

Communications Blockset

For Use with Simulink®

Modeling
|

Simulation
|

Implementation
|

How to Contact The MathWorks:



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup



support@mathworks.com 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. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

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

Using the Communications Blockset

© COPYRIGHT 2001-2002 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: June 2001	Online only	New for Version 2.0.1 (Release 12.1)
July 2002	Online only	Revised for Version 2.5 (Release 13)

Using the Libraries

1

Signal Support	1-3
Signal Terminology	1-3
Processing Matrices, Vectors, and Scalars	1-4
Processing Frame-Based and Sample-Based Signals	1-6
Communications Sources	1-7
Controlled Sources	1-7
Random Data Sources	1-8
Random Noise Generators	1-9
Sequence Generators	1-10
Sequence Generator Examples	1-12
Block Parameters	1-18
Communications Sinks	1-23
Sink Features of the Blockset	1-23
Writing to a File	1-23
Error Statistics	1-23
Scopes	1-24
Example: Viewing a Sinusoid	1-26
Source Coding	1-30
Source Coding Features of the Blockset	1-30
Representing Quantization Parameters	1-31
Quantizing a Signal	1-31
Implementing Differential Pulse Code Modulation	1-35
Companding a Signal	1-38
Selected Bibliography for Source Coding	1-40
Block Coding	1-41
Organization of This Section	1-41
Accessing Block Coding Blocks	1-41
Block Coding Features of the Blockset	1-42
Communications Toolbox Support Functions	1-43
Channel Coding Terminology	1-43

Data Formats for Block Coding	1-43
Using Block Encoders and Decoders Within a Model	1-46
Examples of Block Coding	1-46
Notes on Specific Block Coding Techniques	1-49
Selected Bibliography for Block Coding	1-53
Convolutional Coding	1-54
Organization of This Section	1-54
Accessing Convolutional Coding Blocks	1-54
Convolutional Coding Features of the Blockset	1-55
Parameters for Convolutional Coding	1-55
Example: A Rate 2/3 Feedforward Encoder	1-56
Implementing a Systematic Encoder with Feedback	1-59
Example: Soft-Decision Decoding	1-60
Selected Bibliography for Convolutional Coding	1-67
Cyclic Redundancy Check Coding	1-69
Organization of this Section	1-69
Accessing CRC Blocks	1-69
CRC Coding Features of the Blockset	1-69
CRC Algorithm	1-70
Interleaving	1-72
Interleaving Features of the Blockset	1-72
Block Interleavers	1-72
Convolutional Interleavers	1-75
Selected Bibliography for Interleaving	1-80
Analog Modulation	1-81
Accessing Analog Modulation Blocks	1-81
Analog Modulation Features of the Blockset	1-81
Baseband Modulated Signals Defined	1-82
Representing Signals for Analog Modulation	1-83
Timing Issues in Analog Modulation	1-83
Filter Design Issues	1-87
Digital Modulation	1-92
Accessing Digital Modulation Blocks	1-92
Digital Modulation Features of the Blockset	1-93

Representing Signals for Digital Modulation	1-96
Delays in Digital Modulation	1-97
Upsampled Signals and Rate Changes	1-101
Examples of Digital Modulation	1-104
Selected Bibliography for Digital Modulation	1-112
Channels	1-113
Channel Features of the Blockset	1-113
AWGN Channel	1-113
Fading Channels	1-114
Binary Symmetric Channel	1-117
Selected Bibliography for Channels	1-118
RF Impairments	1-119
Types of RF Impairments the Blocks Model	1-119
Scatter Plot Examples	1-120
Example Using the RF Impairments Library Blocks	1-127
Synchronization	1-130
Synchronization Features of the Blockset	1-130
Overview of PLL Simulation	1-131
Implementing an Analog Baseband PLL	1-131
Implementing a Digital PLL	1-132
Selected Bibliography for Synchronization	1-132

Modeling Communication Systems

2

Computing Delays	2-3
Other References for Delays	2-3
Sources of Delays	2-4
ADSL Demo Model	2-4
Punctured Coding Model	2-9
Manipulating Delays	2-14
Delays and Alignment Problems	2-14
Aligning Words of a Block Code	2-17

Aligning Words for Interleaving	2-19
Aligning Words of a Concatenated Code	2-21
Comparing Baseband and Passband Simulation	2-24
Running a Passband Simulation	2-24
Running an Equivalent Baseband Simulation	2-25
Generating Error Curves	2-26
Speed of Baseband Versus Passband Models	2-28
Comparing Baseband and Passband Signals	2-30
Troubleshooting a Passband Simulation	2-32

Demonstration Models

3

Punctured Convolutional Coding Demo	3-2
Structure of the Demo	3-2
Generating Random Data	3-3
Convolutional Encoding	3-3
Puncturing	3-4
Transmitting Data	3-4
Demodulating	3-5
Inserting Zeros	3-5
Viterbi Decoding	3-6
Calculating the Error Rate	3-6
Evaluating Results	3-6
Bibliography	3-8
Adaptive Equalization Demo	3-9
CPM Phase Tree Demo	3-11
Structure of the demo	3-11
Variables	3-11
Visible Results of the Demo	3-12
Experimenting with the Demo	3-12
GMSK vs. MSK Demo	3-14
Structure of the Demo	3-14

Visible Results of the Demo	3-14
Filtered QPSK vs. MSK Demo	3-16
Structure of the Demo	3-16
Visible Results of the Demo	3-16
Rayleigh Fading Channel Demo	3-17
Structure of the Demo	3-17
Visible Results	3-17
Gray Coded 8-PSK Demo	3-18
How the Model Executes	3-19
Variables in the Model	3-19
Components of the Gray Coding Demo	3-20
Learning More About the Gray Coding Demo	3-26
Discrete Multitone Signaling Demo	3-29
Structure of the Demo	3-29
Discrete Multitone Signaling Demo, Alternative Form	3-30
Selected Bibliography	3-30
Iterative Decoding of a Serially Concatenated	
Convolutional Code (SCCC) - Demo	3-31
Structure of the Demo	3-31
Creating a Serially Concatenated Code	3-32
Decoding Using an Iterative Process	3-33
Visible Results of the Demo	3-34
Selected Bibliography	3-34
Phase Noise Effects in 256-QAM - Demo	3-36
Structure of the Demo	3-36
Visible Results of the Demo	3-36
PLL-Based Frequency Synthesis Demo	3-38
Variables in the Model	3-38
Running the Simulation	3-39
Blocks in the Model	3-40
Pulse Generator	3-40
Divide Frequency by M	3-41

Phase Detector	3-42
Analog Filter Design	3-42
Gain Block	3-43
Voltage-Controlled Oscillator	3-43
Simulation Parameters	3-44
Fractional-N Frequency Synthesis Demo	3-46
Variables in the Model	3-47
Blocks and Subsystems in the Model	3-47
Phase Detector	3-48
Running a Simulation	3-48
Reference	3-49
256-Channel ADSL Demo	3-50
Structure of the Demo	3-50
Transmitting Data	3-50
Processing Received Data	3-51
Displaying Error Statistics	3-51
Selected Bibliography	3-52
Bluetooth Voice Transmission Demo	3-53
Structure of the Demo	3-53
Mask Variables	3-54
Results and Display	3-54
Reference	3-55
Digital Video Broadcasting Demo	3-56
Structure of the Demo	3-56
Variables in the Demo	3-57
Design of the Receiver	3-57
Visible Results of the Demo	3-58
Selected Bibliography	3-58
HiperLAN/2 Demo	3-59
Structure of the demo	3-59
Visible Results and Display	3-60
References	3-60
RF Satellite Link Demo	3-61

Structure of the demo	3-61
Mask Parameters	3-63
Results and Displays	3-66
Experimenting with the Demo	3-66
Selected Bibliography	3-69
WCDMA Coding and Multiplexing Demo	3-70
WCDMA End-to-End Physical Layer Demo	3-71
Overall Structure of the Physical Layer	3-71
Parameters in the Demo	3-74
Visible Results of the Demo	3-77
References	3-78
WCDMA Spreading and Modulation Demo	3-79

Using the Libraries

Signal Support	1-3
Communications Sources	1-7
Communications Sinks	1-23
Source Coding	1-30
Block Coding	1-41
Convolutional Coding	1-54
Cyclic Redundancy Check Coding	1-69
Interleaving	1-72
Analog Modulation	1-81
Digital Modulation	1-92
Channels	1-113
RF Impairments	1-119
Synchronization	1-130

This chapter describes and illustrates how to model communication techniques using the blocks in the Communications Blockset. The first section, “Signal Support,” discusses the types of signals that this blockset supports. Each subsequent section corresponds to one of the core libraries within the Communications Blockset. These sections are:

- “Communications Sources” on page 1-7
- “Communications Sinks” on page 1-23
- “Source Coding” on page 1-30
- “Block Coding” on page 1-41
- “Convolutional Coding” on page 1-54
- “Cyclic Redundancy Check Coding” on page 1-69
- “Interleaving” on page 1-72
- “Analog Modulation” on page 1-81
- “Digital Modulation” on page 1-92
- “Channels” on page 1-113
- “RF Impairments” on page 1-119
- “Synchronization” on page 1-130

For descriptions of individual blocks, see their reference entries. For background or theoretical information about communications techniques, see the works listed in the “Selected Bibliography...” sections that appear in this chapter.

Signal Support

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

- “Signal Terminology”
- “Processing Matrices, Vectors, and Scalars” on page 1-4
- “Processing Frame-Based and Sample-Based Signals” on page 1-6

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 code word, 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, then the block preserves dimension.
- If a block can process sample-based vectors, then
 - 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

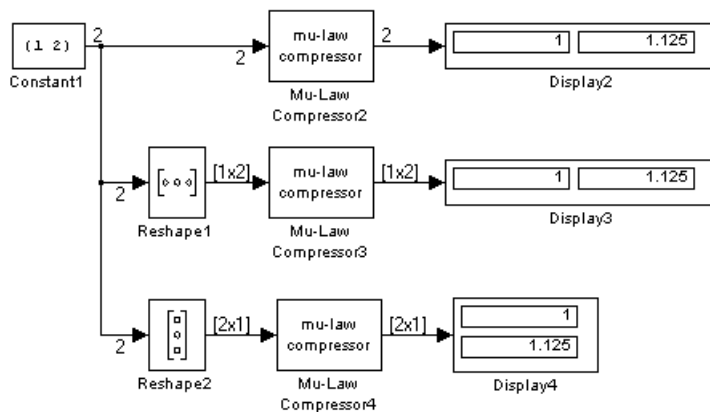
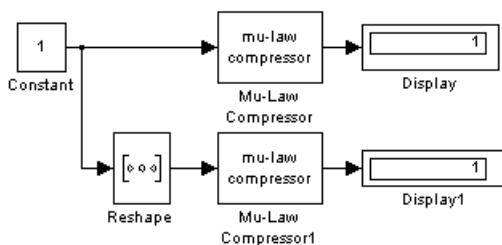
channels. The block does *not* assume that the elements of the input vector are successive samples from a single time series.

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 1-6.

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 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 DSP 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 would make 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 Simulink's Sources library, in the DSP Blockset's DSP Sources library, and in the Communication Blockset's Comm Sources library.

You can open the Comm Sources library by double-clicking its icon in the main Communications Blockset library (`commLib`), or by typing

```
commsource2
```

at the MATLAB prompt.

Blocks in the Comm Sources library can

- Generate controlled sources by reading from a file or by simulating a voltage-controlled oscillator (VCO)
- 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.

Controlled Sources

Blocks in the Controlled Sources sublibrary of the Comm Sources library simulate nonrandom signals by reading from a file or by simulating a voltage-controlled oscillator (VCO):

- The Triggered Read from File block reads a record from a file whenever an input trigger signal has a rising edge. You can set up the block to read at every rising edge of the trigger, or every k th rising edge of the trigger for a positive number k .
- A voltage-controlled oscillator is one part of a phase-locked loop. The Voltage-Controlled Oscillator 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.

