable a receiver to adjust its behavior based on the parameters of the incoming signals or the desired signals.

Synchronization Features of the Blockset

This blockset implements several algorithms for timing phase recovery and carrier phase recovery. It also includes some lower-level components that you can use to build your own PLLs. This section describes the capabilities of the Synchronization library's blocks, in these key sections:

- "Timing Phase Recovery" on page 1-138
- "Carrier Phase Recovery" on page 1-149
- "Components" on page 1-156

For background material on the subject of synchronization, see the works listed in "Selected Bibliography for Synchronization" on page 1-159.

Accessing Synchronization Blocks

Open the Synchronization library by double-clicking its icon in the main Communications Blockset library. Then open the sublibraries by double-clicking their icons in the Synchronization library.

Timing Phase Recovery

The Timing Phase Recovery library contains blocks that implement various algorithms for determining the best instant within a symbol period to sample a signal at the receiver. For example, the best instant for a PSK-modulated signal is at the peak of the pulse shape. Sampling at the best instant improves the receiver's performance on a noisy signal. Typically, you would place

a timing phase recovery block after a receive filter that is matched to the transmitting pulse shape, and before a demodulator.

This section about timing phase recovery covers these topics:

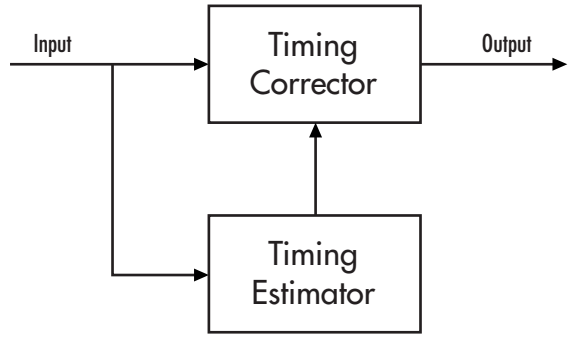
Supported Algorithms for Timing Phase Recovery

This library supports the algorithms listed below, which are all digital recovery methods rather than conventional analog phase-locked loops. For more information about each algorithm, see the reference works cited on each block's reference entry.

Algorithm	Block
Squaring method (feedforward)	Squaring Timing Recovery
Early-late gate method (feedback)	Early-Late Gate Timing Recovery
Gardner's method (feedback)	Gardner Timing Recovery
Fourth-order nonlinearity method (feedback)	MSK-Type Signal Timing Recovery
Mueller-Muller method (feedback)	Mueller-Muller Timing Recovery

Feedforward Method for Timing Phase Recovery

A feedforward method for timing phase recovery is structured as in the following figure.



In the figure,

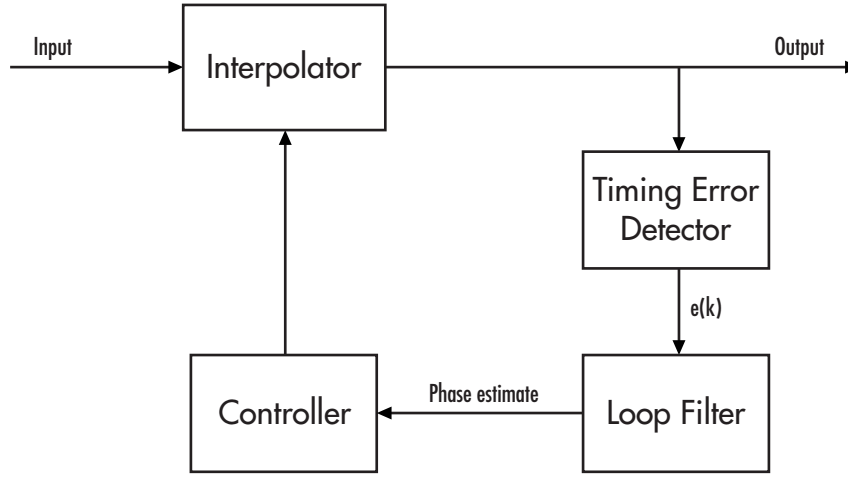
- The input signal is typically the output of a receive filter that is matched to the transmit pulse shape.
- The timing estimator gives an estimate of the input signal's sampling phase.
- The timing corrector is a sampler that outputs the value of the input signal corresponding to the phase estimate. The timing corrector interpolates between input signal values if necessary.

Squaring Timing Recovery block

The Squaring Timing Recovery block implements a feedforward method for timing phase recovery. In this method, the timing estimator uses a complex Fourier coefficient to determine the spectral component of the squared input signal at frequency 1/T, where T is the symbol period. For the specific equation, see the reference page for the Squaring Timing Recovery block.

Feedback Methods for Timing Phase Recovery

The Timing Phase Recovery library implements several feedback methods for timing phase recovery. A feedback method for timing phase recovery is structured as in the following figure.



In the figure,

- The input signal is typically the output of a receive filter that is matched to the transmit pulse shape.
- The interpolator generates additional samples based on the needs of the timing error detector. As implemented here, the interpolator uses linear interpolation between pairs of points.
- The timing error detector generates a timing error signal for each symbol. The algorithm used for timing error detection depends on the library block.
- The loop filter updates the phase estimate for the current symbol using the timing error signal and the previous symbol's phase estimate. The phase estimate for the (k+1)st symbol is $[[TAU]]_{k+1} = [[TAU]]_k + g*e(k)$, where g is the step size (also the **Error update gain** parameter in the feedback-method blocks in this library) and e(k) is the timing error for the kth symbol.
- The controller uses the phase estimates to determine the interpolating instants that the interpolator uses in the next cycle.

Restarting the Phase Estimating Process During the Simulation

When using a feedback method for timing phase recovery in Simulink, you can restart the phase-estimation process at different points during the simulation.

Restarting the process means resetting the data buffer and phase-estimate buffer to the all-zeros state. The table below lists the supported options.

Value of Reset Parameter	When Estimation Process Restarts
`None`	At beginning of simulation only. During the simulation, the block operates continuously, retaining information from one symbol to the next.
`Every frame`	Regularly, at the start of each frame of data. During the simulation, each frame of data is processed independently. This option is valid only with frame-based data.
`On nonzero input via port`	Whenever the second input (`Rst`) is nonzero. When the first input is sample-based, its symbol period must equal the sample time of `Rst`. When the first input is frame-based, its frame period must equal the sample time of `Rst`, and the reset occurs at the start of the frame.

Using the Restarting Options Effectively. If you restart the phase-estimation process during the simulation, be sure to include enough symbols between successive resets for the algorithm to converge to a stable value. Check the phase (`Ph`) output from the block to see whether its values stabilize before the reset occurs. To include more symbols between successive resets, either increase the frame size by buffering frames together (when using the `Every frame` option) or change the `Rst` input so that nonzero values occur less frequently.

Choosing a Method for Timing Phase Recovery

Depending on your system, one or more recovery methods implemented in this library might be suitable. If you use a method that is not suitable for your system, the results might not be accurate. This section discusses the assumptions and suitability of the various methods, covering these topics:

Squaring Timing Recovery Block

The Squaring Timing Recovery block recovers the symbol-timing phase of the input signal using a squaring method. This frame-based, feedforward, nondata-aided method is similar to a conventional squaring loop.

This block is suitable for systems that use linear baseband modulation types such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, and quadrature amplitude modulation (QAM).

The block assumes that the phase offset is constant for all symbols in the entire input frame. If necessary, you can use the Buffer block to reorganize your data into frames over which the phase offset can be assumed constant.

Assumptions Common to All Feedback Method Blocks

The feedback method, as implemented in this library, makes some assumptions about the data it receives:

- The phase varies slowly over time. Although the blocks compute a phase estimate for each symbol, the estimate should remain approximately constant for several symbols or else the algorithm does not converge.
- The symbol frequency is constant and known. Small variations in phase correspond to a frequency offset, but the blocks do not compensate for it. The blocks estimate and correct only the phase, not the frequency.

Although the blocks that implement feedback methods share a common structure and the common assumptions above, the blocks use different algorithms in the timing error detector and incur different delays. See each block's reference entry for details.

Early-Late Gate Timing Recovery Block

The Early-Late Gate Timing Recovery block implements a nondata-aided feedback method.

This block is suitable for systems that use a linear modulation type, such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), with Nyquist pulses (for example, using a raised cosine filter). In the presence of noise, the performance of this timing recovery method improves as the pulse's excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

The early-late gate method is similar to Gardner's method, which is implemented in the Gardner Timing Recovery block. Some differences between the two methods are as follows:

- In the ideal case (that is, when the phase estimate is zero and the input signal has symmetric Nyquist pulses), the timing error detector for the early-late gate method requires samples that span one symbol interval, rather than two symbol intervals as in Gardner's method.
- Compared to Gardner's method, the early-late gate method has higher self noise and thus does not perform as well as Gardner's method in systems with high SNR values.

Gardner Timing Recovery Block

The Gardner Timing Recovery block implements a nondata-aided feedback method that is independent of carrier phase recovery.

This block is suitable for both baseband systems and modulated carrier systems. More specifically, this block is suitable for systems that use a linear modulation type with Nyquist pulses that have an excess bandwidth between approximately 40% and 100%. Examples of suitable systems are those that use pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, or quadrature amplitude modulation (QAM), and that shape the signal using raised cosine filters whose rolloff factor is between 0.4 and 1. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth (rolloff factor in the case of a raised cosine filter) increases.

Gardner's method is similar to the early-late gate method, which is implemented in the Early-Late Gate Timing Recovery block.

MSK-Type Signal Timing Recovery Block

The MSK-Type Signal Timing Recovery block recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general nondata-aided feedback method that is independent of carrier phase recovery but that requires prior compensation for the carrier frequency offset.

This block is suitable for systems that use baseband minimum shift keying (MSK) modulation or Gaussian minimum shift keying (GMSK) modulation. Unlike the other blocks in this library, this block does not require the input signal to have been filtered beforehand.

Mueller-Muller Timing Recovery Block

The Mueller-Muller Timing Recovery block implements a decision-directed, data-aided feedback method that requires prior recovery of the carrier phase.

This block is suitable for systems that use a *binary* linear modulation type, such as binary phase shift keying (BPSK) modulation, or binary phase amplitude modulation (BPAM). The binary requirement arises because the algorithm uses a sign detector (that is, a 1-bit quantizer) to arrive at decisions. When the input signal has Nyquist pulses (for example, using a raised cosine filter), this timing recovery method has no self noise. In the presence of noise, the performance of this timing recovery method improves as the pulse's excess bandwidth factor decreases, making the method a good candidate for narrowband signaling.

Examples of Timing Phase Recovery

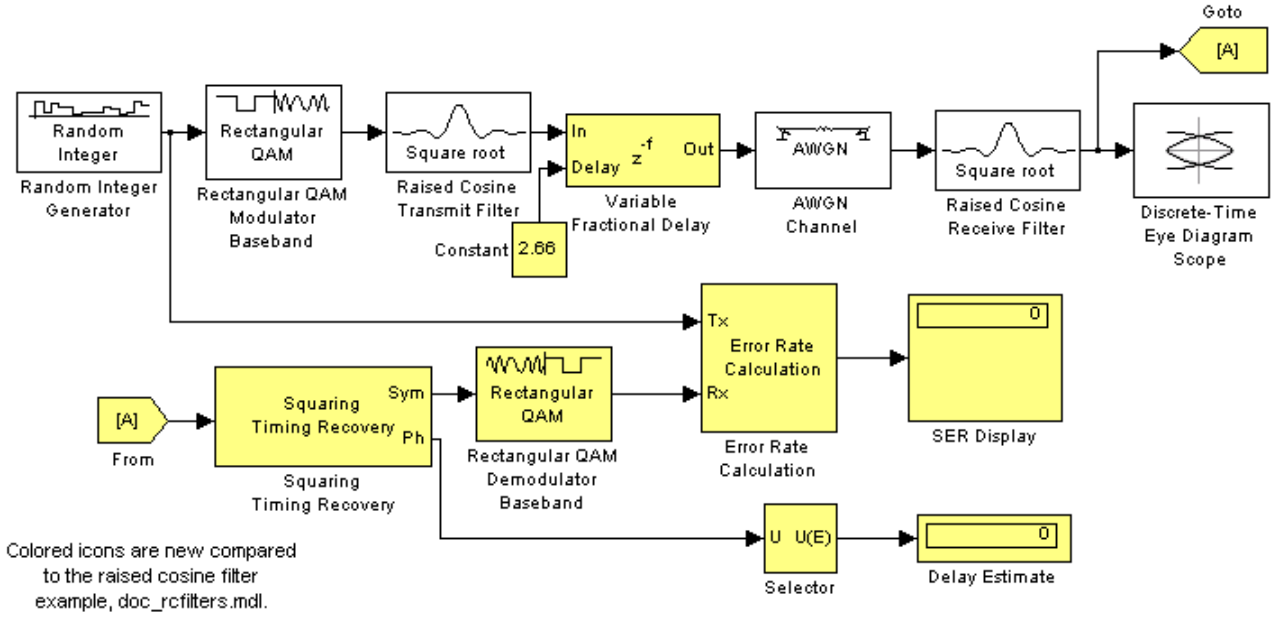
One way to illustrate the usage and behavior of the timing phase recovery blocks is to introduce a fractional delay in a communications link and then see how well the block estimates the delay value and samples the received signal. In this context, a "fractional delay" is a delay that is not a multiple of the signal's symbol period. The examples discussed here are

- "Squaring Timing Phase Recovery Example" on page 1-146, described below. This model introduces a fixed fractional delay and uses a feedforward method for timing phase recovery.
- Gardner timing phase recovery demo, which you can open by entering `gardner_vfracdelay` in the MATLAB Command Window. This model introduces a fractional delay that varies from frame to frame and uses a feedback method for timing phase recovery.

Squaring Timing Phase Recovery Example

This example modifies the one in "Example: Using Raised Cosine Filters" on page 1-119 by introducing and then correcting for a fixed fractional delay. The model uses the Squaring Timing Recovery block to estimate that delay and determine the best instant within the symbol to sample its input signal. The model then demodulates the downsampled signal and computes a symbol error rate.