You can open the Controlled Sources sublibrary by double-clicking its icon in the main Communications Blockset library (`comm1ib`), or by typing

```
commcontsracs2p1
```

at the MATLAB prompt.

Random Data Sources

Blocks in the Data Sources sublibrary of the Comm Sources library generate random data to simulate signal sources. You can use blocks in the 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 Data Sources sublibrary by double-clicking its icon in the main Communications Blockset library (`comm1ib`), or by typing

```
commrandsracs2p1
```

at the MATLAB prompt.

Random Bits

The Bernoulli Binary Generator and Binary Error Pattern Generator blocks both generate random bits, but differ in the way that you specify the distribution of 1s. As a result, the Bernoulli Binary Generator block is suitable for representing sources, while the Binary Error Pattern Generator block is more appropriate for modeling channel errors.

The Bernoulli Binary Generator block considers each element of the signal to be an independent Bernoulli random variable. Also, different elements need not be identically distributed.

The Binary Error Pattern Generator block constructs a random binary signal using a two-stage process. First, using information that you provide in the block mask, it determines how many 1s will appear. Then it determines where to place the required number of 1s, so that each possible arrangement has equal probability.

For example, if you set the **Binary vector length** parameter to 4, set the **Probabilities** parameter to 1, and clear the **Frame-based outputs** check box, then the block generates binary vectors of length 4, each of which contains exactly one 1. You might use these parameters to perturb a binary code that consists of 4-bit codewords. Adding the random vector to your code vector (modulo 2) would introduce exactly one error into each codeword. Alternatively, to perturb each codeword by introducing one error with probability 0.4 and two errors with probability 0.6, set the **Probabilities** parameter to [0.4, 0.6] instead of 1.

Note that the **Probabilities** parameter of the Binary Error Pattern Generator block affects only the *number* of 1s in each vector, not their placement.

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 (comm1ib), or by typing

```
commnoisgen2p1
```

at the MATLAB prompt.

Sequence Generators

You can use blocks in the Sequence Generators sublibrary of the Comms 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 (comm1ib), or by typing

```
commseqgen2p1
```

at the MATLAB prompt.

Blocks in the Sequence Generators sublibrary generate

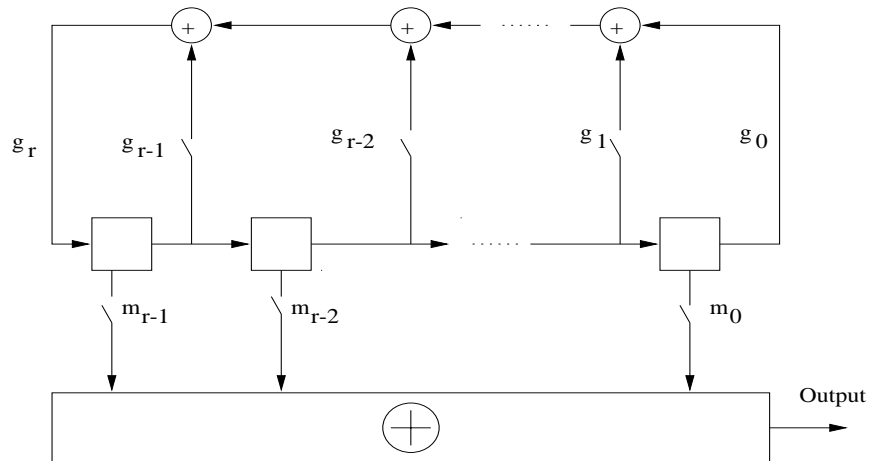
- Pseudorandom sequences
- Orthogonal codes
- Synchronization codes

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 , $g_r z^r + g_{r-1} z^{r-1} + \dots + g_0$. The coefficient g_i is 1 if there is a connection from the i th 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 d th arrow in the lower half of the diagram.

See “Example: Pseudorandom Sequences” on page 1-12 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 “Example: Orthogonal Sequences” on page 1-16 for an example that uses these blocks.

Sequence Generator Examples

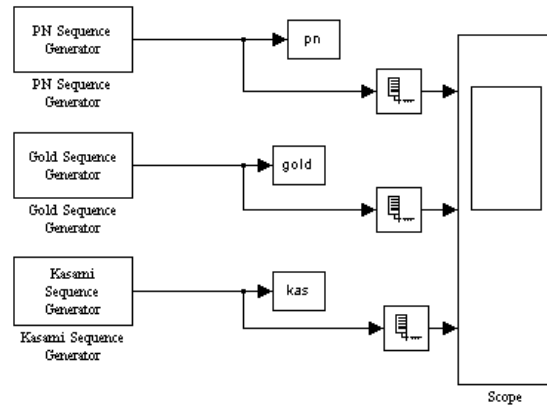
This section presents two example models that illustrate the blocks in the Sequence Generator library.

Example: Pseudorandom Sequences

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

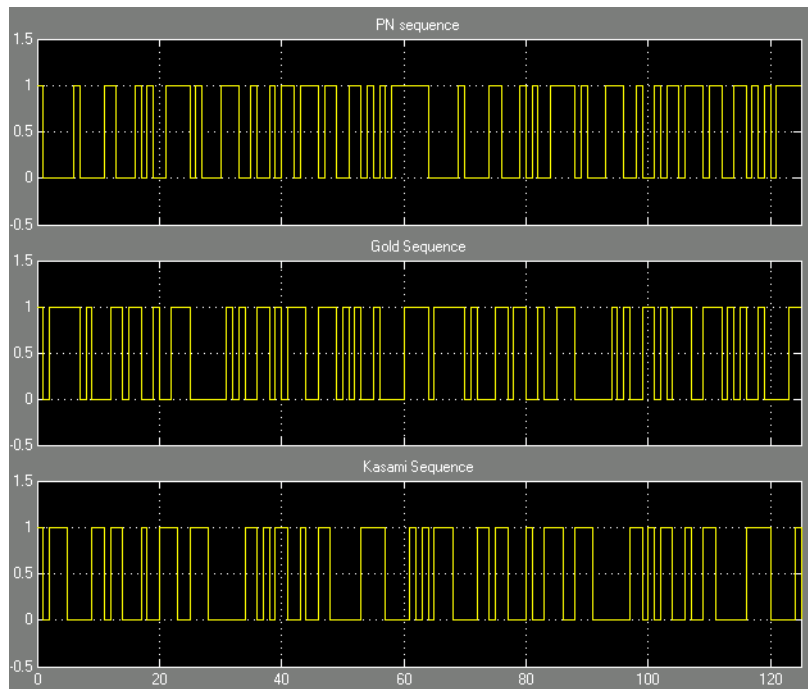
- PN Sequence Generator
- Gold Sequence Generator
- Kasami Sequence Generator

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 the three signals:



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 by entering the following code 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
```

The 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 three values for the Gold sequence are

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

The three 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.

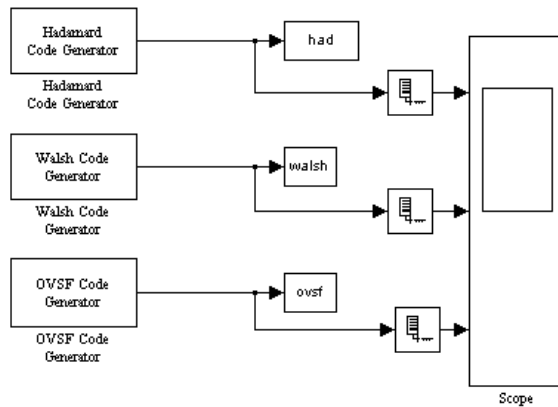
Also note that the peak value of `corrvals` for the Kasami sequence is less than the peak value for the Gold sequence. In fact, the small set of Kasami sequences satisfy the lower bounds for correlation values, and for this reason they are also referred to as optimal sequences.

Example: Orthogonal Sequences

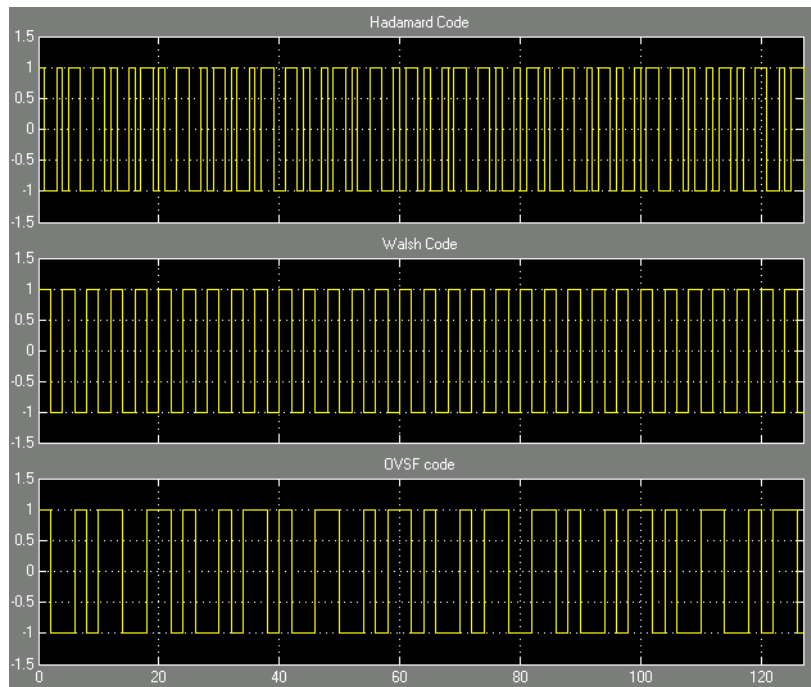
This example demonstrates the orthogonality of pairs of sequences generated using different **Code index** parameters, for each of the following three blocks:

- 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 code 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 code then calculates, for each of the three blocks, the cross correlation between the sequence generated by the first run with 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 twice
```

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

the **Sample time** parameter is the time interval between successive rows of the frame-based matrix.

If you use a Simulink Signal Inspection block to query the period of a frame-based output from a random source block in the Comm Sources library, then note that the Signal Inspection 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.

$$A \text{ seconds/frame} = (B \text{ seconds/sample}) * (S \text{ samples/frame})$$

where

- A is the number shown in the Signal Inspection 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. You should make sure that initial seeds in different blocks in a model have different values, so that they generate statistically independent sequences.

Four of the 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:

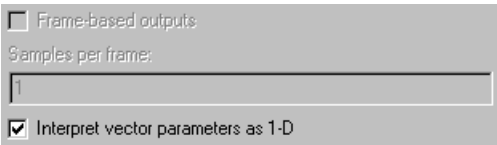
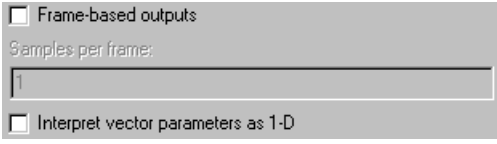
- Binary Error Pattern Generator
- Gaussian Noise Generator
- Rayleigh Noise Generator
- Rician Noise Generator

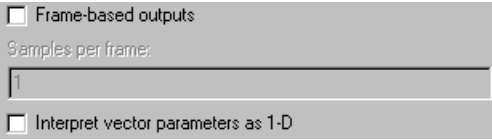

To avoid having to remember whether a block that you are using is on this list, you can simply apply the seed rule to all source blocks that have an **Initial seed** parameter.

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 will be different from that produced with a different constant seed. If you want the noise to be different each time you start the simulation, then you can use a varying seed such as `randseed(cputime)`.

Signal Attribute Parameters for Random Sources

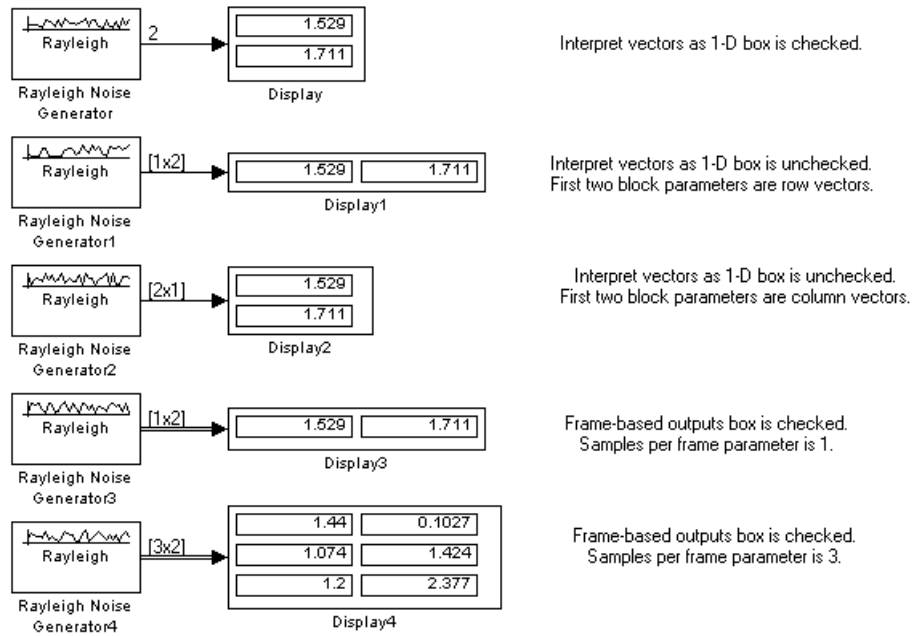
In most random source blocks, the output can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. The following table indicates how to set certain block parameters depending on the kind of signal you want to generate.

Signal Attributes	Parameter Settings
Sample-based, one-dimensional	
Sample-based row vector	 <p data-bbox="733 1251 1322 1315">Also, any vector parameters in the block should be rows, not columns.</p>

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

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. Notice how 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 individual entry in the reference section 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 (commlib), or by typing

```
commsink2
```

at the MATLAB prompt.

Sink Features of the Blockset

Blocks in this library can

- Write to a file when trigger events occur
- Compute error statistics
- Plot an eye diagram
- Generate a scatter diagram
- Plot a signal trajectory

This section describes these capabilities. Other sinks are in Simulink's Sinks library and in the DSP Blockset's DSP Sinks library.

Writing to a File

The Triggered Write to File block writes data to a file whenever an input trigger signal has a rising edge. You can set up the block to write at every rising edge of the trigger, or at every k th rising edge of the trigger for a positive number k . The data can have an ASCII, integer, or floating-point format. If the destination file already exists, then this block overwrites it. For more details, see the reference page for the Triggered Write to File block.

For untriggered writing of MAT files, use Simulink's To File block.

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. Also, if you use frame-based data, then you can have the block consider certain samples and ignore others.

The example in the section “Example: Soft-Decision Decoding” on page 1-60 illustrates the use of the Error Rate Calculation block.

Scopes

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

- Eye Diagrams
- Scatter Plots
- Signal Trajectories

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

Block Name	Plots
Continuous-Time Eye and Scatter Diagrams	Eye diagram, scatter plot, or signal trajectory of a continuous signal
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. Thus 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 following two blocks produce eye diagrams:

- Continuous-Time Eye and Scatter Diagrams, with **Diagram type** set to **Eye Diagram**
- Discrete-Time Eye Diagram Scope

The first processes continuous-time signals, while the second processes discrete-time signals. The blocks also differ in the way you determine the decision timing: the Continuous-Time Eye and Scatter Diagrams block draws a vertical line to indicate a decision every time a trigger signal has a rising edge, whereas the Discrete-Time Eye Diagram Scope block draws a similar line periodically according to a mask parameter.

An example appears in “Example: Viewing a Sinusoid” on page 1-26.

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 following two blocks produce scatter plots:

- Continuous-Time Eye and Scatter Diagrams, with **Diagram type** set to **Scatter Diagram**
- Discrete-Time Scatter Plot Scope

Both the Continuous-Time Eye and Scatter Diagrams block and the Discrete-Time Scatter Plot Scope block produce scatter plots. The first processes continuous-time signals, while the second processes 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.

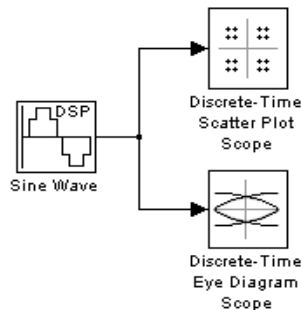
The following two blocks produce signal trajectories:

- Continuous-Time Eye and Scatter Diagrams, with **Diagram type** set to **X-Y Diagram**
- Discrete-Time Signal Trajectory Scope

The Discrete-Time Scatter Plot Scope displays points on the trajectory at discrete time intervals, corresponding to the decision points, while the Discrete-Time Scatter Plot 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.

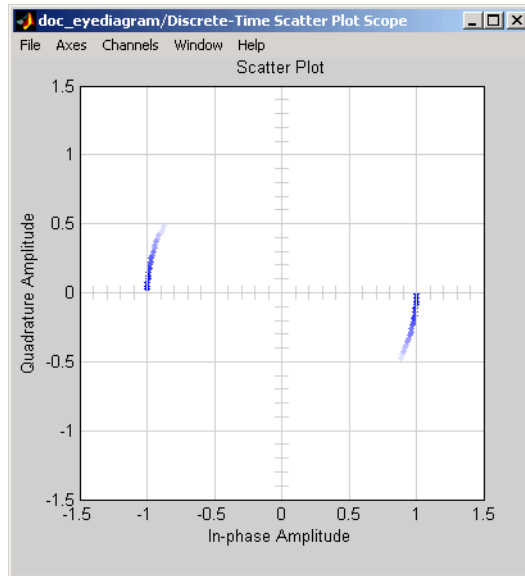


To open the completed model, click [here](#) in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Sine Wave, in the DSP Blockset DSP Sources library (*not* the Sine Wave block in the Simulink Sources library)

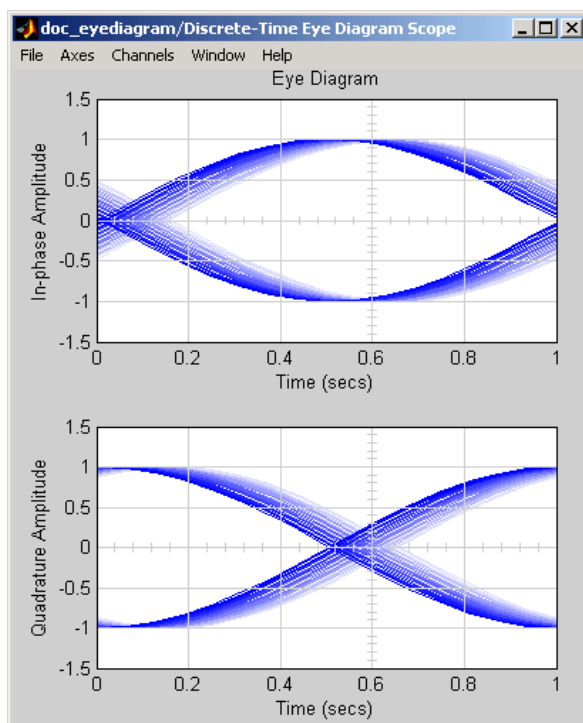
- Set **Frequency** to .502.
- Set **Output complexity** to **Complex**.
- Set **Sample time** to 1/16.
- Discrete-Time Scatter Plot Scope, in the Comm Sinks library
 - Check the box next to **Show Plotting Properties**.
 - Set **Samples per symbol** to 16.
 - Check the box next to **Show Figure Properties**.
 - Set **Scope position** to figposition([2.5 55 35 35]);.
 - Set **Figure name** to Scatter Plot.
- Discrete-Time Eye Diagram Scope, in the Comm Sinks library
 - Check the box next to **Show Plotting Properties**.
 - Set **Samples per symbol** to 16.
 - Check the box next to **Show Figure Properties**.
 - Set **Scope position** to figposition([42.5 55 35 35]);.
 - Set **Figure name** to Eye Diagram.

Connect the blocks as shown in the preceding figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation 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 in the **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.

Source Coding

Source coding, also known as *quantization* or *signal formatting*, is a way of processing data in order 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 1-40.

Source Coding Features of the Blockset

This blockset supports scalar quantization, predictive 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 (commlib), or by typing

```
commsrccod2
```

at the MATLAB prompt.

Blocks in the Source Coding library can

- Use a partition and codebook to quantize a signal
- Implement differential pulse code modulation (DPCM)
- Comband a signal using a μ -law or A-law compressor or expander
- Encode or decode a signal using differential coding

Supporting functions in the Communications Toolbox also allow you to optimize source coding parameters for a set of training data. See the sections “Optimizing Quantization Parameters” and “Optimizing DPCM 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

- $\{x: x \leq 0\}$
- $\{x: 0 < x \leq 1\}$
- $\{x: 1 < x \leq 3\}$
- $\{x: 3 < x\}$

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

$[0, 1, 3]$

Notice that 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 $[0, 1, 3]$.

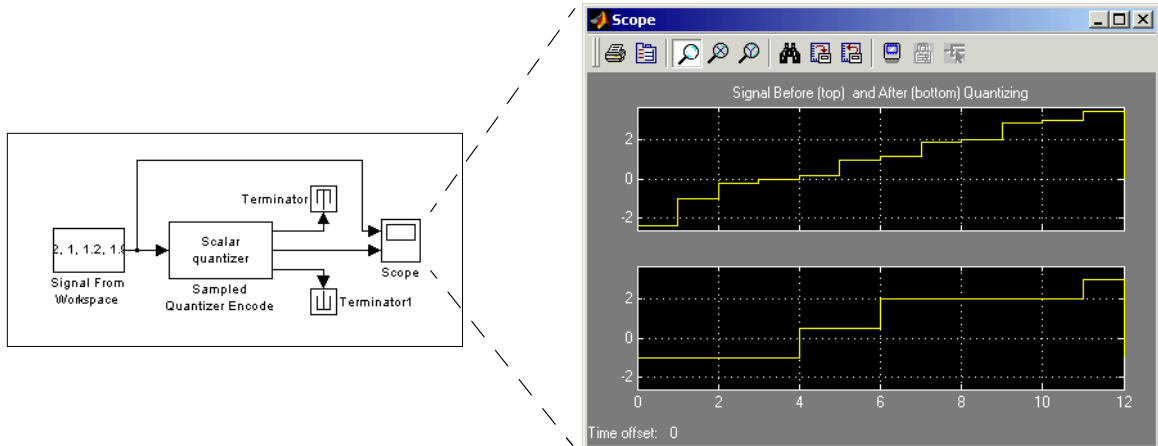
Quantizing a Signal

This section shows how the Sampled Quantizer Encode, Enabled Quantizer Encode, and Quantizer Decode blocks use the partition and codebook parameters. (The Enabled Quantizer Encode block does not appear in an example, but its behavior is similar to that of the Sampled Quantizer Encode

block.) 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 Sampled Quantizer Encode 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:

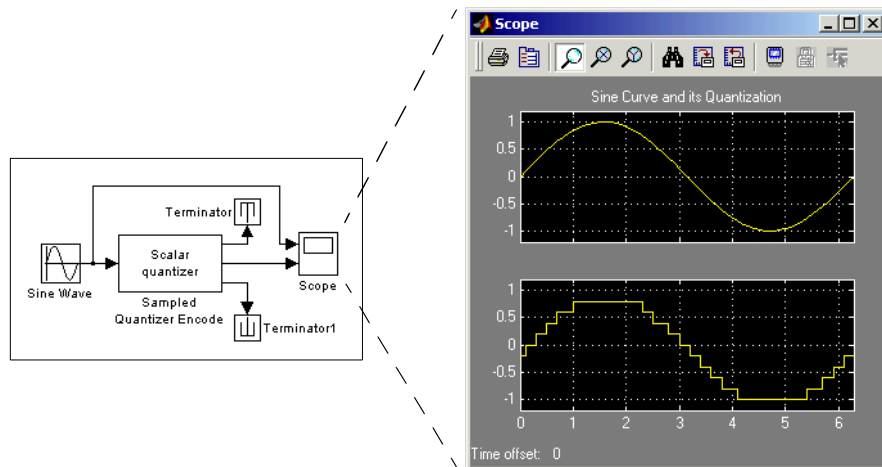
- Signal From Workspace, in the DSP Blockset DSP Sources library
 - Set **Signal** to $[-2.4, -1, -0.2, 0, 0.2, 1, 1.2, 1.9, 2, 2.9, 3, 3.5]^T$.
- Sampled Quantizer Encode
 - Set **Quantization partition** to $[0, 1, 3]$.
 - Set **Quantization codebook** to $[-1, 0.5, 2, 3]$.
 - Set **Input signal vector length** to 1.
 - Set **Sample time** to 1.
- Terminator, in the Simulink Signals & Systems 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. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 12. Running the model produces a scope image similar to the one in the figure. (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 quantizes a sampled sine wave and 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:

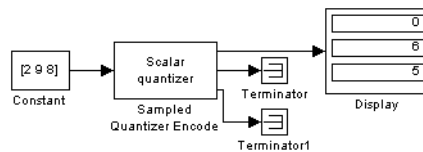
- Sine Wave, in the Simulink Sources library (*not* the Sine Wave block in the DSP Blockset DSP Sources library)

- Sampled Quantizer Encode
 - Set **Quantization partition** to [-1:.2:1].
 - Set **Quantization codebook** to [-1.2:.2:1].
 - Set **Input signal vector length** to 1.
- Terminator, in the Simulink Signals & Systems 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. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 2π . Running the model produces the scope image as shown in the figure. (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 Sampled Quantizer Encode 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].
- Sampled Quantizer Encode
 - 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.
- Set **Input signal vector length** to 3.
- Terminator, in the Simulink Signals & Systems 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. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation 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 Sampled Quantizer Encode block, connecting one branch to the input port of the Quantizer Decode block, and connecting the output of the Quantizer Decode block to another Display block. If the two source coding blocks' **Quantization codebook** parameters match, then the output of the Quantizer Decode block will be the same as the second output of the Sampled Quantizer Encode block. Thus the Quantizer Decode block partially duplicates the functionality of the Sampled Quantizer Encode block, but requires different input data and fewer parameters.

Implementing Differential Pulse Code Modulation

The quantization in the section “Quantizing a Signal” on page 1-31 requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM). The DPCM Encoder and DPCM Decoder blocks can help you implement a DPCM predictive quantizer.

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in “Representing Quantization Parameters” on page 1-31, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. Instead of quantizing x itself, the encoder quantizes the *predictive error*, which is the difference between the educated guess and the actual value. The special case

when the numerator is linear and the denominator is 1 is called *delta modulation*.

For more information about how DPCM works, see [1] in “Selected Bibliography for Source Coding” on page 1-40, or look underneath the masks of the DPCM Encoder and DPCM Decoder blocks.

Representing Predictors

This blockset implements predictors using an IIR filter. Just as you can specify a filter using a rational function of z^{-1} , you specify the predictor by giving its numerator and denominator. In block masks, the numerator and denominator are vectors that list the coefficients in order of ascending powers of z^{-1} .

The numerator’s constant term must be zero. This makes sense conceptually because the filter’s output is meant to *predict* the present signal without actually knowing its value.

In most applications, the denominator is the constant function 1.

Coded and Decoded Signals

If you encode a given signal using DPCM, then two resulting signals are the quantization index and the quantization-encoded signal. These correspond exactly to the two outputs of an ordinary quantization encoder. In both instances, the quantization index tells which partition interval a signal lies in, and the quantization-encoded signal tells which codebook values correspond to those partition intervals.

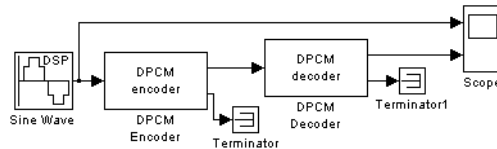
To use the DPCM Decoder block to recover a message that has been through the DPCM Encoder block, connect the quantization index signal, *not* the quantization-encoded signal, to the input port of the DPCM Decoder block.

The DPCM Decoder block outputs two signals. The first output is the attempted recovery of the message that first entered the DPCM encoder (assuming the encoder and decoder have matching parameters). The second output comes directly from the underlying quantization decoder. It represents the quantized predictive error, not the recovered message itself.

Example: Using DPCM Encoding and Decoding

A simple special case of DPCM quantizes the difference between the signal’s current value and its value at the previous step. Thus the predicted value equals the actual value at the previous step. The model below implements this

scheme. It encodes a sine wave, decodes it, and plots both the original and decoded signals.

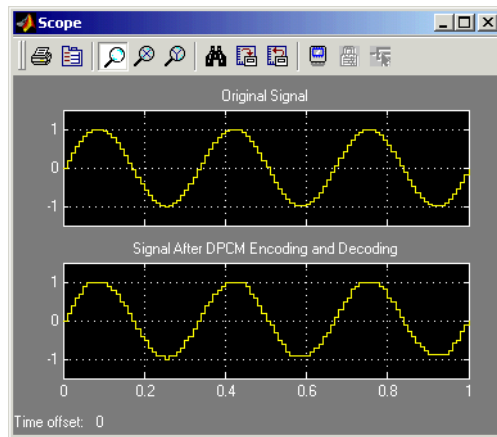


To open the completed model, click [here](#) in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Sine Wave, in the DSP Blockset DSP Sources library (*not* the Sine Wave block in the Simulink Sources library)
 - Set **Frequency** to 3.
 - Set **Sample time** to .01.
- DPCM Encoder
 - Set **Predictor numerator** to [0, 1].
 - Set **Quantization partition** to [-10:9]/10.
 - Set **Quantization codebook** to [-10:10]/10.
 - Set **Sample time** to .01.
- DPCM Decoder
 - Set **Predictor numerator**, **Quantization codebook**, and **Sample time** to the values given for the DPCM Encoder block.
- Terminator, in the Simulink Signals & Systems 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. Also, from the model window's **Simulation** menu, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 1.

Running the model produces scope images similar to those below. (To make the axis ranges and titles exactly match those below, right-click each plot area in the scope and select **Axes properties**.)



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 *comparer*.

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.

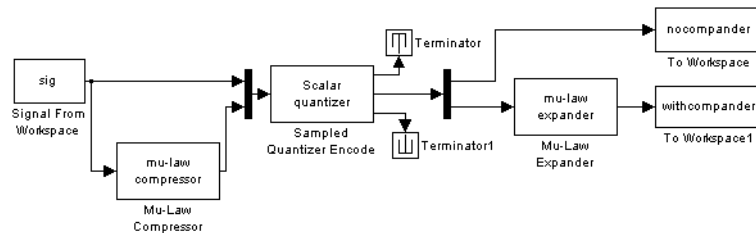
Example: Using a μ -Law Comparer

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
```

Now, the model in the following figure performs two computations. One computation uses the Sampled Quantizer Encode 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 Sampled Quantizer

Encode 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 DSP Blockset DSP Sources library
 - Set **Signal** to sig.
- Mu-Law Compressor
 - Set **Peak signal magnitude** to $\max(\text{sig})$.
- Mux, in the Simulink Signals & Systems library
- Sampled Quantizer Encode, in the Source Coding library
 - Set **Quantization partition** to $0:\text{floor}(\max(\text{sig}))$.
 - Set **Quantization codebook** to $0:\text{ceil}(\max(\text{sig}))$.
 - Set **Sample time** to 1.
- Terminator, in the Simulink Signals & Systems library
- Demux, in the Simulink Signals & Systems library
- Mu-Law Expander
 - Set **Peak signal magnitude** to $\text{ceil}(\max(\text{sig}))$.
- Two copies of To Workspace, in the Simulink Sinks library
 - Set **Variable name** to nocompander and withcomponder, 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. Also, from the model window's **Simulation** menu, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 80. Run the model, then 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] Kondo, A. M. *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.
- [2] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, N.J., 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 maps 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:

- “Accessing Block Coding Blocks” on page 1-41
- “Block Coding Features of the Blockset” on page 1-42
- “Communications Toolbox Support Functions” on page 1-43
- “Channel Coding Terminology” on page 1-43

These topics describe how to simulate linear block coding:

- “Data Formats for Block Coding” on page 1-43
- “Using Block Encoders and Decoders Within a Model” on page 1-46
- “Examples of Block Coding” on page 1-46
- “Notes on Specific Block Coding Techniques” on page 1-49

For background material on the subject of block coding, see the works listed in “Selected Bibliography for Block Coding” on page 1-53.

Accessing Block Coding Blocks

You can open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library (`commLib`), or by typing

```
commedac2
```

at the MATLAB prompt.

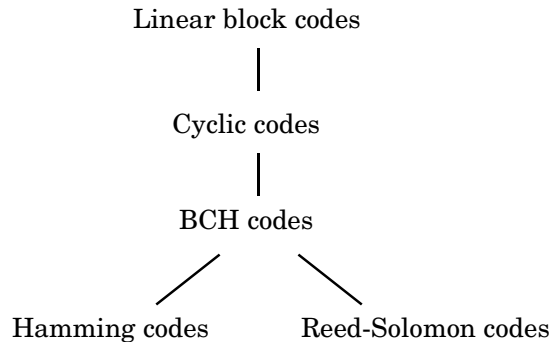
Then you can open the Block sublibrary by double-clicking its icon in the Error Detection and Correction library, or by typing