To open the completed model, click here in the MATLAB Help browser.



To build the model, first open the raised cosine filter model by clicking here in the MATLAB Help browser. Then, gather and configure these blocks:

- Variable Fractional Delay, in the Signal Processing Blockset Signal Operations library. Use default parameters.
- Constant, in the Simulink Sources library
  - Set **Constant value** to `2.66`. This is the number of samples of delay introduced in the system.
- Goto and From, in the Simulink Signal Routing library. Use default parameters.
- Selector, in the Simulink Signal Routing library
  - Set **Elements** to `1`. This causes the block to select the first value in the frame, all of whose entries are actually the same.
  - Set **Input port width** to `100`.
- Squaring Timing Recovery
  - Set **Samples per symbol** to `8`.
- Rectangular QAM Demodulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Peak Power`.
  - Set **Peak power** to `1`.
- Error Rate Calculation, in the Comm Sinks library
  - Set **Receive delay** to `8`. This accounts for the delay of the pair of square root raised cosine filters.
  - Set **Output data** to `Port`.
- Two copies of Display, in the Simulink Sinks library. Make one tall enough to accommodate three values.

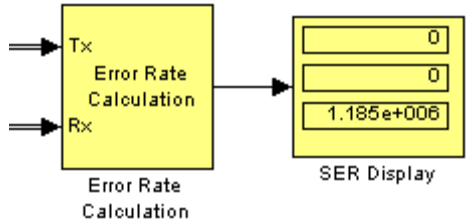
Connect the blocks as in the figure, and then run the simulation.

Results of the Simulation

When you run the simulation, look for these results:

- A delay estimate that varies during the simulation but is near the fixed value of 2.66. The Squaring Timing Recovery block computes this delay estimate for each frame and then uses it to choose a sampling instant for the symbols in that frame.

- A symbol error rate that is small or zero, depending on how long you run the simulation. For most or all symbols, the Squaring Timing Recovery block determines a sampling instant that enables the demodulator to recover data correctly.

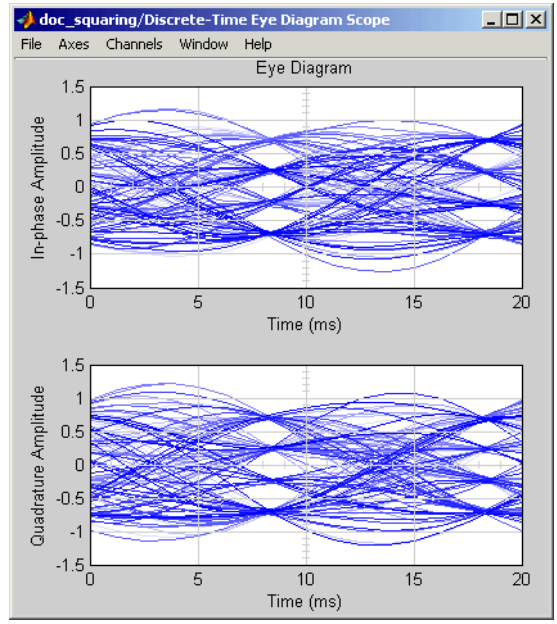


- An eye diagram that has two widely opened “eyes” near 8.325 ms and 18.325 ms. These wide openings indicate appropriate instants at which to sample the filtered signal before demodulating, and reflect the introduced delay of 2.66 samples.

  To arrive at the numbers 8.325 and 18.325, reason as follows: The eye diagram displays two symbols per trace, and each symbol has a period of 10 ms. Without the introduced delay, the centers of the trace’s two symbols are at 5 ms and 15 ms. The delay value in each symbol is

  (2.66 samples) / (8 samples/symbol) * (10 ms/symbol) = 3.325 ms

  Therefore, the traces from the delayed signal have their widest openings at (5+3.325) ms and (15+3.325) ms.



While this example uses a fixed delay throughout the simulation, the blocks in the timing recovery library can also correct for delays that vary (slowly) from symbol to symbol. For an example that uses a varying delay, see the Gardner timing phase recovery demo.

Carrier Phase Recovery

The Carrier Phase Recovery library contains blocks that implement digital algorithms for determining the carrier phase of a baseband digital signal. The blocks assume that the carrier frequency is known and fixed. The blocks output the estimated carrier phase as well as a corrected (that is, rotated) version of the input signal. Typically, you place a carrier phase recovery block before a demodulator, and after a timing phase recovery block or another block that produces symbols rather than an upsampled signal.

This section about carrier phase recovery covers these topics:

Supported Algorithms for Carrier Phase Recovery

This library supports the algorithms listed below, which are all digital recovery methods rather than conventional analog methods. For more information about each algorithm, see the reference works cited on each block's reference entry.

Algorithm	Block
2P-power method, suitable for full-response CPM, MSK, CPFSK, or GMSK signals.	CPM Phase Recovery
M-power method, suitable for M-PSK signals. (Also, the 4-power method is suitable for QAM signals using any alphabet size.)	M-PSK Phase Recovery

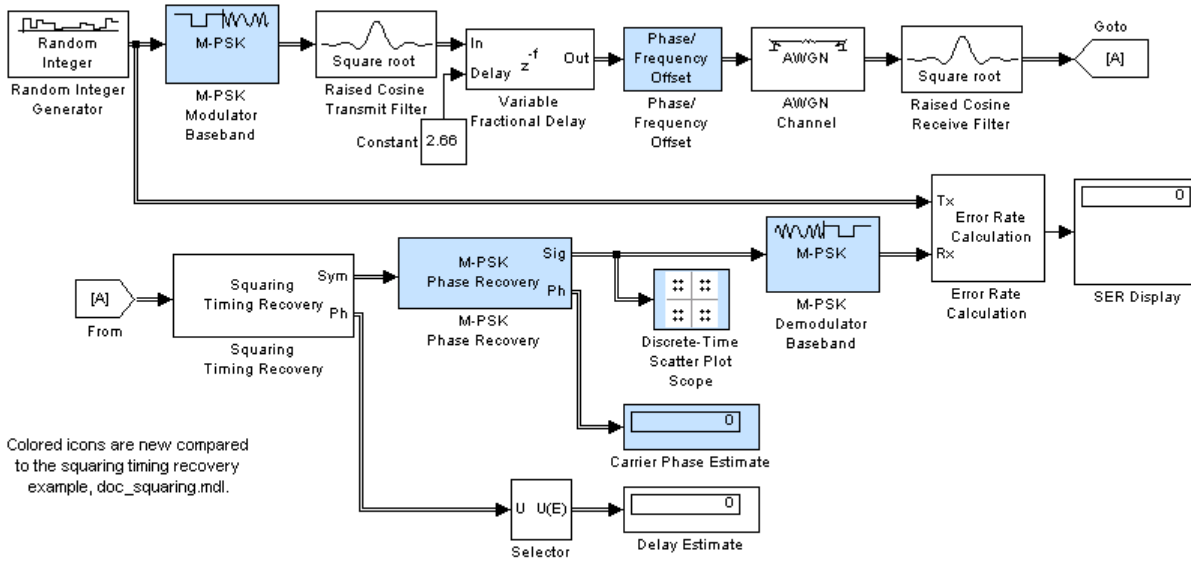
The methods described in the table are nondata-aided, clock-aided, feedforward methods. They assume that timing and carrier frequency are already known and any matched filtering has already been performed.

The methods also assume that the carrier phase to be estimated is constant over a series of consecutive symbols. When you use the blocks in this library, you specify the number of symbols over which the carrier phase is assumed constant.

Carrier Phase Recovery Example

This example modifies the one in "Squaring Timing Phase Recovery Example" on page 1-146 by introducing and then correcting for a fixed phase offset. The model uses the M-PSK Phase Recovery block to estimate the offset and correct the received baseband signal by rotating it. The model then demodulates the corrected signal and computes a symbol error rate.

To open the completed model, click here in the MATLAB Help browser.



To build the model, first open the squaring timing phase recovery model by clicking here in the MATLAB Help browser. Then, gather and configure these blocks:

- M-PSK Modulator Baseband and M-PSK Demodulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation. In each block,
  - Set **M-ary number** to 16.
  - Set **Phase offset** to 0.
- Phase/Frequency Offset, in the RF Impairments library
  - Set **Phase offset** to 10.
- M-PSK Phase Recovery
  - Set **M-ary number** to 16.
  - Set **Observation interval** to 500.
- Discrete-Time Scatter Plot Scope, in the Comm Sinks library
  - Set **Points displayed** to 400.

- Display, in the Simulink Sinks library

Replace the Rectangular QAM Modulator Baseband and Rectangular QAM Demodulator Baseband blocks with the corresponding M-PSK blocks listed above.

Remove the Discrete-Time Eye Diagram Scope block and the branched signal line leading to it.

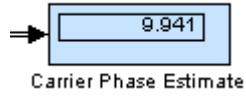
In the Error Rate Calculation block, change **Computation delay** to `500`, because the M-PSK Phase Recovery block has a latency of one observation interval. This latency is described on the M-PSK Phase Recovery block's reference page.

Connect the remaining blocks as in the figure, and then run the simulation.

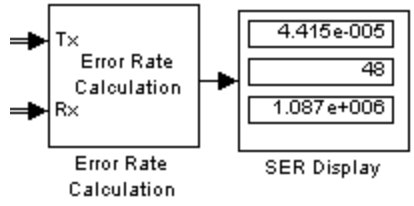
Results of the Simulation

When you run the simulation, look for these results:

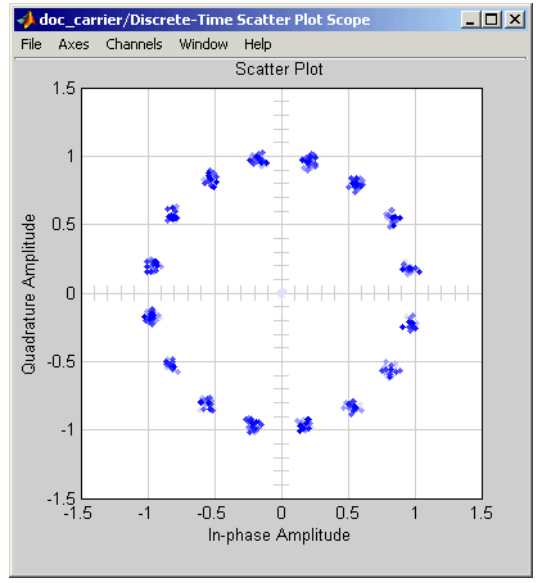
- A carrier phase estimate that varies during the simulation but is near the fixed value of 10 degrees. The M-PSK Phase Recovery block computes this carrier phase estimate for each frame and then uses it to correct the phase of the symbols in that frame.



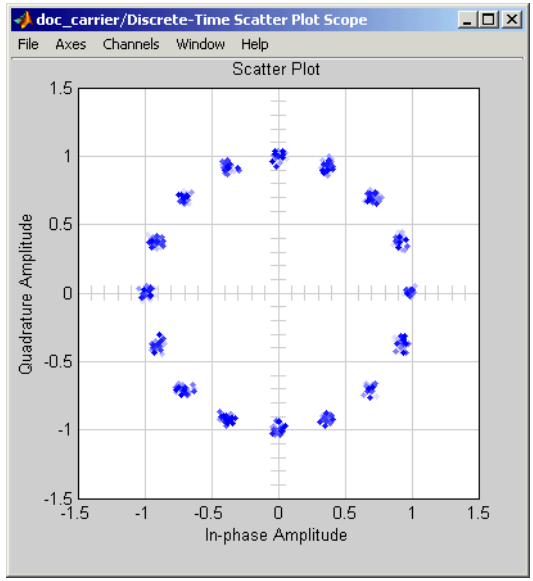
- A symbol error rate that is small or zero, depending on how long you run the simulation. For most or all symbols, the M-PSK Phase Recovery block enables the demodulator to recover data correctly.



- A signal constellation that reflects the signal whose phase the M-PSK Phase Recovery block has corrected. When you first begin the simulation and the block is in an initial latency period, the constellation reflects the phase offset of 10 degrees, with no correction. After the latency period is over, the constellation shows no phase offset because the M-PSK Phase Recovery block has corrected for it. The constellations before and after the end of the latency period appear below. The easiest way to see the 10-degree rotation between the two constellations is to look at the axes.



Before End of Latency Period



After End of Latency Period

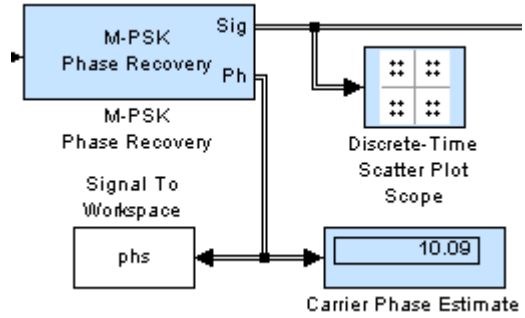
Exploring the Simulation Further

Another way to examine the performance of the carrier phase recovery is to check how much the phase estimates from successive observation intervals differ from each other. You do this using the plotting capabilities of MATLAB along with the simulation capabilities of Simulink:

1. Add a Signal to Workspace block, from the Signal Processing Sinks library, to the carrier phase recovery example model.

2. In the Signal to Workspace block, set **Variable name** to `phs` and set **Limit data points to last** to `200`.

3. Connect the Signal to Workspace block to the `Ph` output of the M-PSK Phase Recovery block, as shown in the following figure.



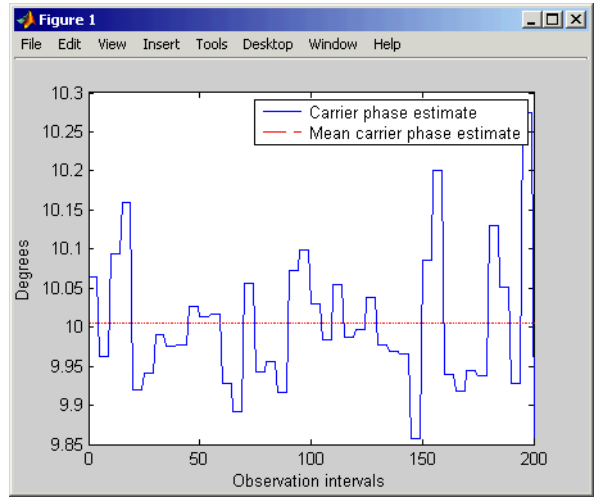
4 In the MATLAB Command Window, enter this command to run the simulation for a finite period of time:

```
sim('doc_carrier',205);
```

You make the simulation run faster by closing the window containing the signal constellation plot. When the simulation ends, the MATLAB workspace contains a variable called `phs` that contains the last 200 phase estimates from the M-PSK Phase Recovery block. Initial zeros from the delay period are omitted.

5 Create a plot showing the phase estimate values as well as their mean value by entering the following in the MATLAB Command Window:

```
plot(1:200,phs,'b-',1:200,mean(phs),'r--')
legend('Carrier phase estimate','Mean carrier phase estimate')
xlabel('Observation intervals'); ylabel('Degrees')
```



The plot shows that the mean is very close to the expected value of 10 degrees, while the individual phase estimates vary within an interval that includes 10 degrees.

Components

The Components sublibrary contains voltage-controlled oscillator (VCO) models as well as phase-locked loop (PLL) models.

This section discusses these topics:

For details about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" on page 1-159.

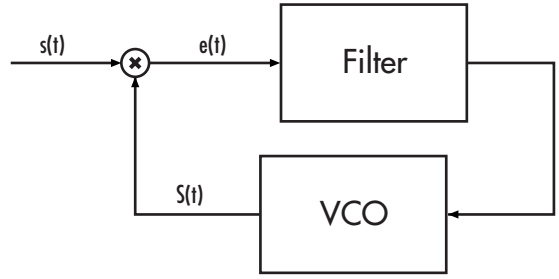
Voltage-Controlled Oscillator Blocks

A voltage-controlled oscillator is one part of a phase-locked loop. The Continuous-Time VCO and Discrete-Time VCO blocks implement voltage-controlled oscillators. These blocks produce continuous-time and discrete-time output signals, respectively. Each block's output signal is sinusoidal, and changes its frequency in response to the amplitude variations of the input signal.

Overview of PLL Simulation

A phase-locked loop (PLL), when used in conjunction with other components, helps synchronize the receiver. A PLL is an automatic control system that adjusts the phase of a local signal to match the phase of the received signal. The PLL design works best for narrowband signals.

A simple PLL consists of a phase detector, a loop filter, and a voltage-controlled oscillator (VCO). For example, the following figure shows how these components are arranged for an analog passband PLL. In this case, the phase detector is just a multiplier. The signal e(t) is often called the error signal.



The following table indicates the supported types of PLLs and the blocks that implement them.

Supported PLLs in Components Library

Type of PLL	Block
Analog passband PLL	Phase-Locked Loop
Analog baseband PLL	Baseband PLL

Supported PLLs in Components Library (Continued)

Type of PLL	Block
Linearized analog baseband PLL	Linearized Baseband PLL
Digital PLL using a charge pump	Charge Pump PLL

Different PLLs use different phase detectors, filters, and VCO characteristics. Some of these attributes are built into the PLL blocks in this blockset, while others depend on parameters that you set in the block mask:

- You specify the filter's transfer function in the block mask using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each of these parameters is a vector that lists the coefficients of the respective polynomial in order of descending exponents of the variable *s*. To design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in the Signal Processing Toolbox.
- You specify the key VCO characteristics in the block mask. All four PLL blocks use a **VCO input sensitivity** parameter. Some blocks also use **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.
- The phase detector for each of the PLL blocks is a feature that you cannot change from the block mask.

Implementing an Analog Baseband PLL

Unlike passband models for a phase-locked loop, a baseband model does not depend on a carrier frequency. This allows you to use a lower sampling rate in the simulation. Two blocks implement analog baseband PLLs:

- Baseband PLL
- Linearized Baseband PLL

The linearized model and the nonlinearized model differ in that the linearized model uses the approximation

$$\sin\left(\Delta\theta(t)\right) \cong \Delta\theta(t)$$

to simplify the computations. This approximation is close when $\Delta\theta(t)$ is near zero. Thus, instead of using the input signal and the VCO output signal directly, the linearized PLL model uses only their *phases*.

Implementing a Digital PLL

The charge pump PLL is a classical digital PLL. Unlike the analog PLLs mentioned above, the charge pump PLL uses a sequential logic phase detector, which is also known as a digital phase detector or a phase/frequency detector.

Selected Bibliography for Synchronization

[1] Gardner, F.M., "Charge-pump Phase-lock Loops," *IEEE Trans. on Communications*, Vol. 28, November 1980, pp. 1849–1858.

[2] Gardner, F.M., "Phase Accuracy of Charge Pump PLLs," *IEEE Trans. on Communications*, Vol. 30, October 1982, pp. 2362–2363.

[3] Gupta, S.C., "Phase Locked Loops," *Proceedings of the IEEE*, Vol. 63, February 1975, pp. 291–306.

[4] Lindsay, W.C. and C.M. Chie, "A Survey on Digital Phase-Locked Loops," *Proceedings of the IEEE*, Vol. 69, April 1981, pp. 410–431.

[5] Mengali, Umberto, and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[6] Meyr, Heinrich, and Gerd Ascheid, *Synchronization in Digital Communications*, Vol. 1, New York, John Wiley & Sons, 1990.

[7] Moeneclaey, Marc, and Geert de Jonghe, "ML-Oriented NDA Carrier Synchronization for General Rotationally Symmetric Signal Constellations," *IEEE Transactions on Communications*, Vol. 42, No. 8, Aug. 1994, pp. 2531–2533.

Equalizers

Time-dispersive channels can cause intersymbol interference (ISI). For example, in a multipath scattering environment, the receiver sees delayed versions of a symbol transmission, which can interfere with other symbol transmissions. An equalizer attempts to mitigate ISI and thus improve the receiver's performance.

This section describes the capabilities of the Equalizers library's blocks, covering these topics:

Sources of Background Material

To learn more about equalizers in general, see these sections of the Communications Toolbox documentation about equalizer functions:

- "Overview of Adaptive Equalizer Classes" gives brief background material on the supported adaptive equalizer types
- "Choosing an Adaptive Algorithm" has a brief description of the different adaptive algorithms, to help you decide which one might be appropriate for your application
- "Selected Bibliography for Equalizers" has a list of published works that contain more detailed background material

Equalization Features of the Blockset

Open the Equalizers library by double-clicking its icon in the main Communications Blockset library. This blockset supports these distinct classes of equalizers, each with a different overall structure:

- Linear equalizers, a class that includes symbol-spaced equalizers and fractionally spaced equalizers

- Decision-feedback equalizers
- MLSE (maximum-likelihood sequence estimation) equalizer that uses the Viterbi algorithm

Linear and decision-feedback equalizers are adaptive equalizers that use an adaptive algorithm when operating. For each of the adaptive equalizer classes listed above, this blockset supports these adaptive algorithms:

- Least mean square (LMS)
- Normalized LMS
- Signed LMS, including these types: sign LMS, sign regressor LMS, and sign-sign LMS
- Variable-step-size LMS
- Recursive least squares (RLS)
- Constant Modulus Algorithm (CMA)

Using Adaptive Equalizers

Several blocks from the Equalizers library implement adaptive equalizers, differing in the equalizer structure and the type of adaptive algorithm that they use. In all cases, you specify information about the equalizer structure (such as the number of taps), the adaptive algorithm (such as the step size), and the signal constellation used by the modulator in your model. You also specify an initial set of weights for the taps of the equalizer; the block adaptively updates the weights throughout the simulation. For adaptive algorithms other than CMA, the equalizer can adapt the weights in two modes: training mode and decision-directed mode. The following sections discuss some configuration details for blocks in the library:

- "Specifying the Signal Constellation of the Modulated Signal" on page 1-162
- "Equalizing Using a Training Sequence" on page 1-162
- "Equalizing in Decision-Directed Mode" on page 1-163
- "Controlling the Use of Training or Decision-Directed Mode" on page 1-163
- "Retrieving the Weights and Error Signal" on page 1-164

Specifying the Signal Constellation of the Modulated Signal

Each equalizer block has a **Signal constellation** parameter that specifies the constellation for the modulated signal, as determined by the modulator in your model. **Signal constellation** is a vector of complex numbers, where the kth complex number in the vector is the constellation point to which the modulator maps the integer k-1.

Note The sequence of constellation points must be consistent between the modulator in your model and the **Signal constellation** parameter in the equalizer block.

Equalizing Using a Training Sequence

In typical applications, an equalizer begins by using a known sequence of transmitted symbols when adapting the equalizer weights. The known sequence, called a *training sequence*, enables the equalizer to gather information about the channel characteristics. After the equalizer finishes processing the training sequence, it adapts the equalizer weights in decision-directed mode using a detected version of the output signal. CMA equalizers are an exception, using neither training mode nor decision-directed mode.

To train a non-CMA equalizer block at the beginning of each frame throughout the simulation, follow these steps:

1. Clear the **Mode input port** check box.
2. Provide the training sequence at the input port labeled `Desired`. Valid training symbols are those listed in the **Signal constellation** vector. The block operates in training mode at the beginning of each frame and switches to decision-directed mode when it runs out of training symbols.

Typically, the symbol periods of the `Input` and `Desired` inputs match; that is, the sample time of the `Desired` signal is k times the sample time of the `Input` signal, where k is the **Number of samples per symbol** parameter in the equalizer block. If your training sequence is constant throughout the simulation, the Simulink Constant block is a convenient way to specify the sequence without having to specify a sample time explicitly.

To train a non-CMA equalizer block only on selected frames during the simulation, see "Controlling the Use of Training or Decision-Directed Mode" on page 1-163.

Equalizing in Decision-Directed Mode

Decision-directed mode means that the equalizer uses a detected version of its output signal when adapting the weights. Adaptive equalizers typically start with a training sequence (as mentioned in "Equalizing Using a Training Sequence" on page 1-162) and switch to decision-directed mode after exhausting all symbols in the training sequence. CMA equalizers are an exception, using neither training mode nor decision-directed mode. The non-CMA equalizer blocks in this library operate in decision-directed mode when one of these conditions is true:

- The equalizer started processing the current input frame in training mode, exhausted all symbols in the training sequence frame, and still has more input symbols to process.
- The **Mode input port** check box is selected and the `Mode` input signal is `0`.

Controlling the Use of Training or Decision-Directed Mode

You can configure a non-CMA equalizer block so that it adapts in training mode for the beginning or the entirety of *selected* frames. To achieve this level of control over the equalizer's mode, follow these steps:

1. Enable the `Mode` input port by checking the **Mode input port** check box.
2. Send a binary-valued scalar signal to the `Mode` input port. The `Mode` input enables you to toggle back and forth between training mode and decision-directed mode. The significance of this signal is as follows:
   - When the `Mode` input is `0`, the equalizer operates in decision-directed mode on the entire frame and ignores the `Desired` input.
   - When the `Mode` input is `1`, the equalizer operates in training mode at the beginning of the frame until it exhausts the symbols in the `Desired` input, and operates in decision-directed mode afterwards. If the `Mode` input is `1` and the `Desired` input has as many symbols as the `Input` signal has, then the equalizer operates in training mode on the entire frame.

Retrieving the Weights and Error Signal

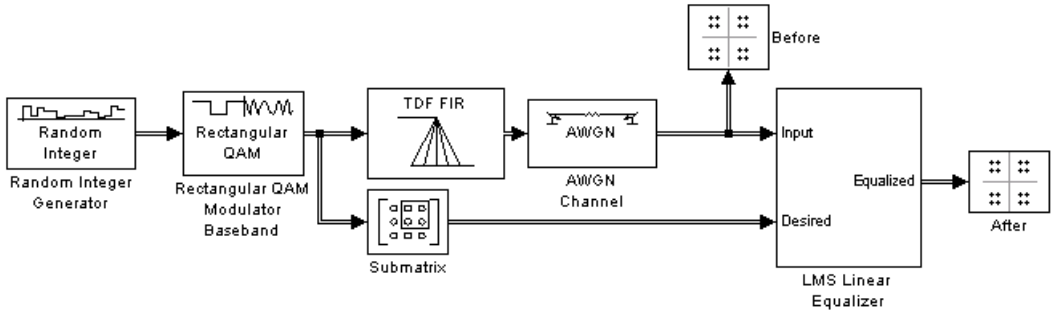
You can configure the equalizer block so that it outputs the weights and/or error signal throughout the simulation. The table below indicates how to enable the extra output ports for these signals.

Signal	Port Label	How to Enable
Error signal. For the exact definition, see the block's online reference page.	`Err`	Check the **Output error** check box.
A vector listing the weights after the block has processed either the current input frame or, in sample-based mode, the current input sample.	`Wts`	Check the **Output weights** check box.

Example: LMS Linear Equalizer

This example illustrates the usage of an LMS linear equalizer. The simulation transmits a 16-QAM signal, modeling the channel using an FIR filter followed by additive white Gaussian noise. The equalizer receives the signal from the channel and, as training symbols, a subset of the modulator's output. The equalizer operates in training mode at the beginning of each frame and switches to decision-directed mode when it runs out of training symbols. The example contrasts the signals before and after equalization to illustrate the effect of the equalizer.

To open the completed model, click here in the MATLAB Help browser.