```
commb1kcod2
```

at the MATLAB prompt.

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 the section “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.

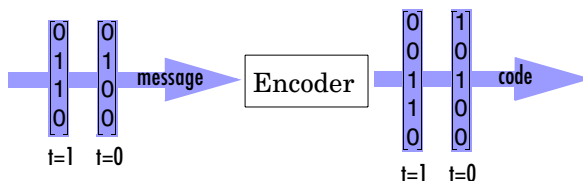
Data Formats for Block Coding

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)”. Reed-Solomon coding blocks also let you choose between binary and integer data, as described in “Integer Format (Reed-Solomon Only)” on page 1-45.

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 $M \cdot K$ and the corresponding code vector has length $M \cdot N$. 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, then 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.

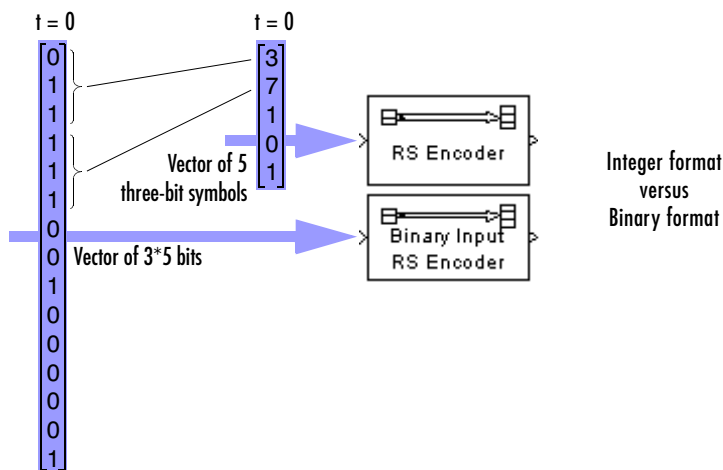
Using Serial Signals. If you prefer to structure messages and codewords as scalar signals, where several samples jointly form a message word or codeword, then you can use the Buffer and Unbuffer blocks in the DSP Blockset. Be aware that buffering involves latency and multirate processing. See the reference 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 model's **Format** menu, or attaching Signal Inspection 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 $M \cdot K$ 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 would be 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 will 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.

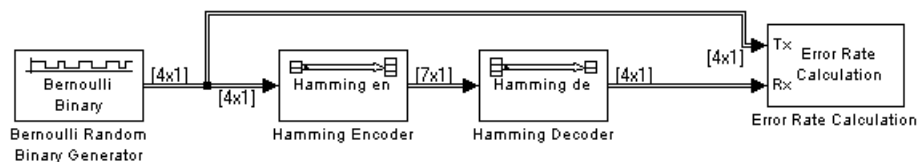
If you are unsure about the size of signals in your model, selecting **Signal dimensions** from the model's **Format** menu might help.

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. Also, 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:

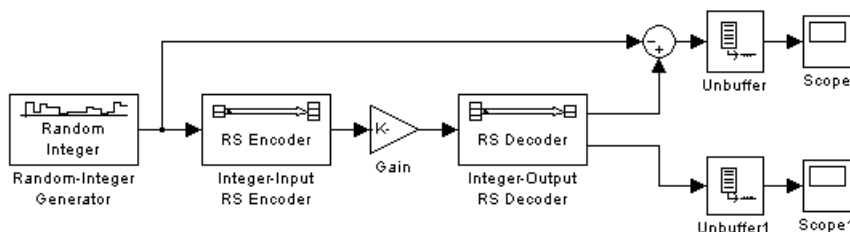
- Bernoulli Binary Generator, in the Comm Sources library
 - Set **Probability of a zero** to .5.

- 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. Also, use the **Signal dimensions** feature from 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 simplistic 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:

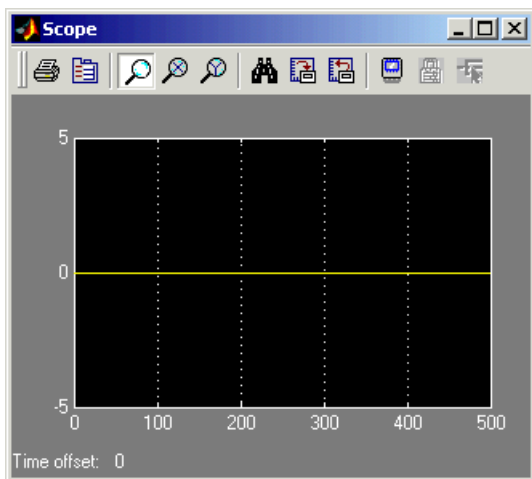
- Random Integer Generator, in the Comm Sources library
 - Set **M-ary number** to 15.
 - Set **Initial seed** to any prime number greater than 30, preferably the output of the randseed function.

- 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 `| - +`

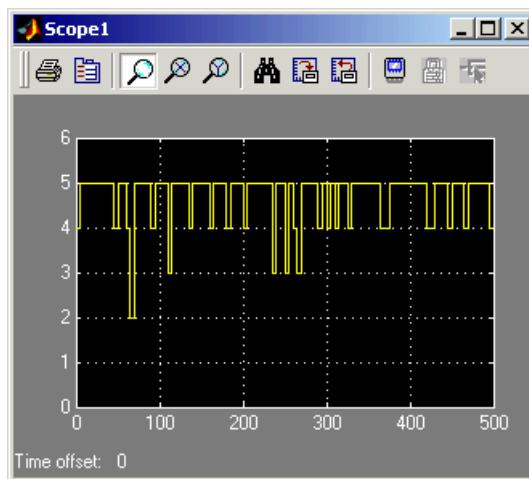
Connect the blocks as in the preceding figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then, in the **Simulation Parameters** dialog box, set **Stop time** to 500.

The vector length numbers appear on the connecting lines only if you select **Signal dimensions** from the model's **Format** menu. Notice that 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. Also, 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 scope images below. 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 left scope in the figure, 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 plot on the right 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 plot on the left 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. You should 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, then 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 greater than or equal to 3. The message length K can have only particular values. To see which values of K are valid for a given N , use the `bchpoly` function in the Communications Toolbox. For example, in the output below, the second column lists all possible message lengths that correspond to a codeword length of 31. The third column lists the corresponding error-correction capabilities.

```
params = bchpoly(31)
```

```
params =
```

31	26	1
31	21	2
31	16	3
31	11	5
31	6	7

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 allows you to state the error-correction capability of the code as the **Error-correction capability T** parameter. Providing the value here speeds the computation. If you do not know the code’s error-correction capability, setting this parameter to zero causes the block to calculate the error-correction capability when initializing. You can find out the error-correction capability using the `bchpoly` function in the Communications Toolbox.

The BCH Decoder block also returns error-related information during the simulation. The 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.

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 $\text{floor}((N - K) / 2)$, 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 $\text{floor}((N - K) / 2)$. 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 $\text{GF}(2^M)$ instead of $\text{GF}(2)$. 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 $M \cdot K$ (not K) for messages and the same integer $M \cdot N$ (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, N.J., Prentice-Hall, 1983.
- [3] Peterson, W. Wesley and E. J. Weldon, Jr., *Error-correcting Codes*, 2nd ed. Cambridge, Mass., 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.

Organization of This Section

- Describes how to access the Convolutional sublibrary of Error Detection and Correction
- Summarizes the software's capabilities
- Discusses parameters for convolutional coding (polynomial description and trellis description)
- Illustrates the use of the coding blocks for a rate $2/3$ code
- Explains how to implement a systematic encoder
- Illustrates soft-decision decoding

For background material on the subject of convolutional coding, see the works listed in "Selected Bibliography for Convolutional Coding" on page 1-67.

Accessing Convolutional Coding Blocks

You can open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library (comm1ib), or by typing

```
commedac2
```

at the MATLAB prompt.

Then you can open the Convolutional sublibrary by double-clicking its icon in the Error Detection and Correction library, or by typing

```
commconvcod2
```

at the MATLAB prompt.

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 useful for processing turbo codes.

Parameters for Convolutional Coding

To process convolutional codes (including turbo 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, then 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, then 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 1-56.
- If you design your encoder using a trellis diagram, then 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, you can 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]);
```

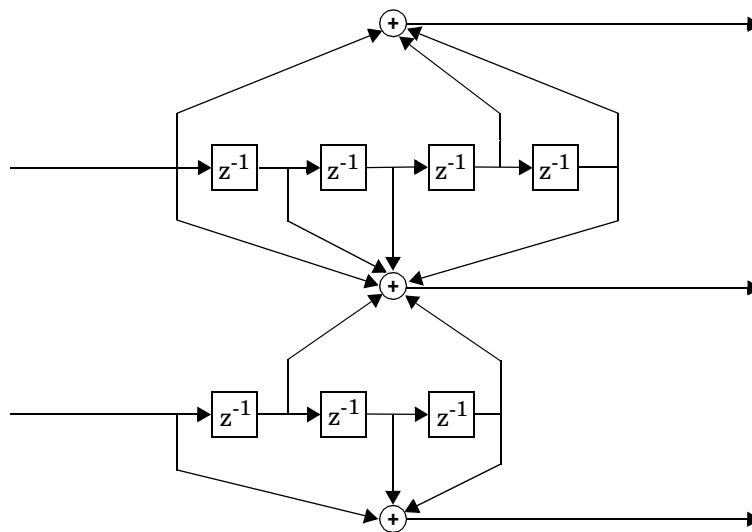
To use this encoder with one of the convolutional coding blocks, simply place a `poly2trellis` command such as

```
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.

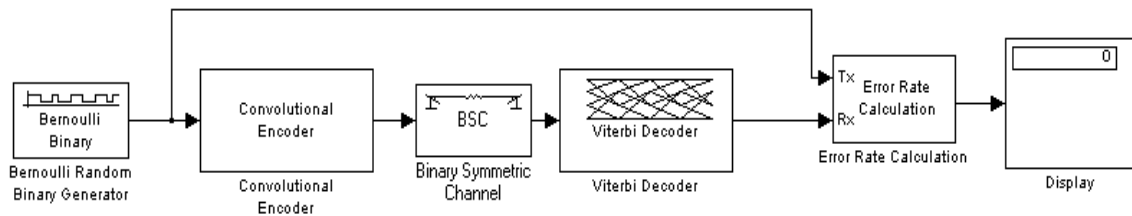
The encoder's constraint length is a vector of length 2 since the encoder has two inputs. The elements of this vector indicate the number of bits stored in each shift register, including the current input bits. Counting memory spaces in

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 i th row and j th column to indicate how the i th input contributes to the j th 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. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to `inf`.

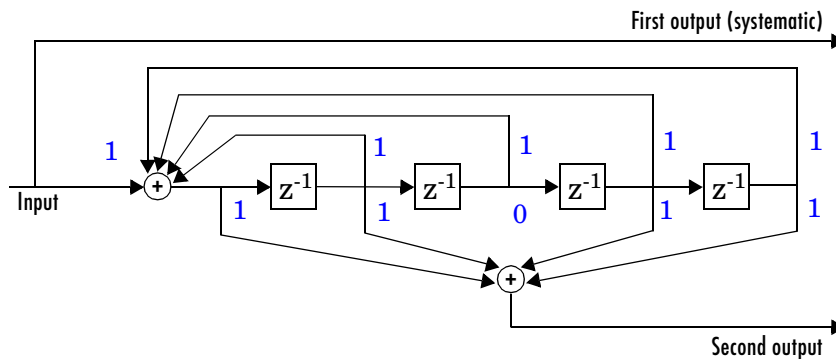
Notes on the model. The matrix size annotations appear on the connecting lines only if you select **Signal Dimensions** from the model's **Format** menu. Notice that the encoder accepts a 2-by-1 frame-based vector and produces a 3-by-1 frame-based vector, while the decoder does the opposite. The **Samples per frame** parameter in the Bernoulli Binary Generator block is 2 because the block must generate a message word of length 2.

Also notice that 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 examined the transmitted and received signals as matrices in the MATLAB workspace, then you would 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 three numbers indicate 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. Note that 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 code words. 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. Note that 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.

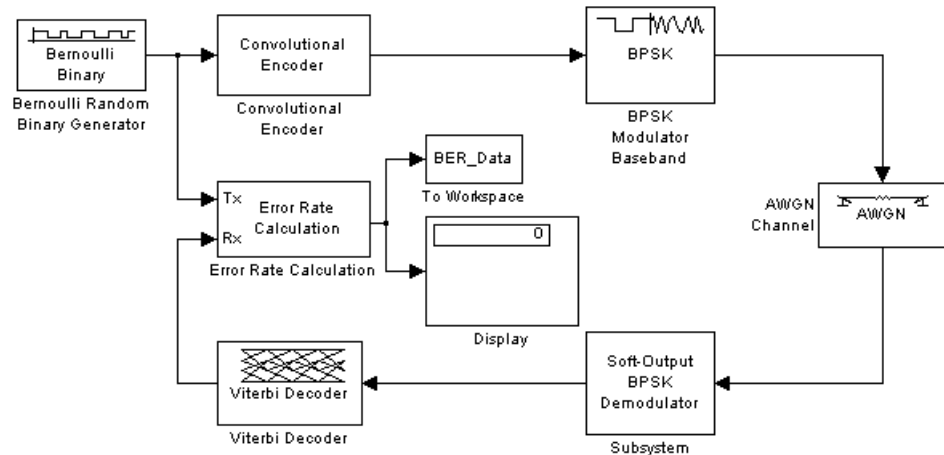
Example: Soft-Decision Decoding

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” on page 1-60
- “Defining the Convolutional Code” on page 1-61
- “Mapping the Received Data” on page 1-62
- “Decoding the Convolutional Code” on page 1-63
- “Delay in Received Data” on page 1-64
- “Comparing Simulation Results with Theoretical Results” on page 1-64

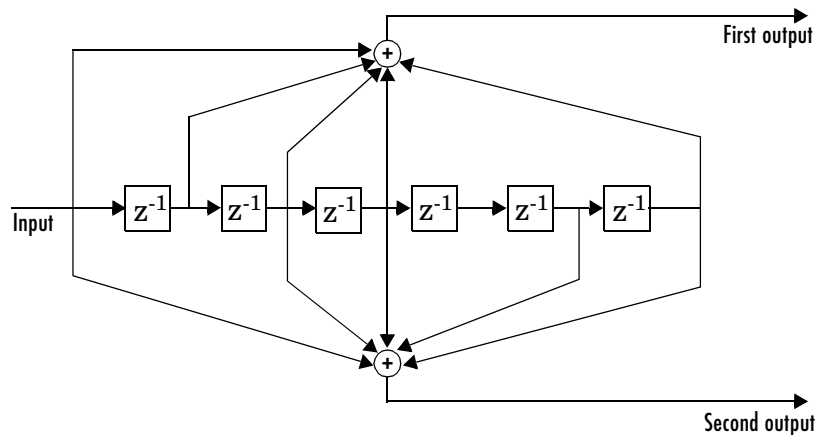
Overview of the Simulation

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.

Notice that 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”
- “Simulating Multiple Times to Collect Bit Error Rates” on page 1-65

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

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

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

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

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 probabilities, 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.

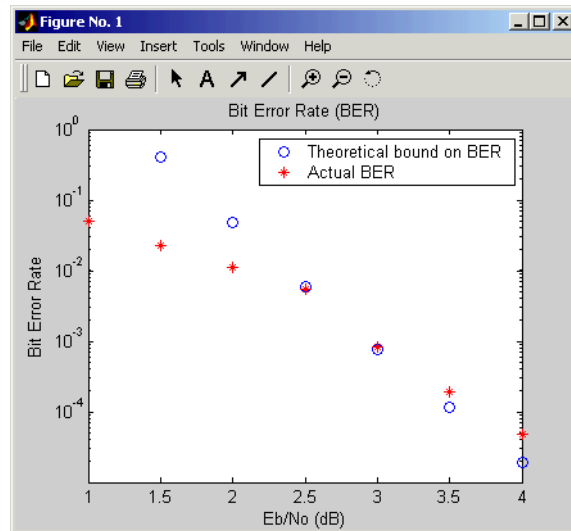
Note First open the model by clicking here in the MATLAB Help browser. Then execute these commands, which might take a few minutes.