To build the model, gather and configure these blocks:

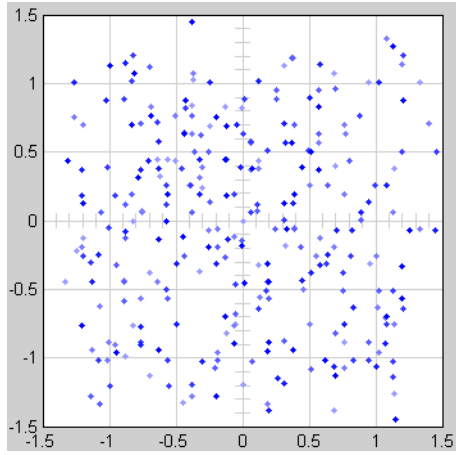
- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `16`.
  - Set **Sample time** to `1/1000`.
  - Select **Frame-based outputs**.
  - Set **Samples per frame** to `1000`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Average Power`.
  - Set **Average power** to `1`.
- Digital Filter, in the Signal Processing Blockset Filter Designs sublibrary of Filtering
  - Set **Transfer function type** to `FIR (all zeros)`.
  - Set **Filter structure** to `Direct form transposed`.
  - Set **Numerator coefficients** to `[1 -.3 .1 .2j]`.
- Submatrix, in the Signal Processing Blockset Indexing sublibrary of Signal Management
  - Set **Ending row** to `Index`.
  - Set **Ending row index** to `100`.
- AWGN Channel, in the Channels library
  - Set **Mode** to `Signal to noise ratio (SNR)`.
  - Set **SNR** to `40`.
- LMS Linear Equalizer
  - Set **Number of taps** to `6`.
  - Clear the **Mode input port**, **Output error**, and **Output weights** check boxes.
- Two copies of Discrete-Time Scatter Plot Scope, in the Comm Sinks library

- Set **Points displayed** to 400 in each of the two copies.

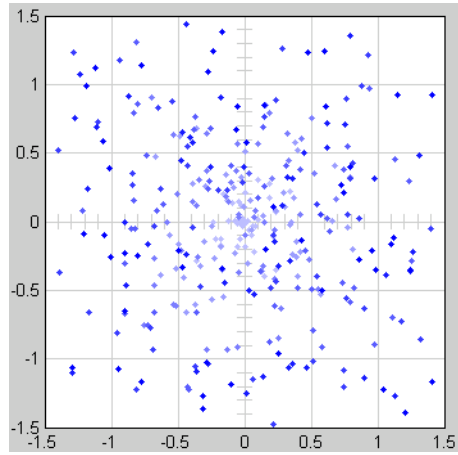
Connect the blocks as in the figure. Running the simulation produces two scatter plots that display the signal before and after equalization, respectively.

Scatter Plots in the Example

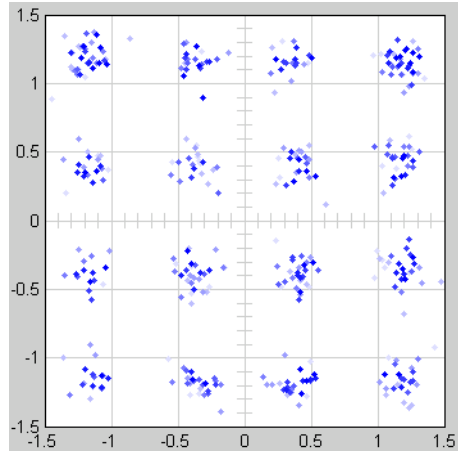
Throughout the simulation, the signal before equalization deviates noticeably from a 16-QAM signal constellation, as below.



Early in the simulation, the equalizer does not appear to improve the scatter plot. In fact, the equalizer is busy trying to adapt its weights appropriately. The following figure shows the equalized signal very early in the simulation.



After some simulation time passes, the equalizer's weights work well on the received signal. As a result, the equalized signal looks far more like a 16-QAM signal constellation than the received signal does. The figure below shows the equalized signal in its steady state.



Using MLSE Equalizers

The MLSE Equalizer block uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The block outputs the

maximum likelihood sequence estimate (MLSE) of the signal, using your estimate of the channel modeled as a finite input response (FIR) filter.

The block decodes the received signal using these steps:

1. Applies the FIR filter corresponding to the channel estimate to the symbols in the input signal.
2. Uses the Viterbi algorithm to compute the traceback paths and the state metric, which are the numbers assigned to the symbols at each step of the Viterbi algorithm.
3. Outputs the maximum likelihood sequence estimate of the signal, as a sequence of complex numbers corresponding to the constellation points of the modulated signal.

An MLSE equalizer yields the best possible performance, in theory, but is computationally intensive.

When using the MLSE Equalizer block, you specify the channel estimate and the signal constellation that the modulator in your model uses. If applicable, you can also specify a preamble and/or postamble that you expect to accompany your data. For full details on options, see the reference page for the MLSE Equalizer block.

2

Modeling Communication Systems

This chapter presents several examples that illustrate techniques for modeling a full communication system rather than a small fragment of one. Because the techniques are mainly relevant in models that involve multiple areas of functionality (for example, modulation combined with block coding), the examples in this chapter are more complicated than the examples of earlier chapters.

Computing Delays

Some models require you to know how long it takes for data in one portion of a model to influence a signal in another portion of a model. For example, when configuring an error rate calculator, you must indicate the delay between the transmitter and the receiver. If you miscalculate the delay, the error rate calculator processes mismatched pairs of data and consequently returns a meaningless result.

This section illustrates the computation of delays in multirate models and in models where the total delay in a sequence of blocks comprises multiple delays from individual blocks. This section also indicates how to use the Find Delay and Align Signals blocks to help deal with delays in a model. The section covers the following topics:

Other References for Delays

Other parts of this documentation set also discuss delays. For information about dealing with delays or about delays in specific types of blocks, see

- Find Delay block reference page
- Align Signals block reference page
- "Delays in Digital Modulation" on page 1-99
- "Delays of Convolutional Interleavers" on page 1-84
- Viterbi Decoder block reference page
- Derepeat block reference page

For discussions of delays in simpler examples than the ones in this section, see

- "Building a Frequency-Shift Keying Model" in the Getting Started with the Communications Blockset documentation. (See "Learning About Delays in the Model".)
- "Example: A Rate 2/3 Feedforward Encoder" on page 1-62.
- "Example: Soft-Decision Decoding" on page 1-67. (See "Delay in Received Data" on page 1-71.)
- "Example: Delays from Demodulation" on page 1-101.

Sources of Delays

While some blocks can determine their current output value using only the current input value, other blocks need input values from multiple time steps to compute the current output value. In the latter situation, the block incurs a delay. An example of this case is when the Derepeat block must average five samples from a scalar signal. The block must delay computing the average until it has received all five samples.

In general, delays in your model might come from various sources:

- Digital demodulators
- Convolutional interleavers or deinterleavers
- Equalizers
- Viterbi Decoder block
- Buffering, downsampling, derepeating, and similar signal operations
- Explicit delay blocks, such as Delay and Variable Integer Delay
- Filters

The following discussions include some of these sources of delay.

ADSL Demo Model

This section examines the asymmetric digital subscriber line (ADSL) demonstration model and aims to compute the correct **Receive delay** parameter value in one of the Error Rate Calculation blocks in the model.

The model includes delays from convolutional interleaving and an explicit delay block. To open the ADSL demo model, enter `adsl_sim` in the MATLAB Command Window.

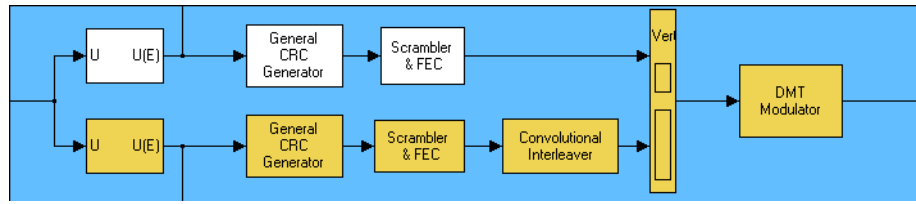
In the ADSL demo, data follows one of two parallel paths, one with a nonzero delay and the other with a delay of zero. One path includes a convolutional interleaver and deinterleaver, while the other does not. Near the end of each path is an Error Rate Calculation block, whose **Receive delay** parameter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

Frame Periods in the Model

Before searching for individual delays, first observe that most signal lines throughout the model share the same frame period; to see this, enable the **Sample time colors** option from the **Port/signal displays** submenu of the model window's **Format** menu. This option colors blocks and signals according to their frame periods (or sample periods, in the case of sample-based signals). All signal lines at the top level of the model are the same color, which means that they share the same frame period. As a consequence, frames are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by each frame's period) reduces to a sum.

Path for Interleaved Data

In the transmitter portion of the model, the interleaved path is the lower branch, shown in yellow below. Similarly, the interleaved path in the receiver portion of the model is the lower branch. Near the end of the interleaved path is an Error Rate Calculation block that computes the value labeled `Interleaved BER`.



The following table summarizes the delays in the path for noninterleaved data. Subsequent paragraphs explain the delays in more detail and explain why the total delay relative to the Error Rate Calculation block is one frame, or 776 samples.

Block	Delay, in Output Samples from Individual Block	Delay, in Frames	Delay, in Input Samples to Error Rate Calculation Block
Convolutional Interleaver and Convolutional Deinterleaver pair	40	1 (combined)	776 (combined)
Delay	800		
Total	N/A	1	776

Interleaving. Unlike the noninterleaved path, the interleaved path contains a Convolutional Interleaver block in the transmitter and a Convolutional Deinterleaver block in the receiver. The delay of the interleaver/deinterleaver pair is the product of the **Rows of shift registers** parameter, the **Register length step** parameter, and one less than the **Rows of shift registers** parameter. In this case, the delay of the interleaver/deinterleaver pair turns out to be 5*2*4 = 40 samples.

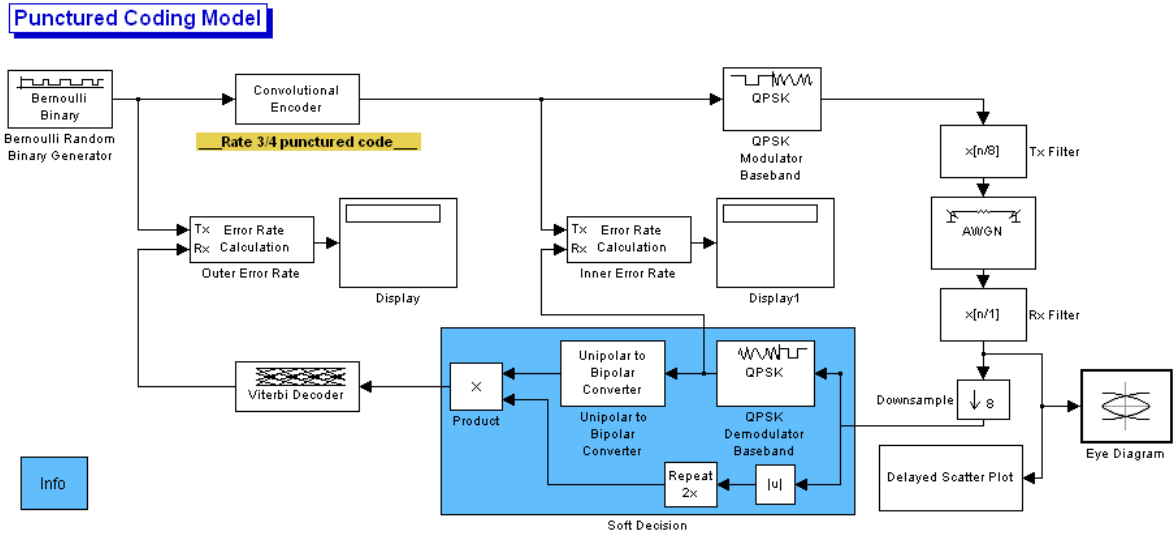
Delay Block. The receiver portion of the interleaved path also contains a Delay block, whose purpose is explained in “Aligning Words of a Block Code” on page 2-18. This block explicitly causes a delay of 800 samples having the same sample time as the 40 samples of delay from the interleaver/deinterleaver pair. Therefore, the total delay from interleaving, deinterleaving, and the explicit delay is 840 samples. These 840 samples make up one frame of data leaving the Delay block.

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

Total Delay Relative to Error Rate Calculation Block. The Error Rate Calculation block that computes the value labeled `Interleaved BER` requires a **Receive delay** parameter value that is equivalent to one frame. The **Receive delay** parameter is measured in samples and each input frame to the Error Rate Calculation block contains 776 samples. Also, the frame rate at the outports of all delay-causing blocks in the interleaved path equals the frame rate at the inport of the Error Rate Calculation block. Therefore, the correct value for the **Receive delay** parameter is `776` samples.

Punctured Coding Model

This section discusses a punctured coding model that includes delays from decoding, downsampling, and filtering. Two Error Rate Calculation blocks in the model work correctly if and only if their **Receive delay** parameters accurately reflect the delays in the model. To open the model, enter `punctdoc` in the MATLAB Command Window.



Frame Periods in the Model

Before searching for individual delays, first enable the **Sample time colors** option from the **Port/signal displays** submenu of the model window's

Format menu. Only the rightmost portion of the model differs in color from the rest of the model. This means that all signals and blocks in the model except those in the rightmost edge share the same frame period. Consequently, frames at this predominant frame rate are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by each frame's period) reduces to a sum.

The yellow blocks represent multirate systems, while the AWGN Channel block and the Rx Filter block run at a higher frame rate than most other blocks in the model.

Inner Error Rate Block

The block labeled Inner Error Rate, located near the center of the model, is a copy of the Error Rate Calculation block from the Sinks library. It computes the bit error rate for the portion of the model that excludes the punctured convolutional code. In the portion of the model between this block's two input signals, delays come from the Tx Filter, Rx Filter, and Downsample blocks, as summarized in the following table. This section explains why the Inner Error Rate block's **Receive delay** parameter is the total delay value of 16.

Block	Delay, in Samples at Individual Block	Delay, in Frames at Predominant Frame Rate	Delay, in Input Samples to Inner Error Rate Block
Tx Filter	3	3/2	6
Rx Filter	3 (relative to input of Tx Filter block)	3/2	6
Downsample	2	1	4
Total	N/A	4	16

Tx Filter Block. The block labeled Tx Filter is a copy of the FIR Interpolation block in the Signal Processing Blockset. It upsamples the input signal by a factor of 8 and applies a square-root raised cosine filter. The value of the block's **FIR filter coefficients** parameter is

```
rcosine(1, 8, 'sqrt', 0.5, 3)
```

where the ratio 3/1 indicates that the delay caused by the filter is 3 times the sample period (not frame period) of the signal before upsampling. Because the input signal is not upsampled and is a two-sample frame at the model's predominant frame rate, the delay is equivalent to 3/2 frames at the predominant frame rate.

Rx Filter Block. The block labeled Rx Filter is another copy of the FIR Interpolation block, but it differs from the Tx Filter block in that its **Interpolation factor** parameter is `1` instead of `8`. The values of that parameter differ in the two filter blocks because the Tx Filter block needs to upsample the signal to prepare for transmission along the channel, while the Rx Filter processes a signal that is already upsampled and needs no further upsampling. Thus the Rx Filter block merely applies a square-root raised cosine filter without upsampling its input data. As in the case of the Tx Filter block, the delay caused by the Rx Filter block is three times the sample period (not frame period) of the signal *without* upsampling. The frame rate without upsampling is just the model's predominant frame rate, so the delay of the Rx Filter block is the same as that of the Tx Filter block. That is, the delay is equivalent to 3/2 frames at the predominant frame rate.

Note This example uses the FIR Interpolation block approach to illustrate how to deal with multiple frame rates in the same model. Alternatively, the model could have used the Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks in the Communications Blockset. In that case, the frame rate would be constant throughout the system, and the total delay values discussed in this document would be different.

Downsample Block. The Downsample block reduces the frame rate of the filtered received data. Its delay is one output frame, as stated on the reference page for the Downsample block. Because the frame rate at the outport equals the model's predominant frame rate, the delay of the Downsample block is one frame at the predominant frame rate.

Summing the Delays. No other blocks in the portion of the model between the Inner Error Rate block's two input signals cause any delays. Adding the two 3/2-frame delays from the two filter blocks with the one-frame delay from the Downsample block indicates that the total delay in this portion of the model is four frames.

Total Delay Relative to Inner Error Rate Block. The Inner Error Rate block requires a **Receive delay** parameter value that is equivalent to four frames. The **Receive delay** parameter is measured in samples and each input frame to the Inner Error Rate block contains four samples. Therefore, the correct value for the **Receive delay** parameter is 16 samples.

Outer Error Rate Block

The block labeled Outer Error Rate, located at the left of the model, is a copy of the Error Rate Calculation block from the Sinks library. It computes the bit error rate for the entire model, including the punctured convolutional code. Delays come from the Tx Filter, Rx Filter, Downsample, and Viterbi Decoder blocks, as summarized in the table below. This section explains why the Outer Error Rate block's **Receive delay** parameter is the total delay value of 108.

Block	Delay, in Samples at Individual Block	Delay, in Frames at Predominant Frame Rate	Delay, in Input Samples to Outer Error Rate Block
Tx Filter	3	3/2	9/2
Rx Filter	3 (relative to input of Tx Filter block)	3/2	9/2
Downsample	2	1	3
Viterbi Decoder	96	32	96
Total	N/A	36	108

Filter and Downsample Blocks. The Tx Filter, Rx Filter, and Downsample blocks have a combined delay of four frames at the model's predominant frame rate. For details, see "Inner Error Rate Block" on page 2-7.

Viterbi Decoder Block. The Viterbi Decoder block decodes the convolutional code, and the algorithm's use of a traceback path causes a delay. The block processes a frame-based signal and has **Operation mode** set to `Continuous`. Therefore, the delay, measured in output samples, is equal to the **Traceback depth** parameter value of `96`. (The delay amount is stated on the reference page for the Viterbi Decoder block.) Because the output of the Viterbi Decoder block is precisely one of the inputs to the Outer Error Rate block, it is easier to consider the delay to be 96 samples rather than to convert it to an equivalent number of frames.

Total Delay Relative to Outer Error Rate Block. The Outer Error Rate block requires a **Receive delay** parameter value that is equivalent to four frames plus 96 samples. The **Receive delay** parameter is measured in samples, and each input frame to the Outer Error Rate block contains three samples. Therefore, the correct value for the **Receive delay** parameter is `4*3+96 = 108` samples.

Note The Outer Error Rate block accounts for the four-frame delay from filtering and downsampling by expressing it as 12 samples when computing the **Receive delay** parameter. Recall that the Inner Error Rate block accounts for the same four-frame delay but expresses it as 16 samples, not 12. The expressions differ because the two error rate blocks express delays in terms of samples rather than frames, yet process signals of different sizes.

Using the Find Delay and Align Signals Blocks

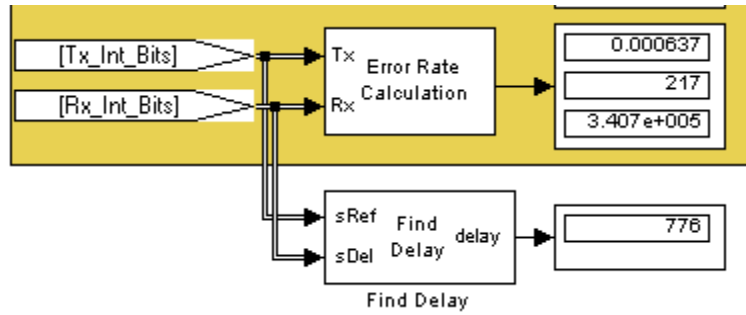
The preceding discussions explained why certain Error Rate Calculation blocks in the models had specific **Receive delay** parameter values. You could have arrived at those numbers independently by using the Find Delay block, or you could have avoided needing to know those numbers by using the Align Signals block. This section explains both techniques using the ADSL demo model, `adsl_sim`, as an example. Applying the techniques to the punctured convolutional coding example, discussed in "Punctured Coding Model" on page 2-6, would be similar.

Using the Find Delay Block to Determine the Correct Receive Delay

Recall from "Path for Interleaved Data" on page 2-4 that the delay in the path for interleaved data is 776 samples. To have the Find Delay block compute that value for you, use this procedure:

1. Insert a Find Delay block and a Display block in the model near the Error Rate Calculation block that computes the value labeled `Interleaved BER`.

2. Connect the blocks as shown below.



3. Set the Find Delay block's **Correlation window length** parameter to a value substantially larger than 776, such as 2000.

Note You must use a sufficiently large correlation window length or else the values produced by the Find Delay block do not stabilize at a correct value.

---

4. Run the simulation.

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

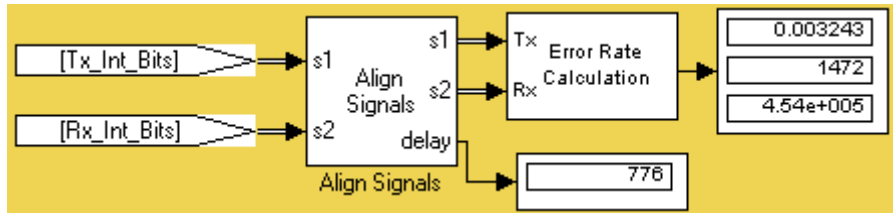
Using the Align Signals Block Before Computing the Error Rate

To use the Error Rate Calculation block to compute the value labeled `Interleaved BER` *without* having to set the **Receive delay** parameter to a nonzero value, you can use the Align Signals block to automatically align the

transmitted and received signals before the Error Rate Calculation block performs its computations. Use this procedure:

1. Insert an Align Signals block and a Display block in the model near the Error Rate Calculation block that computes the value labeled `Interleaved BER`.

2. Connect the blocks as shown below.



3. Set the Align Signals block's **Correlation window length** parameter to a value substantially larger than 776, such as 2000.

Note You must use a sufficiently large correlation window length or else the Align Signals block cannot find the correct amount by which to delay one of the signals. If the `delay` output from the Align Signals block does not stabilize at a constant value, the correlation window length is probably too small.

---

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

5. Run the simulation.

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

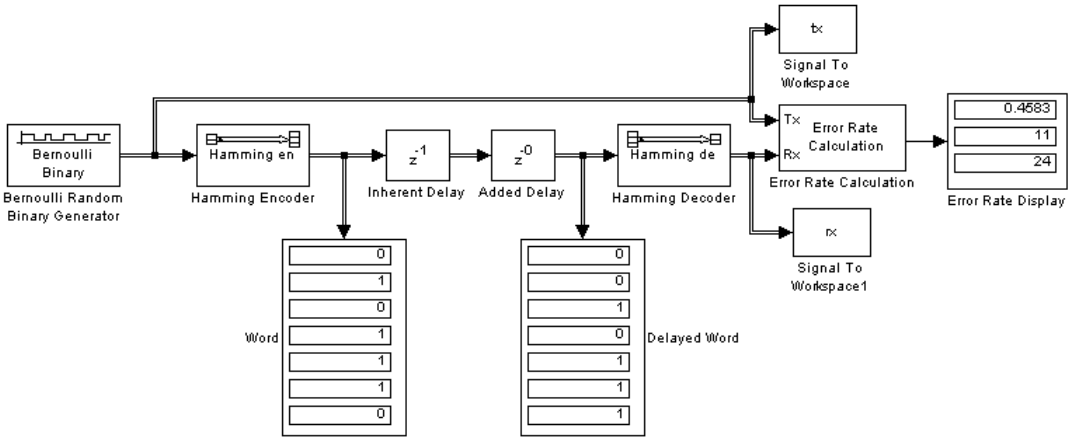
Examining the delay output signal from the Align Signals block, using the Display block as in the figure above, is important because if the delay output signal does not stabilize at a constant value, the signals are not truly aligned and the error rate is not reliable. In this case, the Align Signals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

Manipulating Delays

Some models require you not only to compute delays but to manipulate them. For example, if a model incurs a delay between a block encoder and its corresponding decoder, the decoder might misinterpret the boundaries between the codewords that it receives and, consequently, return meaningless results. More generally, such a situation can arise when the path between paired components of a block-oriented operation (such as interleaving, block coding, or bit-to-integer conversions) includes a delay-causing operation (such as those listed in "Sources of Delays" on page 2-3). To avoid this problem, you can insert an additional delay of an appropriate amount between the encoder and decoder. If the model also computes an error rate, then the additional delay affects that process, as described in "Computing Delays" on page 2-2. This section uses examples to illustrate the purpose, methods, and implications of manipulating delays in a variety of circumstances. The subsections are

Delays and Alignment Problems

This section illustrates the sensitivity of block-oriented operations to delays, using a small model that aims to capture the essence of the problem in a simple form. Open the model by entering `alignmentdoc` in the MATLAB Command Window. Then run the simulation so that the Display blocks show relevant values.



In this model, two coding blocks create and decode a block code. Two copies of the Delay block create a delay between the encoder and decoder. The two Delay blocks have different purposes in this illustrative model:

- The Inherent Delay block represents any delay-causing blocks that might occur in a model between the encoder and decoder. See “Sources of Delays” on page 2-3 for a list of possibilities that might occur in a more realistic model.
- The Added Delay block is an explicit delay that you insert to produce an appropriate amount of total delay between the encoder and decoder. For example, the `adsl_sim` model contains a Delay block that serves this purpose.

Observing the Problem

By default, the **Delay** parameters in the Inherent Delay and Added Delay blocks are set to `1` and `0`, respectively. This represents the situation in which some operation causes a one-bit delay between the encoder and decoder, but you have not yet tried to compensate for it. The total delay between the encoder and decoder is one bit. You can see from the blocks labeled Word and Delayed Word that the codeword that leaves the encoder is shifted downward by one bit by the time it enters the decoder. The decoder receives a signal in which the boundary of the codeword is at the second bit in the frame, instead

of coinciding with the beginning of the frame. That is, the codewords and the frames that hold them are not aligned with each other.

This nonalignment is problematic because the Hamming Decoder block assumes that each frame begins a new codeword. As a result, it tries to decode a word that consists of the last bit of one output frame from the encoder followed by the first six bits of the next output frame from the encoder. You can see from the Error Rate Display block that the error rate from this decoding operation is close to 1/2. That is, the decoder rarely recovers the original message correctly.

To use an analogy, suppose someone corrupts a paragraph of prose by moving each period symbol from the end of the sentence to the end of the first word of the next sentence. If you try to read such a paragraph while assuming that a new sentence begins after a period, you misunderstand the start and end of each sentence. As a result, you might fail to understand the meaning of the paragraph.

To see how delays of different amounts affect the decoder's performance, vary the values of the **Delay** parameter in the Added Delay block and the **Receive delay** parameter in the Error Rate Calculation block and then run the simulation again. Many combinations of parameter values produce error rates that are close to 1/2. Furthermore, if you examine the transmitted and received data by entering

```
[tx rx]
```

in the MATLAB Command Window, you might not detect any correlation between the transmitted and received data.

Correcting the Delays

Some combinations of parameter values produce error rates of zero because the delays are appropriate for the system. For example:

- In the Added Delay block, set **Delay** to `6`.
- In the Error Rate Calculation block, set **Receive delay** to `4`.
- Run the simulation.
- Enter `[tx rx]` in the MATLAB Command Window.

The top number in the Error Rate Display block shows that the error rate is zero. The decoder recovered each transmitted message correctly. However, the Word and Displayed Word blocks do not show matching values. It is not immediately clear how the encoder's output and the decoder's input are related to each other. To clarify the matter, examine the output in the MATLAB Command Window. The sequence along the first column (`tx`) appears in the second column (`rx`) four rows later. To confirm this, enter

```
isequal(tx(1:end-4),rx(5:end))
```

in the MATLAB Command Window and observe that the result is `1` (true). This last command tests whether the first column matches a shifted version of the second column. Shifting the MATLAB vector `rx` by four rows corresponds to the Error Rate Calculation block's behavior when its **Receive delay** parameter is set to `4`.

To summarize, these special values of the **Delay** and **Receive delay** parameters work for these reasons:

- Combined, the Inherent Delay and Added Delay blocks delay the encoded signal by a full codeword rather than by a partial codeword. Thus the decoder is correct in its assumption that a codeword boundary falls at the beginning of an input frame and decodes the words correctly. However, the delay in the encoded signal causes each recovered message to appear one word later, that is, four bits later.
- The Error Rate Calculation block compensates for the one-word delay in the system by comparing each word of the transmitted signal with the data four bits later in the received signal. In this way, it correctly concludes that the decoder's error rate is zero.

Note These are not the only parameter values that produce error rates of zero. Because the code in this model is a (7, 4) block code and the inherent delay value is 1, you can set the **Delay** and **Receive delay** parameters to 7k-1 and 4k, respectively, for any positive integer k. It is important that the sum of the inherent delay (1) and the added delay (7k-1) is a multiple of the codeword length (7).

Aligning Words of a Block Code

The ADSL demo, discussed in "ADSL Demo Model" on page 2-3, illustrates the need to manipulate the delay in a model so that each frame of data that enters a block decoder has a codeword boundary at the beginning of the frame. The need arises because the path between a block encoder and block decoder includes a delay-causing convolutional interleaving operation. This section explains why the model uses a Delay block to manipulate the delay between the convolutional deinterleaver and the block decoder, and why the Delay block is configured as it is. To open the ADSL demo model, enter `adsl_sim` in the MATLAB Command Window.

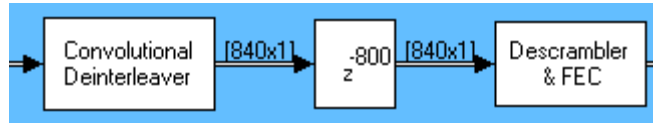
Misalignment of Codewords

In the ADSL demo, the Convolutional Interleaver and Convolutional Deinterleaver blocks appear after the Scrambler & FEC subsystems but before the Descrambler & FEC subsystems. These two subsystems contain blocks that perform Reed-Solomon coding, and the coding blocks expect each frame of input data to start on a new word rather than in the middle of a word.

As discussed in "Path for Interleaved Data" on page 2-4, the delay of the interleaver/deinterleaver pair is 40 samples. However, the input to the Descrambler & FEC subsystem is a frame of size 840, and 40 is not a multiple of 840. Consequently, the signal that exits the Convolutional Deinterleaver block is a frame whose first entry does *not* represent the beginning of a new codeword. As described in "Observing the Problem" on page 2-15, this misalignment, between codewords and the frames that contain them, prevents the decoder from decoding correctly.

Inserting a Delay to Correct the Alignment

The ADSL demo solves the problem by moving the word boundary from the 41st sample of the 840-sample frame to the first sample of a successive frame. Moving the word boundary is equivalent to delaying the signal. To this end, the demo contains a Delay block between the Convolutional Deinterleaver block and the Descrambler & FEC subsystem.



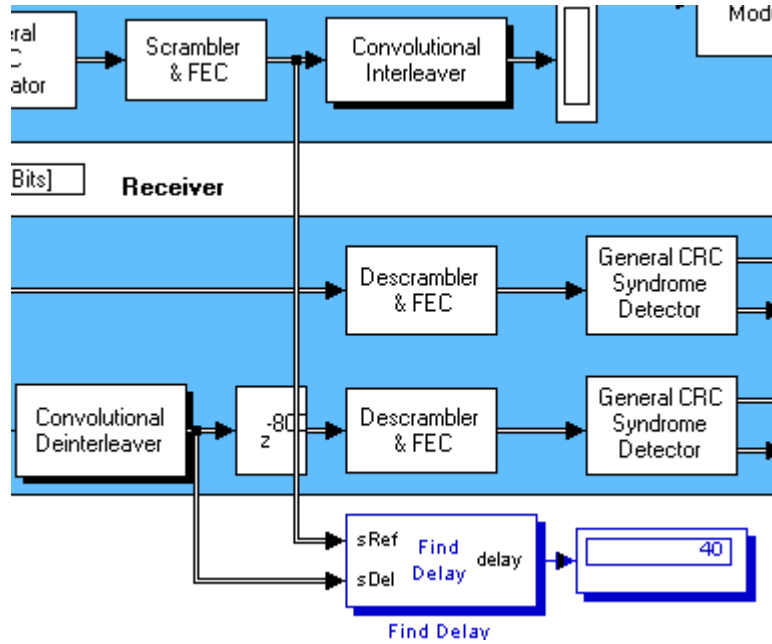
The **Delay** parameter in the Delay block is `800` because that is the minimum number of samples required to shift the 41st sample of one 840-sample frame to the first sample of the next 840-sample frame. In other words, the sum of the inherent 40-sample delay (from the interleaving/deinterleaving process) and the artificial 800-sample delay is a full frame of data, not a partial frame.

This 800-sample delay has implications for other parts of the model, specifically, the **Receive delay** parameter in one of the Error Rate Calculation blocks. For details about how the delay influences the value of that parameter, see “Path for Interleaved Data” on page 2-4.

Using the Find Delay Block

The preceding discussion explained why an 800-sample delay is necessary to correct the misalignment between codewords and the frames that contain them. Knowing that the Descrambler & FEC subsystem requires frame boundaries to occur on word boundaries, you could have arrived at the number 800 independently by using the Find Delay block. Use this procedure:

1. Insert a Find Delay block and a Display block in the model.
2. Create a branch line that connects the input of the Convolutional Interleaver block to the `sRef` input of the Find Delay block.
3. Create another branch line that connects the output of the Convolutional Deinterleaver block to the `sDel` input of the Find Delay block.
4. Connect the `delay` output of the Find Delay block to the new Display block. The modified part of the model now looks like the following image (which also shows drop shadows on key blocks to emphasize the modifications).



5 Show the dimensions of each signal in the model by enabling the **Signal dimensions** feature from the **Port/signal displays** submenu of the model window's **Format** menu.

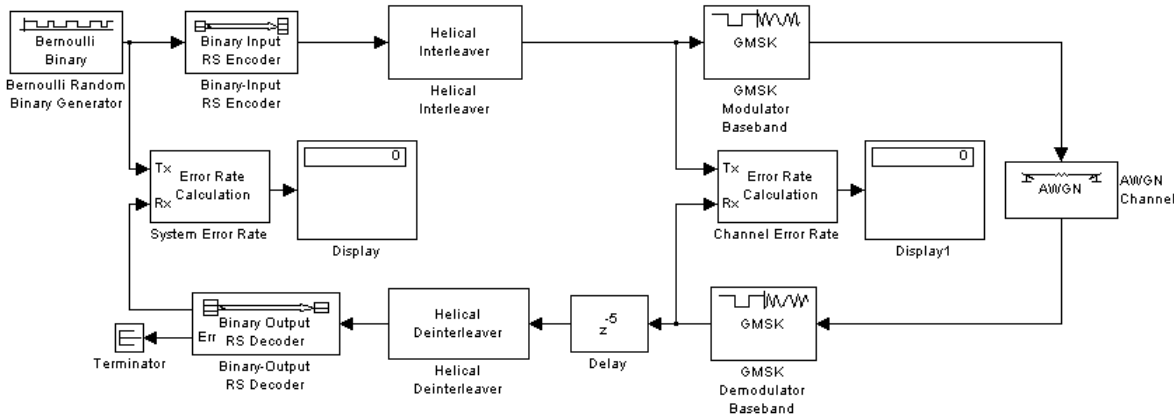
6 Run the simulation.

The new Display block now shows the value 40. Also, the display of signal dimensions shows that the output from the Convolutional Deinterleaver block is a frame of length 840. These results indicate that the sequence of blocks between the Convolutional Interleaver and Convolutional Deinterleaver, inclusive, delays an 840-sample frame by 40 samples. An additional delay of 800 samples brings the total delay to 840. Because the total delay is now a multiple of the frame length, the delayed deinterleaved data can be decoded.

Aligning Words for Interleaving

This section describes an example that manipulates the delay before a deinterleaver, because the path between the interleaver and deinterleaver

includes a delay from demodulation. To open the model, enter `gmskintdoc` in the MATLAB Command Window.



The model includes block coding, helical interleaving, and GMSK modulation. The table below summarizes the individual block delays in the model.

Block	Delay, in Output Samples from Individual Block	Reference
GMSK Demodulator Baseband	16	"Delays in Digital Modulation" on page 1-99
Helical Deinterleaver	42	"Delays of Convolutional Interleavers" on page 1-84
Delay	5	Delay reference page

Misalignment of Interleaved Words

The demodulation process in this model causes a delay between the interleaver and deinterleaver. Because the deinterleaver expects each frame of input data to start on a new word, it is important to ensure that the total

delay between the interleaver and deinterleaver includes one or more full frames but no partial frames.

The delay of the demodulator is 16 output samples. However, the input to the Helical Deinterleaver block is a frame of size 21, and 16 is not a multiple of 21. Consequently, the signal that exits the GMSK Demodulator Baseband block is a frame whose first entry does *not* represent the beginning of a new word. As described in "Observing the Problem" on page 2-15, this misalignment between words and the frames that contain them hinders the deinterleaver.

Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 17th sample of the 21-sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by five samples. The Delay block between the GMSK Demodulator Baseband block and the Helical Deinterleaver block accomplishes such a delay. The Delay block has its **Delay** parameter set to `5`.

Combining the effects of the demodulator and the Delay block, the total delay between the interleaver and deinterleaver is a full 21-sample frame of data, not a partial frame.

Checking Alignment of Block Codewords

The interleaver and deinterleaver cause a combined delay of 42 samples measured at the output from the Helical Deinterleaver block. Because the delayed output from the deinterleaver goes next to a Reed-Solomon decoder, and because the decoder expects each frame of input data to start on a new word, it is important to ensure that the total delay between the encoder and decoder includes one or more full frames but no partial frames.

In this case, the 42-sample delay is exactly two frames. Therefore, it is not necessary to insert a Delay block between the Helical Deinterleaver block and the Binary-Output RS Decoder block.

Computing Delays to Configure the Error Rate Calculation Blocks

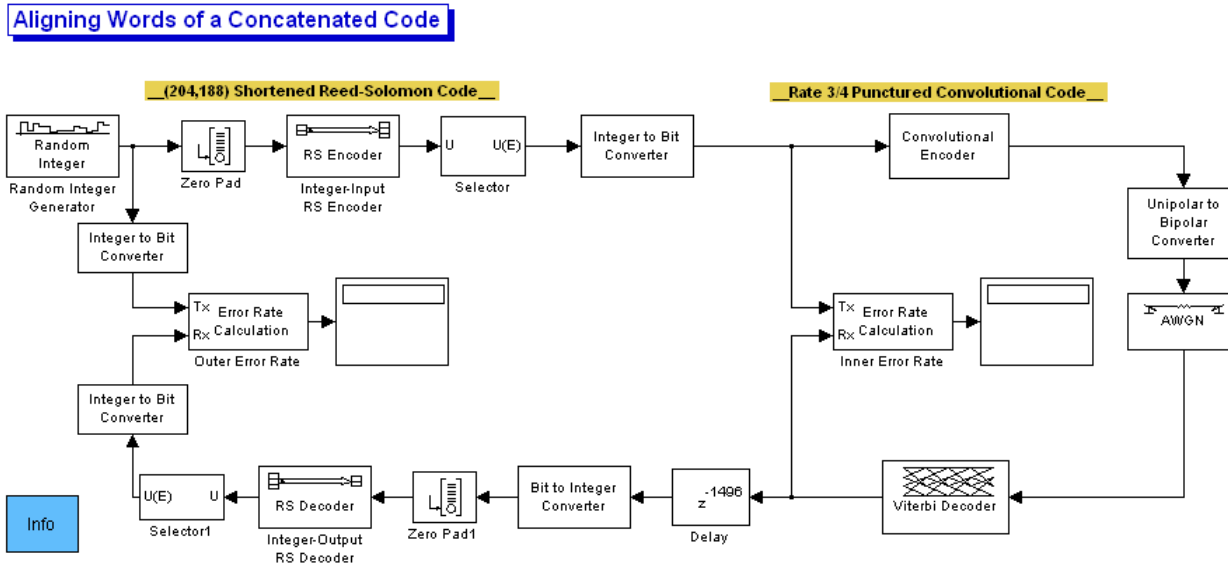
The model contains two Error Rate Calculation blocks, labeled Channel Error Rate and System Error Rate. Each of these blocks has a **Receive delay** parameter that must reflect the delay of the path between the block's

`Tx` and `Rx` signals. The following table explains the **Receive delay** values in the two blocks.

Block	Receive Delay Value	Reason
Channel Error Rate	`16`	Delay of GMSK Demodulator Baseband block, in samples
System Error Rate	`15*3`	Three fifteen-sample frames: one frame from the GMSK Demodulator Baseband and Delay blocks, and two frames from the interleaver/deinterleaver pair

Aligning Words of a Concatenated Code

This section describes an example that manipulates the delay between the two portions of a concatenated code decoder, because the first portion includes a delay from Viterbi decoding while the second portion expects frame boundaries to coincide with word boundaries. To open the model, enter `concatdoc` in the MATLAB Command Window. It uses the block and convolutional codes from the `dvbt_sim` demo, but simplifies the overall design a great deal.



The model includes a shortened block code and a punctured convolutional code. All signals and blocks in the model share the same frame period. The following table summarizes the individual block delays in the model.

Block	Delay, in Output Samples from Individual Block
Viterbi Decoder	136
Delay	1496 (that is, 1632 - 136)

Misalignment of Block Codewords

The Viterbi decoding process in this model causes a delay between the Integer to Bit Converter block and the Bit to Integer Converter block. Because the latter block expects each frame of input data to start on a new 8-bit word, it is important to ensure that the total delay between the two converter blocks includes one or more full frames but no partial frames.

The delay of the Viterbi Decoder block is 136 output samples. However, the input to the Bit to Integer Converter block is a frame of size 1632.

Consequently, the signal that exits the Viterbi Decoder block is a frame whose first entry does *not* represent the beginning of a new word. As described in "Observing the Problem" on page 2-15, this misalignment between words and the frames that contain them hinders the converter block.

Note The outer decoder in this model (Integer-Output RS Decoder) also expects each frame of input data to start on a new codeword. Therefore, the misalignment issue in this model affects many concatenated code designs, not just those that convert between binary-valued and integer-valued signals.

Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 137th sample of the 1632-sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by 1632-136 samples. The Delay block between the Viterbi Decoder block and the Bit to Integer Converter block accomplishes such a delay. The Delay block has its **Delay** parameter set to `1496`.

Combining the effects of the Viterbi Decoder block and the Delay block, the total delay between the interleaver and deinterleaver is a full 1632-sample frame of data, not a partial frame.

Computing Delays to Configure the Error Rate Calculation Blocks

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

Block	Receive Delay Value	Reason
Inner Error Rate	`136`	Delay of Viterbi Decoder block, in samples
Outer Error Rate	`1504 (188*8 bits)`	One 188-sample frame, from the combination of the inherent delay of the Viterbi Decoder block and the added delay of the Delay block

3

Data Type Support

The inputs and outputs of the communications blocks support various data types. This chapter gives an overview of the data type support. For details, refer to the block reference pages.

Communications Block Data Type Support

The following table shows which data types are supported for each block in the Communications Blockset. '•' indicates support for both main input and main output ports. 'I' and 'O' respectively indicate support for either the main input ports or main output ports.

For some blocks, changing to single outputs can lead to different results when compared with double outputs for the same set of parameters. Also, some blocks may naturally output different data types than what they receive (e.g. digital modulators). See individual block reference pages for details.

Channels Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
AWGN Channel	•	•			
Binary Symmetric Channel	•		•		
Multipath Rayleigh Fading Channel	•				
Multipath Rician Fading Channel	•				

Communications Filters Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Gaussian Filter	•	•			•
Ideal Rectangular Pulse Filter	•	•			•
Integrate and Dump	•	•			
Raised Cosine Receive Filter	•	•			•

Communications Filters Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Raised Cosine Transmit Filter	•	•			•
Windowed Integrator	•	•			•

Communications Sinks Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Discrete-Time Eye Diagram Scope	•	•	•	•	•
Discrete-Time Scatter Plot Scope	•	•		•	•
Discrete-Time Signal Trajectory Scope	•	•		•	•
Error Rate Calculation	•	•	•	•	

Communications Sources (Noise Generators) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Gaussian Noise Generator	•	O			
Rayleigh Noise Generator	•	O			

Communications Sources (Noise Generators) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Rician Noise Generator	•	O			
Uniform Noise Generator	•	O			

Communications Sources (Random Data Sources) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Bernoulli Binary Generator	•	•	•	•	
Poisson Integer Generator	•			•	
Random Integer Generator	•	•	•	•	

Communications Sources (Sequence Generators) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Barker Code Generator	•			•	
Gold Sequence Generator	•		•		
Hadamard Code Generator	•			•	
Kasami Sequence Generator	•		•		
OVSF Code Generator	•			•	

Communications Sources (Sequence Generators) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
PN Sequence Generator	•		•		
Walsh Code Generator	•			•	

Equalizers Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
CMA Equalizer	•				
LMS Decision Feedback Equalizer	•				
LMS Linear Equalizer	•				
MLSE Equalizer	•	•			
Normalized LMS Decision Feedback Equalizer	•				
Normalized LMS Linear Equalizer	•				
RLS Decision Feedback Equalizer	•				
RLS Linear Equalizer	•				
Sign LMS Decision Feedback Equalizer	•				
Sign LMS Linear Equalizer	•				

Equalizers Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Variable Step LMS Decision Feedback Equalizer	•				
Variable Step LMS Linear Equalizer	•				

Error Detection and Correction (Block) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
BCH Decoder	•		•		
BCH Encoder	•		•		
Binary Cyclic Decoder	•		•		
Binary Cyclic Encoder	•		•		
Binary Linear Decoder	•		•		
Binary Linear Encoder	•		•		
Binary-Input RS Encoder	•		•		
Binary-Output RS Decoder	•		•		
Hamming Decoder	•		•		
Hamming Encoder	•		•		

Error Detection and Correction (Block) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Integer-Input RS Encoder	•	•	•	•	
Integer-Output RS Decoder	•	•	•	•	

Error Detection and Correction (Convolutional) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
APP Decoder	•	•			
Convolutional Encoder	•	•	•	•	
Viterbi Decoder	•	•	•	•	•

Error Detection and Correction (CRC) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
CRC-N Generator	•		•		
CRC-N Syndrome Detector	•		•		
General CRC Generator	•		•		
General CRC Syndrome Detector	•		•		

Interleaving (Block) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Algebraic Deinterleaver	•	•	•	•	•
Algebraic Interleaver	•	•	•	•	•
General Block Deinterleaver	•	•	•	•	•

Interleaving (Block) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
General Block Interleaver	•	•	•	•	•
Matrix Deinterleaver	•	•	•	•	•
Matrix Helical Scan Deinterleaver	•	•	•	•	•
Matrix Helical Scan Interleaver	•	•	•	•	•
Matrix Interleaver	•	•	•	•	•
Random Deinterleaver	•	•	•	•	•
Random Interleaver	•	•	•	•	•

Interleaving (Convolutional) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Convolutional Deinterleaver	•	•	•	•	•
Convolutional Interleaver	•	•	•	•	•
General Multiplexed Deinterleaver	•	•	•	•	•
General Multiplexed Interleaver	•	•	•	•	•
Helical Deinterleaver	•	•	•	•	•
Helical Interleaver	•	•	•	•	•

Modulation (Analog Passband) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
DSB AM Demodulator Passband	•				
DSB AM Modulator Passband	•				

Modulation (Analog Passband) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
DSBSC AM Demodulator Passband	•				
DSBSC AM Modulator Passband	•				
FM Demodulator Passband	•				
FM Modulator Passband	•				
PM Demodulator Passband	•				
PM Modulator Passband	•				
SSB AM Demodulator Passband	•				
SSB AM Modulator Passband	•				

Modulation (Digital Baseband AM) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
General QAM Demodulator Baseband	•	•	O	O	
General QAM Modulator Baseband	•	O	I	I	•
M-PAM Demodulator Baseband	•	•	O	O	•
M-PAM Modulator Baseband	•	O	I	I	•

Modulation (Digital Baseband AM) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Rectangular QAM Demodulator Baseband	•	•	O	O	
Rectangular QAM Modulator Baseband	•	O	I	I	•

Modulation (Digital Baseband CPM) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
CPFSK Demodulator Baseband	•	I	O	O	
CPFSK Modulator Baseband	•	O	I	I	
CPM Demodulator Baseband	•	I	O	O	
CPM Modulator Baseband	•	O	I	I	
GMSK Demodulator Baseband	•	I	O	O	
GMSK Modulator Baseband	•	O	I	I	

Modulation (Digital Baseband CPM) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
MSK Demodulator Baseband	•	I	O	O	
MSK Modulator Baseband	•	O	I	I	

Modulation (Digital Baseband FM) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
M-FSK Demodulator Baseband	•	I	O	O	
M-FSK Modulator Baseband	•	O	I	I	

Modulation (Digital Baseband PM) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
BPSK Demodulator Baseband	•	I	O	O	
BPSK Modulator Baseband	•	O	I	I	•
DBPSK Demodulator Baseband	•	I	O	O	
DBPSK Modulator Baseband	•	O	I	I	
DQPSK Demodulator Baseband	•	I	O	O	
DQPSK Modulator Baseband	•	O	I	I	
M-DPSK Demodulator Baseband	•	I	O	O	
M-DPSK Modulator Baseband	•	O	I	I	
M-PSK Demodulator Baseband	•	I	O	O	
M-PSK Modulator Baseband	•	O	I	I	•
OQPSK Demodulator Baseband	•	I	O	O	

Modulation (Digital Baseband PM) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
OQPSK Modulator Baseband	•	O	I	I	•
QPSK Demodulator Baseband	•	I	O	O	
QPSK Modulator Baseband	•	O	I	I	•

Modulation (Digital Baseband TCM) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
General TCM Decoder	•	I	O		
General TCM Encoder	•	O	I		
M-PSK TCM Decoder	•	I	O		
M-PSK TCM Encoder	•	O	I		
Rectangular QAM TCM Decoder	•	I	O		
Rectangular QAM TCM Encoder	•	O	I		

RF Impairments Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Free Space Path Loss	•	•			
I/Q Imbalance	•	•			
Memoryless Nonlinearity	•	•			
Phase Noise	•	•			

RF Impairments Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Phase/Frequency Offset	•	•			
Receiver Thermal Noise	•	•			

Sequence Operations Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Deinterlacer	•	•	•	•	•
Derepeat	•	•			
Descrambler	•		•	•	
Insert Zero	•	•	•	•	•
Interlacer	•	•	•	•	•
Puncture	•	•	•	•	•
Scrambler	•		•	•	

Source Coding Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
A-Law Compressor	•				
A-Law Expander	•				
Differential Decoder	•	•	•	•	
Differential Encoder	•	•	•	•	
Mu-Law Compressor	•				
Mu-Law Expander	•				

Source Coding Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Quantizing Decoder	•	•			
Quantizing Encoder	•	•			

Synchronization (Carrier Phase Recovery) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
CPM Phase Recovery	•	•			
M-PSK Phase Recovery	•	•			

Synchronization (Components) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Baseband PLL	•				
Charge Pump PLL	•				
Discrete-Time VCO	•	•			
Linearized Baseband PLL	•				
Phase-Locked Loop	•				
Continuous-Time VCO	•				

Synchronization (Timing Phase Recovery) Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Early-Late Gate Timing Recovery	•	•			
Gardner Timing Recovery	•	•			
MSK-Type Signal Timing Recovery	•	•			

Synchronization (Timing Phase Recovery) Library (Continued)

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Mueller-Muller Timing Recovery	•	•			
Squaring Timing Recovery	•	•			

Utility Library

Block	Double	Single	Boolean	Base Integer	Fixed-Point
Align Signals	•	•		• (signed integers only)	
Bipolar to Unipolar Converter	•	•	O	•	
Bit to Integer Converter	•	•	I	•	
Complex Phase Difference	•	•			
Complex Phase Shift	•	•			
Data Mapper	•	•		•	
dB Conversion	•	•			
Find Delay	•	•		• (signed integers only)	
Integer to Bit Converter	•	•	O	•	
Unipolar to Bipolar Converter	•	•	I	•	

4

Simulink and Real-Time Workshop Related Enhancements

This chapter gives an overview of Communications block enhancements related to Simulink and Real-Time Workshop®.

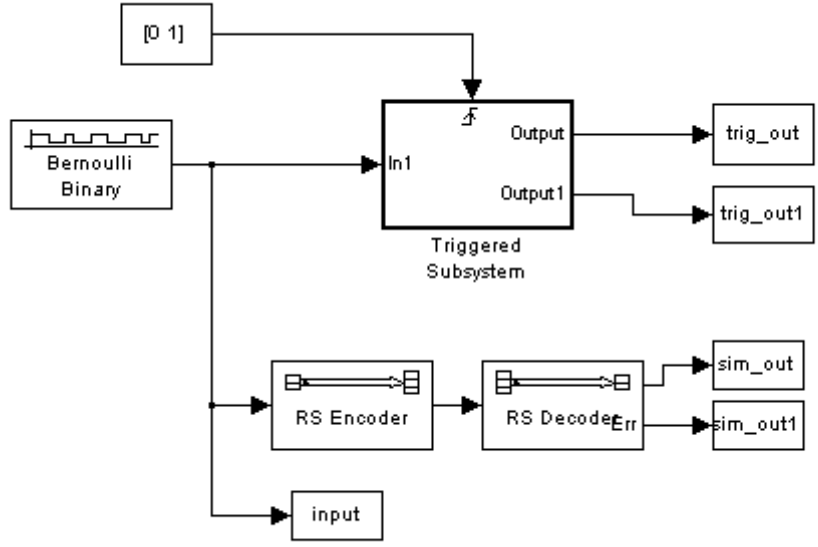
Communications Blocks in Triggered Subsystems

Many blocks in the Communications Blockset work within triggered subsystems. Refer to “Communications Blocks Enhancement Charts” on page 4-8 to see which blocks have this ability.

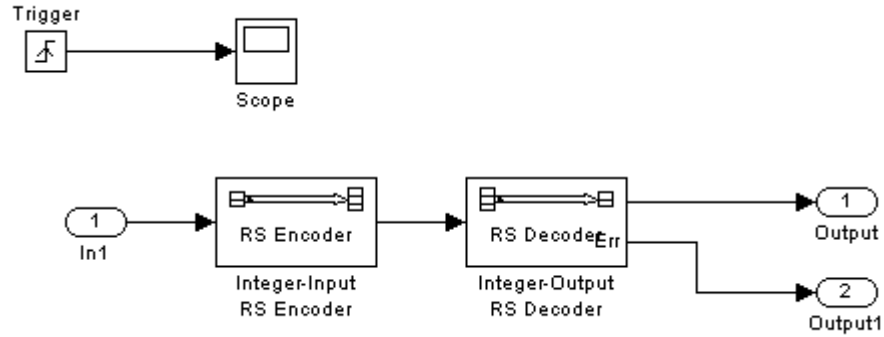
A triggered subsystem can be enabled or disabled with an input trigger. In the two examples that follow, the behavior of Communications blocks in triggered subsystems is explained.

Example 1: A Basic Triggered Subsystem

The first example is a nonsource triggered subsystem. This compares the outputs from RS Encoder and RS Decoder pairs when they are inside or outside a triggered subsystem.



Model Used for Output Comparison



Inside the Triggered Subsystem

The source is a Bernoulli Binary Generator block with frame-based input, three samples per frame, and a sample time of `0.1`. The triggered signal is `[0 1]` with a sample time of `0.3`.

The RS Encoder and RS Decoder pair within the triggered subsystem is enabled only when there is a rising signal in the trigger input. When the trigger receives a falling signal, any processing in the subsystem is disabled, and the subsystem outputs only what it has stored in memory.

After running the simulation, compare the input signal (`input`), the output from the RS Encoder and RS Decoder pair (`sim_out`), and the output from the triggered subsystem (`trig_out`):

```
>> [input  sim_out  trig_out]

ans =

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

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

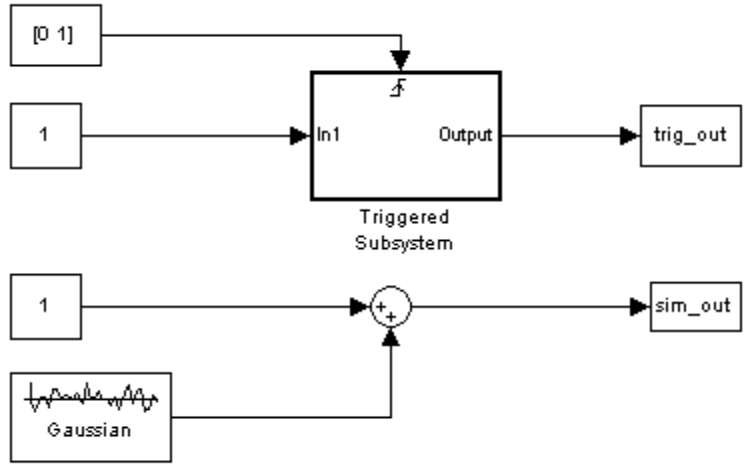
`sim_out` directly reproduces `input`, as expected from an RS Encoder and RS Decoder pair.

`trig_out`, on the other hand, reproduces `input` only during the 4th, 5th, 6th...10th, 11th, and 12th samples, and so on, in triplets. According to the simulation parameters, the trigger was toggling after every third sample of the input. The mirroring behavior was therefore also being toggled after every third sample.

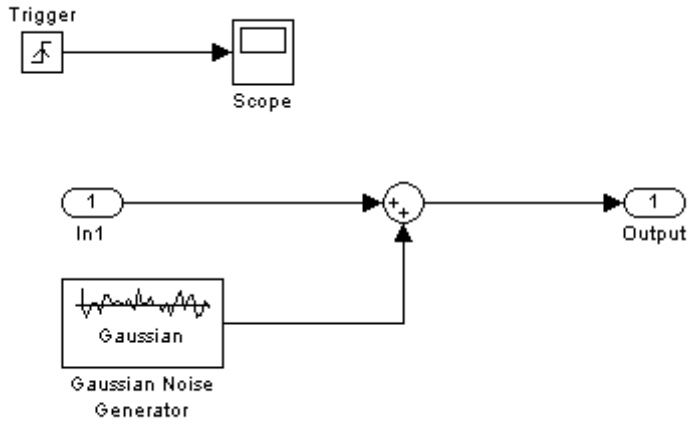
On the falling trigger, the subsystem turns off and repeats its previous three outputs, as can be seen in the 7th, 8th, 9th...13th, 14th, and 15th samples, and so on. The first three outputs of `trig_out` are all `0` because, while the trigger is off, the subsystem did not have any previous outputs in memory.

Example 2: Importance of the Block Location

This is an example of a triggered subsystem that is also a source. The key point in this model is that everything within the triggered subsystem is enabled or disabled, including any source blocks.



Model Used for Output Comparison



Inside the Triggered Subsystem

The source is a Gaussian Noise Generator with sample-based input and a sample time of 1. The triggered signal is `[0 1]` with a sample time of `1`. The DSP constant blocks serve as driver blocks in this model.

In the triggered subsystem, the sample time for the Gaussian Noise Generator is `-1`.

After running the simulation, compare the outputs from the nontriggered system (`sim_out`) and the triggered subsystem (`trig_out`).

```
>> [sim_out trig_out]

ans =

    0.7311         0
    1.6390    0.7311
    0.0091    0.7311
    1.7771    1.6390
    1.6605    1.6390
    0.2890    0.0091
    0.5074    0.0091
    3.1751    1.7771
```

```
    1.6019     1.7771
    0.0438     1.6605
   -0.7200     1.6605
```

In this case, the output of the triggered subsystem is not just switching its output on and off, but seems to be falling behind the output of the nontriggered system. This is because the Gaussian Noise Generator contained in the subsystem is also being toggled on and off with the trigger. As a result, its preset sequence of numbers is not being used up at the same rate as its counterpart outside the subsystem.

Communications Blocks Enhancement Charts

The following tables show the enhancements made to the Communications blocks.

The “Triggered Subsystem” column shows which blocks work within a triggered subsystem. A dot (•) in this column indicates that the block works.

The “Embeddable RTW C-Code” column shows the level of Real-Time Workshop support as described below:

- Blocks that have been inlined, i.e., S-functions that have TLC, generate embeddable Real-Time Workshop C-code, and participate in the Simulink and Real-Time Workshop optimizations. These blocks have a dot (•) in this column.
- Blocks that are subsystems built of Simulink or Signal Processing Blockset blocks generate embeddable Real-Time Workshop C-code. These might not, however, participate in all Simulink and Real-Time Workshop optimizations, because this is dependent on the underlying blocks. These blocks have an asterisk (*) in this column.
- Blocks that do not support integer-only code generation but otherwise generate embeddable Real-Time Workshop C-code have a minus sign (-) in this column.
- There are blocks that are compatible with Real-Time Workshop but might not be able to generate embeddable C-code, as they have not yet been inlined. These still work with GRT and ERT targets when the **non-inlined S-Functions** check box is selected, but the generated code or executable might not work on the MathWorks supported boards. These blocks have a plus sign (+) in this column.
- Blocks for which code generation might not be applicable, as they are analog and/or instrumentation blocks, have an N/A in this column.

Channels Library

Block	Triggered Subsystem	Embeddable RTW C-Code
AWGN Channel	•	•
Binary Symmetric Channel	•	*
Multipath Rayleigh Fading Channel		•
Multipath Rician Fading Channel		+

Communications Filters Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Gaussian Filter	•	*
Ideal Rectangular Pulse Filter	•	*
Integrate and Dump	• (single rate operation)	•
Raised Cosine Receive Filter	•	*
Raised Cosine Transmit Filter	•	*
Windowed Integrator	•	*

Communications Sinks Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Discrete-Time Eye Diagram Scope		N/A
Discrete-Time Scatter Plot Scope		N/A

Communications Sinks Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Discrete-Time Signal Trajectory Scope		N/A
Error Rate Calculation	•	•

Communications Sources (Noise Generators) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Gaussian Noise Generator	•	*
Rayleigh Noise Generator	•	*
Rician Noise Generator	•	*
Uniform Noise Generator	•	*

Communications Sources (Random Data Sources) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Bernoulli Binary Generator	•	*
Poisson Integer Generator	•	•
Random Integer Generator	•	*

Communications Sources (Sequence Generators) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Barker Code Generator	•	*, -
Gold Sequence Generator	•	•
Hadamard Code Generator	•	*, -

Communications Sources (Sequence Generators) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Kasami Sequence Generator	•	•
OVSF Code Generator	•	*, -
PN Sequence Generator	•	•
Walsh Code Generator	•	*, -

Equalizers Library

Block	Triggered Subsystem	Embeddable RTW C-Code
CMA Equalizer	•	*
LMS Decision Feedback Equalizer	•	*
LMS Linear Equalizer	•	*
MLSE Equalizer	•	•
Normalized LMS Decision Feedback Equalizer	•	*
Normalized LMS Linear Equalizer	•	*
RLS Decision Feedback Equalizer	•	*
RLS Linear Equalizer	•	*
Sign LMS Decision Feedback Equalizer	•	*
Sign LMS Linear Equalizer	•	*

Equalizers Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Variable Step LMS Decision Feedback Equalizer	•	*
Variable Step LMS Linear Equalizer	•	*

Error Detection and Correction (Block) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
BCH Decoder	•	•
BCH Encoder	•	*
Binary Cyclic Decoder	•	*
Binary Cyclic Encoder	•	*
Binary Linear Decoder	•	*
Binary Linear Encoder	•	*
Binary-Input RS Encoder	•	•
Binary-Output RS Decoder	•	•
Hamming Decoder	•	*
Hamming Encoder	•	*
Integer-Input RS Encoder	•	•
Integer-Output RS Decoder	•	•

Error Detection and Correction (Convolutional) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
APP Decoder	•	•

Error Detection and Correction (Convolutional) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Convolutional Encoder	•	•
Viterbi Decoder	•	•

Error Detection and Correction (CRC) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
CRC-N Generator	•	•
CRC-N Syndrome Detector	•	•
General CRC Generator	•	•
General CRC Syndrome Detector	•	•

Interleaving (Block) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Algebraic Deinterleaver	•	•
Algebraic Interleaver	•	•
General Block Deinterleaver	•	•
General Block Interleaver	•	•
Matrix Deinterleaver	•	•
Matrix Helical Scan Deinterleaver	•	•
Matrix Helical Scan Interleaver	•	•
Matrix Interleaver	•	•

Interleaving (Block) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Random Deinterleaver	•	•
Random Interleaver	•	•

Interleaving (Convolutional) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Convolutional Deinterleaver	•	•
Convolutional Interleaver	•	•
General Multiplexed Deinterleaver	•	•
General Multiplexed Interleaver	•	•
Helical Deinterleaver	•	•
Helical Interleaver	•	•

Modulation (Analog Passband) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
DSB AM Demodulator Passband	•	+
DSB AM Modulator Passband	•	+
DSBSC AM Demodulator Passband	•	+
DSBSC AM Modulator Passband	•	+
FM Demodulator Passband	•	+
FM Modulator Passband	•	+
PM Demodulator Passband	•	+
PM Modulator Passband	•	+

Modulation (Analog Passband) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
SSB AM Demodulator Passband	•	+
SSB AM Modulator Passband	•	+

Modulation (Digital Baseband AM) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
General QAM Demodulator Baseband	•	•
General QAM Modulator Baseband	•	•
M-PAM Demodulator Baseband	•	•
M-PAM Modulator Baseband	•	•
Rectangular QAM Demodulator Baseband	•	•
Rectangular QAM Modulator Baseband	•	•

Modulation (Digital Baseband CPM) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
CPFSK Demodulator Baseband	• (for frame-based operation only)	•
CPFSK Modulator Baseband	• (for frame-based operation only)	•
CPM Demodulator Baseband	• (for frame-based operation only)	•

Modulation (Digital Baseband CPM) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
CPM Modulator Baseband	• (for frame-based operation only)	•
GMSK Demodulator Baseband	• (for frame-based operation only)	•
GMSK Modulator Baseband	• (for frame-based operation only)	•
MSK Demodulator Baseband	• (for frame-based operation only)	•
MSK Modulator Baseband	• (for frame-based operation only)	•

Modulation (Digital Baseband FM) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
M-FSK Demodulator Baseband		•
M-FSK Modulator Baseband		•

Modulation (Digital Baseband PM) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
BPSK Demodulator Baseband	•	•
BPSK Modulator Baseband	•	•

Modulation (Digital Baseband PM) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
DBPSK Demodulator Baseband	•	•
DBPSK Modulator Baseband	•	•
DQPSK Demodulator Baseband	•	•
DQPSK Modulator Baseband	•	•
M-DPSK Demodulator Baseband	•	•
M-DPSK Modulator Baseband	•	•
M-PSK Demodulator Baseband	•	•
M-PSK Modulator Baseband	•	•
OQPSK Demodulator Baseband	• (for frame-based operation only)	•
OQPSK Modulator Baseband	• (for frame-based operation only)	•
QPSK Demodulator Baseband	•	•
QPSK Modulator Baseband	•	•

Modulation (Digital Baseband TCM) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
General TCM Decoder	•	•
General TCM Encoder	•	*
M-PSK TCM Decoder	•	•
M-PSK TCM Encoder	•	*

Modulation (Digital Baseband TCM) Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Rectangular QAM TCM Decoder	•	•
Rectangular QAM TCM Encoder	•	*

RF Impairments Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Free Space Path Loss	•	*
I/Q Imbalance	•	*
Memoryless Nonlinearity	•	*
Phase Noise	•	*
Phase/Frequency Offset	•	*
Receiver Thermal Noise		*

Sequence Operations Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Deinterlacer	•	•
Derepeat	• (single rate operation)	•
Descrambler	•	•
Insert Zero	•	•
Interlacer	•	•

Sequence Operations Library (Continued)

Block	Triggered Subsystem	Embeddable RTW C-Code
Puncture	•	•
Scrambler	•	•

Source Coding Library

Block	Triggered Subsystem	Embeddable RTW C-Code
A-Law Compressor	•	+
A-Law Expander	•	+
Differential Decoder	•	+
Differential Encoder	•	+
Mu-Law Compressor	•	+
Mu-Law Expander	•	+
Quantizing Decoder	•	+
Quantizing Encoder	•	+

Synchronization (Carrier Phase Recovery) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
CPM Phase Recovery		*
M-PSK Phase Recovery		*

Synchronization (Components) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Baseband PLL		+
Charge Pump PLL		+
Discrete-Time VCO		+
Linearized Baseband PLL		+
Phase-Locked Loop		+
Continuous-Time VCO		N/A

Synchronization (Timing Phase Recovery) Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Early-Late Gate Timing Recovery	• (for frame-based operation only)	•
Gardner Timing Recovery	• (for frame-based operation only)	•
MSK-Type Signal Timing Recovery	• (for frame-based operation only)	•
Mueller-Muller Timing Recovery	• (for frame-based operation only)	•
Squaring Timing Recovery	• (for frame-based operation only)	*

Utility Blocks Library

Block	Triggered Subsystem	Embeddable RTW C-Code
Align Signals		*
Bipolar to Unipolar Converter	•	*
Bit to Integer Converter	•	•
Complex Phase Difference	•	*
Complex Phase Shift	•	*
Data Mapper	•	*
dB Conversion	•	*
Find Delay		*
Integer to Bit Converter	•	•
Unipolar to Bipolar Converter	•	*

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V