```
% 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, pp.77-79, May 1997.
- [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 TMO Progress Report*, vol. 42-127, November 1996. [This electronic journal is available at http://tmo.jpl.nasa.gov/tmo/progress_report/index.html.]
- [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, pp. 317-319, November 1999.

- [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, pp. 835-848, October 1971.
- [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, pp. 260-264, February 1998.

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 removes the appended checksum, applies the same rule to the truncated word, and compares the resulting checksum with the received checksum. If the two checksums differ, an error has occurred, and the transmitter must resend the message word.

Organization of this Section

This section covers the following topics:

- “Accessing CRC Blocks” on page 1-69
- “CRC Coding Features of the Blockset” on page 1-69
- “CRC Algorithm” on page 1-70

Accessing CRC Blocks

You can open the Error Detection and Correction library by double-clicking its icon in the main Communications Blockset library (`commlib`), or by typing

```
commedac2
```

at the MATLAB prompt.

Then you can open the CRC sublibrary by double-clicking on its icon in the Error Detection and Correction library, or by typing

```
commcrc2
```

at the MATLAB prompt.

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 receives a transmitted word and removes the appended checksum. The block then calculates a new checksum, and compares the received checksum with the new checksum. The block has two outputs. The first is the message word without the received checksum. The second output is a boolean error flag, which is 0 if the received checksum agrees with the new checksum, and a 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 $T = M \cdot x^r + C$, 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 $T = Q \cdot P + R$, where R is a polynomial of degree less than r, then 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, which implement the detection phase of the CRC algorithm, do the following:

- 1 Remove the checksum from the received input frame.
- 2 Compute the checksum for the received message word as in the transmission phase.
- 3 Compare the new checksum with the received checksum.
- 4 Output a 0 if the two checksums agree 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 $x^3 + x^2 + 1$.

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, $M \cdot x^3 = (x^6 + x^3 + x) \cdot P + x$. The remainder is $R = x$, so that the checksum is then [0 1 0]'. Note that an extra 0 is added on the left to make the checksum have length 3.

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.

You can open the Interleaving library by double-clicking its icon in the main Communications Blockset library (`commlib`), or by typing

```
comminterleave2
```

at the MATLAB prompt.

Then you can open the interleaving sublibraries by double-clicking their icons in the Interleaving library, or by typing these commands at the MATLAB prompt.

```
commblkintrlv2  
commcnvintrlv2
```

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 1-80.

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.

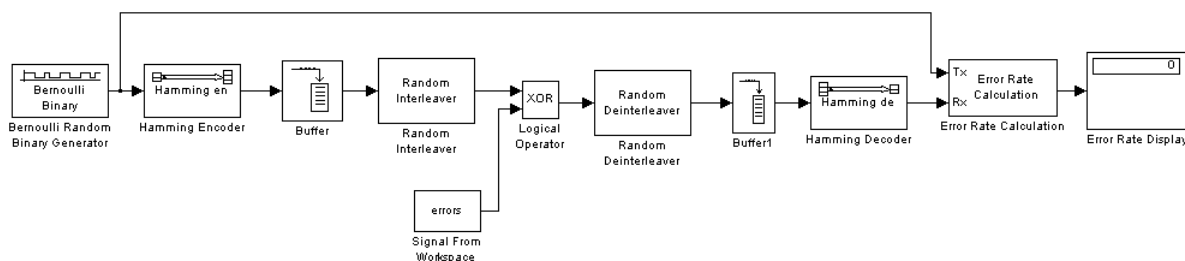
The Matrix Interleaver block accomplishes block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port column by column. For example, if the interleaver uses a 2-by-3 matrix to do its internal computations, then for an input of [1 2 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 1-75. The Signal From Workspace block imports this vector from the MATLAB workspace into the model, where the Logical Operator block XOR’s it 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 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 DSP 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 DSP Sources library
 - Set **Signal** to errors.
 - Set **Sample time** to 4/7.
 - Set **Samples per frame** to 84.
- Random Deinterleaver, 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 DSP 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 **Simulation 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`, type

```
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.

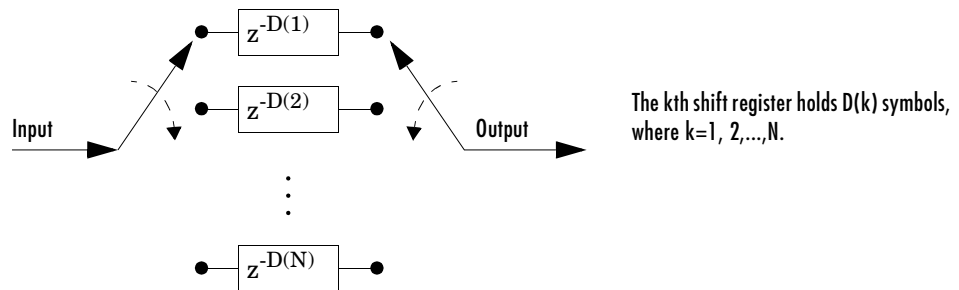
Consequently, you expect to see three 1s in 43 terms of the sequence. If there were no error correction in the model, the bit error rate would be approximately .0698.

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 code word.

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 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, you would use an **Interleaver delay** parameter of $[D(1); D(2); \dots; D(N)]$.

More specific is the Convolutional Interleaver block, in which the delay value for the k th 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

$$(\text{Number of shift registers}) * (\text{Maximum delay among all shift registers})$$

for the most general multiplexed interleaver. If your model incurs an additional delay between the interleaver output and the deinterleaver input, then 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 Integer Delay block in the DSP Blockset to adjust delays manually, if necessary.

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

$$(\text{Register length step}) * (\text{Rows of shift registers} - 1)$$

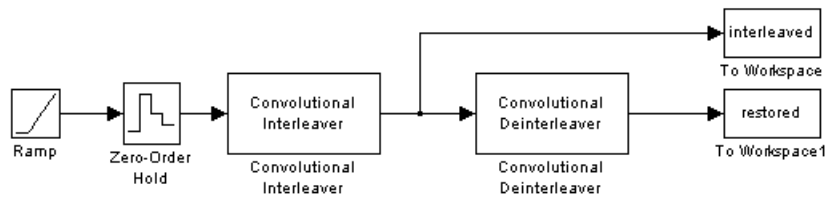
Helical Interleaver block. In the special case implemented by the Helical Interleaver/Helical Deinterleaver pair, the delay between the restored sequence and the original sequence is

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

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. Also, from the model window's **Simulation** menu, choose **Simulation parameters**; then, in the

Simulation Parameters dialog box, set **Stop time** to 20. Run the simulation, then 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).

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 D. J. Costello, 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. After giving instructions for accessing the analog modulation blocks, it goes on to discuss these topics:

- “Analog Modulation Features of the Blockset” on page 1-81
- “Baseband Modulated Signals Defined” on page 1-82
- “Representing Signals for Analog Modulation” on page 1-83
- “Timing Issues in Analog Modulation” on page 1-83
- “Filter Design Issues” on page 1-87

Accessing Analog Modulation Blocks

You can open the Modulation library by double-clicking its icon in the main Communications Blockset library (`commLib`), or by typing

```
commod2
```

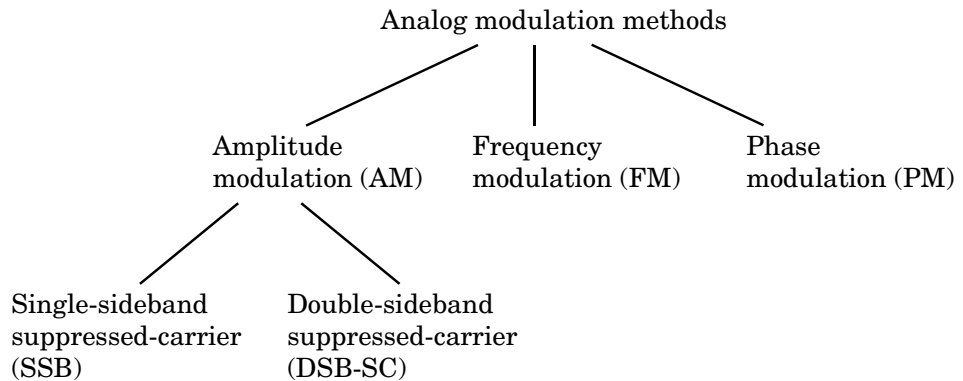
at the MATLAB prompt.

Then you can open the Analog Baseband and Analog Passband sublibraries by double-clicking their icons in the Modulation library or by typing these commands at the MATLAB prompt:

```
commanabbnd2  
commanapbnd2
```

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*. Baseband simulation, also known as the *lowpass equivalent method*, requires less computation. This blockset supports both baseband and passband simulation. This guide recommends and focuses more on baseband simulation. For a comparison of baseband simulation and passband simulation using example models, see “Comparing Baseband and Passband Simulation” on page 2-24.

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

Baseband Modulated Signals Defined

A baseband representation of a modulated signal is often more convenient for simulation than the passband representation is, because modeling a high-frequency carrier signal is computationally intensive. Suppose the modulated signal has the waveform

$$Y_1(t) \cos(2\pi f_c t + \theta) - Y_2(t) \sin(2\pi f_c t + \theta)$$

where Y_1 and Y_2 are amplitude terms, f_c is the carrier frequency, and θ is the carrier signal’s initial phase. A baseband simulation recognizes that this equals the real part of

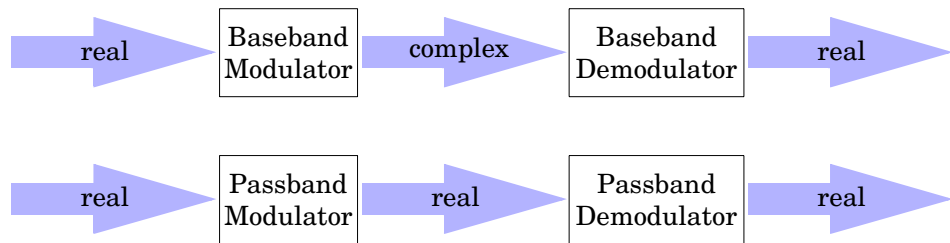
$$[(Y_1(t) + jY_2(t))e^{j\theta}]e^{j2\pi f_c t}$$

and models only the part inside the square brackets. Here j is the square root of -1. The complex vector y is a sampling of the complex signal

$$(Y_1(t) + jY_2(t))e^{j\theta}$$

Representing Signals for Analog Modulation

Analog modulation blocks in this blockset process only sample-based scalar signals. The data types of inputs and outputs are depicted in the figure below.



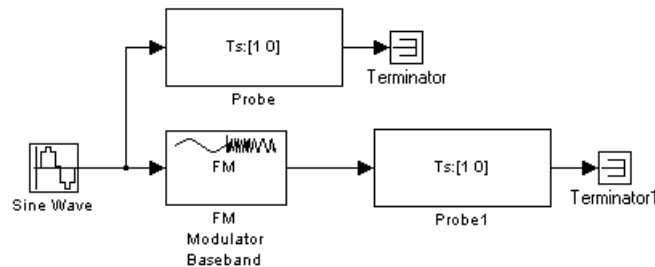
Timing Issues in Analog Modulation

A few timing issues are important for simulating analog modulation with this blockset. The following sections illustrate choices of signal sample times and simulation step sizes.

Signal Sample Times

All analog demodulators in this blockset produce discrete-time, not continuous-time, output. These blocks require you to specify the output sample time as a mask parameter. In addition, some analog modulators require you to specify the sample time as a mask parameter. Modulators in this category are FM Modulator Baseband, FM Modulator Passband, SSB AM Modulator Baseband, and SSB AM Modulator Passband.

Example Using a Modulator. In the following figure, both Signal Inspection blocks show a sample time of 1 second in their icons. (The display Ts: [1 0] indicates a sample time of 1 second and a sample time offset of 0.) Setting the **Sample time** parameter in the FM Modulator Baseband block to 1 is appropriate because the input to this block also has a sample time of 1 second.



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 DSP Blockset DSP Sources library)
 - Set **Sample time** to 1.
- FM Modulator Baseband, in the Analog Baseband sublibrary of the Modulation library
 - Set **Sample time** to 1.
- Two copies of Signal Inspection, in the Simulink Signal Attributes library
 - Clear all check boxes except **Probe sample time**.
- Two copies of Terminator, in the Simulink Signals & Systems library.

Connect the blocks as in the figure. Then select **Update diagram** from the model window's **Edit** menu, which updates the display on each Signal Attributes block's icon. (Running the model is not particularly instructive because it does not represent a complete system.)

Simulation Step Sizes

If you use passband modulation with continuous-time signals, then you need to set the simulation step size, based on the carrier frequency. By the Nyquist theorem, the simulation sampling rate must be at least twice as large as the modulation carrier frequency. Equivalently, the simulation step size must be no larger than half the modulation carrier period.

When you begin a new model, Simulink automatically determines the default step size. To change the step size from the default to a different value, use this procedure:

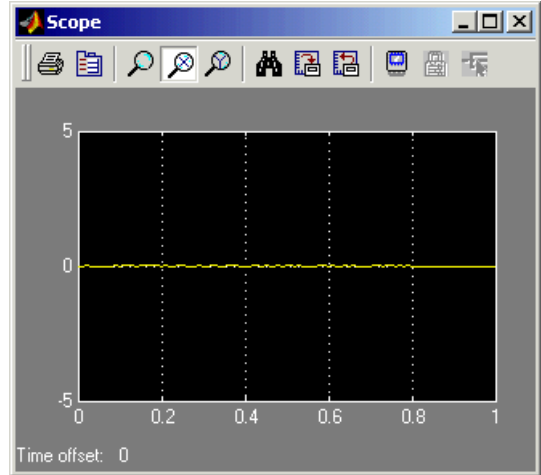
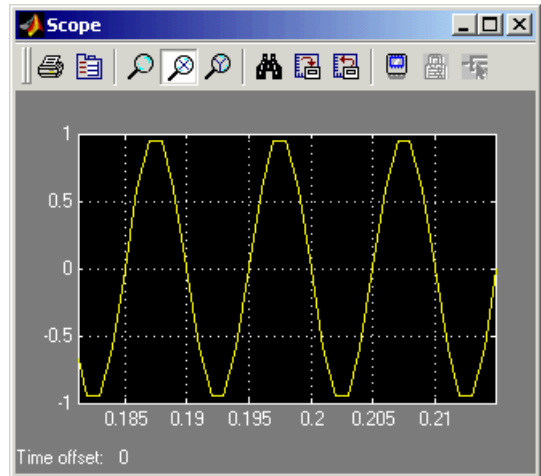
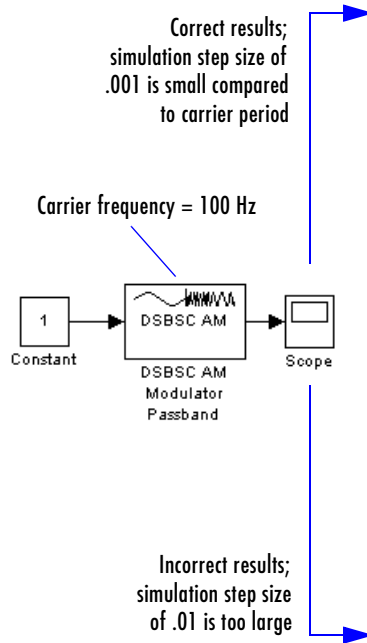
- 1 Select **Simulation parameters** from the model window's **Simulation** menu.
- 2 Select the **Solver** panel.
- 3 Set the **Max step size** and **Initial step size** parameters to numerical values (that is, not auto) that are appropriate for your model.

In some situations, Simulink automatically corrects a faulty simulation step size. For example, if a signal in your model has a sample time of .1 second and you set the model's **Max step size** parameter to 1, then running the model produces this response in the command window.

```
Warning: Maximum step size (1) is larger than the fastest discrete
sampling period (0.1) time. Setting maximum step size to 0.1.
```

Example Using Step Size Relative to Carrier Period. The model below illustrates why the simulation step size in a passband simulation must be appropriate for a given carrier frequency. The first Scope image shows the correct result of modulating a constant signal using double-sideband suppressed-carrier amplitude modulation, while the second Scope image shows incorrect results. The incorrect results occur because the simulation step size is too large relative to the modulation carrier frequency.

In this case, the DSBSC AM Modulator Passband block uses a **Carrier frequency** parameter of 100. That is, the carrier's period is .01 second and an appropriate simulation step size is no larger than .005. Therefore, a simulation step size of .01 second is too large to satisfy the Nyquist criterion. However, a simulation step size of .001 second is sufficiently small.



To open the completed model, click here in the MATLAB Help browser. To build the model, gather these blocks with their default parameters:

- Constant, in the Simulink Sources library
- DSBSC AM Modulator Passband, in the Analog Passband sublibrary of the Modulation library
- Scope, in the Simulink Sinks library

Connect the blocks as in the figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 1.

To generate the correct results as in the first Scope image in the figure, return to the **Simulation Parameters** dialog box and set both the **Max step size** and **Initial step size** parameters to .001. Then run the model and use the Scope window's zooming tools to study the sinusoidal output curve. You can also generate incorrect results, as in the second Scope image in the figure, by changing the **Max step size** and **Initial step size** parameters to .01 and running the model again.

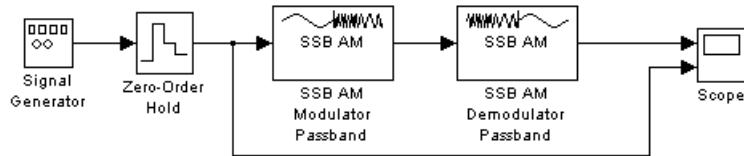
Filter Design Issues

After demodulating, you might want to filter out the carrier signal, especially if you are using passband simulation. The Signal Processing Toolbox provides functions that can help you design your filter, such as `butter`, `cheby1`, `cheby2`, and `ellip`. Different demodulation methods have different properties, and you might need to test your application with several filters before deciding which is most suitable. This section mentions two issues that relate to the use of filters: cutoff frequency and time lag.

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 limits the bandwidth of the message signal. If the cutoff frequency is too high, the carrier frequency 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.



Before building the model, first execute this command at the MATLAB prompt:

```
[num,den] = butter(2,25*.01);
```

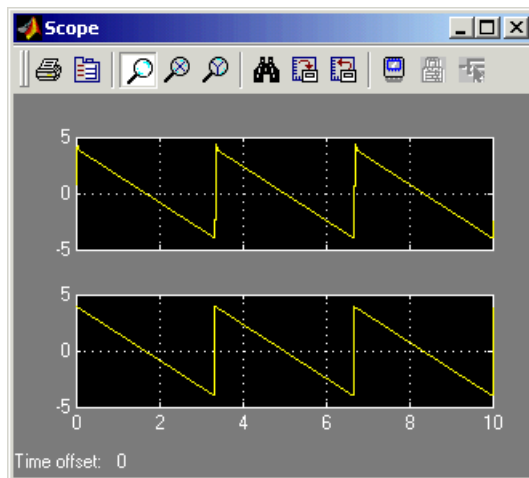
Here, 2 is the order of the Butterworth filter, 25 is the carrier signal frequency, and .01 is the sample time of the signal in Simulink. The variables num and den represent the numerator and denominator, respectively, of the filter's transfer function. These variables reside in the MATLAB workspace, where Simulink can access them during the simulation. The butter function is in the Signal Processing Toolbox.

Now to open the completed model, click here in the MATLAB Help browser. 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 **Time delay for Hilbert transform filter** to .1.
 - Set **Sample time** to .01.
- SSB AM Demodulator Passband, in the Analog Passband sublibrary of the Modulation library
 - Set **Carrier frequency** to 25.
 - Set **Lowpass filter numerator** to num.
 - Set **Lowpass filter denominator** to den.

- Set **Sample time** to `.01`.
- Scope, in the Simulink Sinks library
 - After double-clicking on the block to open it, click the **Parameters** icon and set **Number of axes** to 2.

Connect the blocks as in the figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation 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.



Other Filter Cutoffs. To see the effect of a lowpass filter with a *higher* cutoff frequency, type

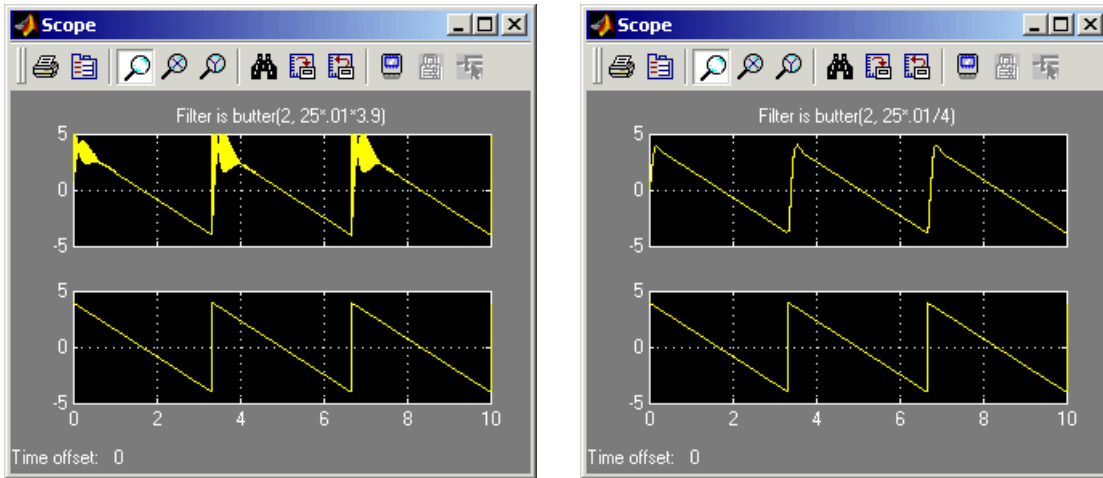
```
[num,den] = butter(2,25*.01*3.9);
```

at the MATLAB prompt and then run the simulation again. The new result is the left image in the following figure. 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, type

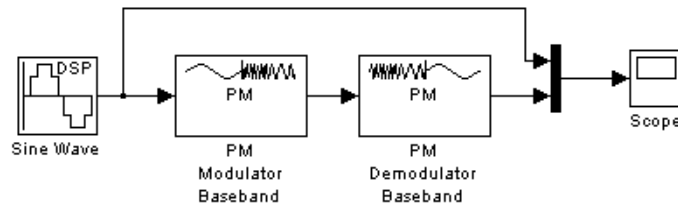
```
[num,den] = butter(2,25*.01/4);
```

at the MATLAB prompt and then run the simulation again. The new result is the right image in the figure below. The lower cutoff frequency narrows the bandwidth of the demodulated signal.



Example: Time Lag from Filtering

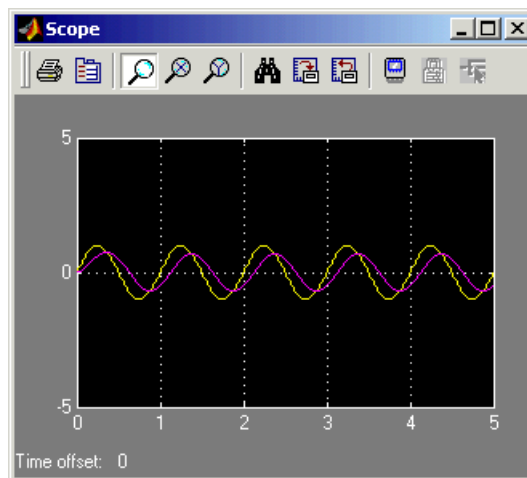
There is invariably a delay between a demodulated signal and the original received signal. Both the filter order and the filter parameters directly affect the length of this delay. The following example illustrates the delay by plotting a signal before modulation and after demodulation. The curve with amplitude 1 is the original sine wave and the other curve is the recovered signal.



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 DSP Blockset DSP Sources library (*not* the Sine Wave block in the Simulink Sources library)
 - Set **Frequency** to 1
- PM Modulator Baseband, in the Analog Baseband sublibrary of the Modulation library. Use default parameters.
- PM Demodulator Baseband, in the Analog Baseband sublibrary of the Modulation library. Use default parameters.
- Mux, in the Simulink Signals & Systems library
- Scope, in the Simulink Sinks library

Connect the blocks as in the figure. Also, from the model window's **Simulation** menu, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 5. Running the model produces the following scope image.



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. After giving instructions for accessing the digital modulation blocks, it goes on to discuss these topics:

- “Digital Modulation Features of the Blockset” on page 1-93
- “Representing Signals for Digital Modulation” on page 1-96
- “Delays in Digital Modulation” on page 1-97
- “Upsampled Signals and Rate Changes” on page 1-101
- “Examples of Digital Modulation” on page 1-104

For background material on the subject of digital modulation, see the works listed in “Selected Bibliography for Digital Modulation” on page 1-112.

Accessing Digital Modulation Blocks

You can open the Modulation library by double-clicking on icon in the main Communications Blockset library (`comm1lib`), or by typing

```
commod2
```

at the MATLAB prompt.

Then you can open the Digital Baseband and Digital Passband sublibraries by double-clicking on their icons in the Modulation library, or by typing these commands at the MATLAB prompt.

```
commdigbbnd2
```

```
commdigpbnd2
```

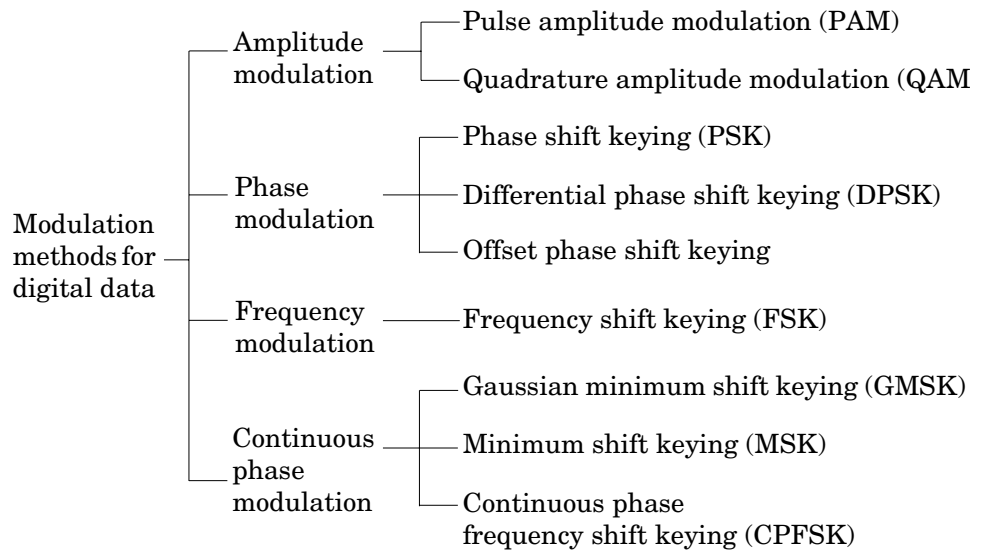
The Digital Baseband and Digital Passband libraries have sublibraries of their own. You can open each of these sublibraries by double-clicking on the icon listed in the table below, or by typing its name at the MATLAB prompt.

Table 1-1: Sublibraries of Digital Baseband and Digital Passband

Kind of Modulation	Icon in Digital Baseband or Digital Passband Library	Name of Sublibrary Model
Amplitude modulation	AM	commdigbbndam2, commdigpbndam2
Phase modulation	PM	commdigbbndpm2, commdigpbndpm2
Frequency modulation	FM	commdigbbndfm2, commdigpbndfm2
Continuous phase modulation	CPM	commdigbbndcpm2, commdigpbndcpm2

Digital Modulation Features of the Blockset

The figure below shows the modulation techniques that the Communications Blockset supports for digital data. All of the methods at the far right are implemented in both passband and baseband blocks. For a comparison of baseband simulation and passband simulation, see “Comparing Baseband and Passband Simulation” on page 2-24.



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 table below lists special-case modulators, their general counterparts, and the conditions under which the two are equivalent. The situation is analogous for demodulators.

Table 1-2: General and Specific Blocks

General Modulator	Specific Modulator	Specific Conditions
General QAM Modulator Baseband, General QAM Modulator Passband	Rectangular QAM Modulator Baseband, Rectangular QAM Modulator Passband	Predefined constellation containing 2^K points on a rectangular lattice

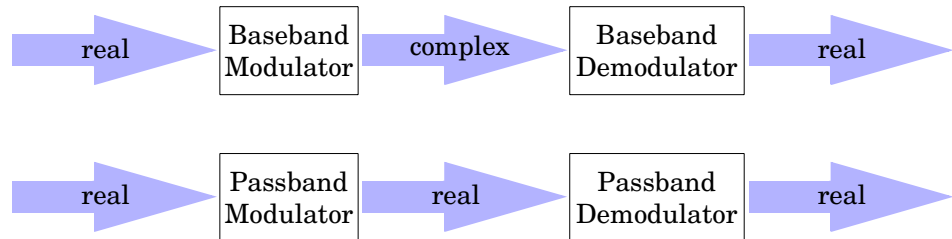
Table 1-2: General and Specific Blocks

General Modulator	Specific Modulator	Specific Conditions
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, CPM Modulator Passband	GMSK Modulator Baseband, GMSK Modulator Passband	M-ary number parameter is 2, Frequency pulse shape parameter is Gaussian .
	MSK Modulator Baseband, MSK Modulator Passband	M-ary number parameter is 2, Frequency pulse shape parameter is Rectangular, Pulse length parameter is 1.
	CPFSK Modulator Baseband, CPFSK Modulator Passband	Frequency pulse shape parameter is Rectangular, Pulse length parameter is 1.

Furthermore, the CPFSK Modulator Baseband and CPFSK Modulator Passband blocks are similar to the M-FSK Modulator Baseband and M-FSK Modulator Passband blocks, respectively, when the M-FSK blocks use 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.

Representing Signals for Digital Modulation

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



Note If you are simulating baseband modulation and want to separate the in-phase and quadrature components of the complex modulated signal, then 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**, then the block accepts binary representations of integers between 0 and $M-1$. It modulates each group of K bits, called a binary *word*. Also, these rules apply to the binary input mode:

- For baseband modulation, the input vector length must be an integer multiple of K . If the input is frame-based, then it must be a column vector.
- For passband modulation, the input must have length K and must be sample-based.

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**, then the block maps $[u(1) u(2) \dots u(K)]$ to the integer

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

and subsequently behaves as if this integer were the input value. Notice that $u(1)$ 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 $[1 1 0]$, then 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**, then 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.

Note The situations in the table are *not* mutually exclusive. If more than one situation applies to a given block or model, then the separate delays are additive. For example, if a passband demodulator in the AM sublibrary processes a sample-based signal and has a **Samples per symbol** parameter of 8, then the block's total delay is two output periods. As another example, if a passband OQPSK modulator-demodulator pair has a **Baseband samples per symbol** parameter of 7, then the two blocks together have a total delay of three output periods from the demodulator.

Table 1-3: 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, and Samples per symbol or Baseband samples per symbol parameter is greater than 1	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
All passband demodulators except OQPSK	Always	One output period
OQPSK modulator-demodulator baseband <i>pair</i>	Frame-based input	One output period
	Sample-based input, Samples per symbol parameter is greater than 1	Two output periods
	Sample-based input, Samples per symbol parameter is equal to 1, and the model uses a fixed-step solver with Mode parameter set to Auto or MultiTasking	Two output periods

Table 1-3: Delays Resulting from Digital Modulation or Demodulation

Modulation or Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
OQPSK modulator-demodulator baseband pair	Sample-based input, Samples per symbol parameter is equal to 1, and the model uses a variable-step solver or the Mode parameter is not set to Auto or MultiTasking	One output period
OQPSK modulator-demodulator passband pair	Sample-based input, and Baseband samples per symbol parameter is equal to 1	One output period
	Sample-based input, and Baseband samples per symbol parameter is greater than 1	Two output periods

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

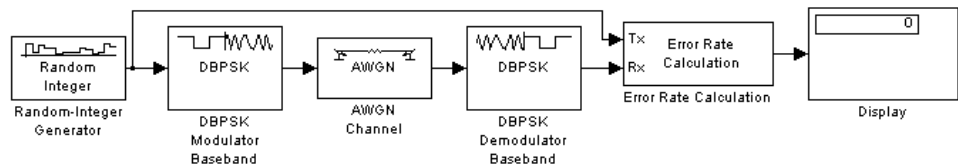
First Output Sample in DPSK Demodulation. In addition to the delays mentioned above, the DPSK, DQPSK, and DBPSK demodulators produce output whose first sample is unrelated to the input. This is related to the differential modulation technique, not the particular implementation of it.

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, then 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 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 **Samples per symbol** to 8.
- 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 **Samples per symbol** to 8.
- Error Rate Calculation, in the Comm Sinks library
 - Set **Receive delay** to 1.
 - Set **Computation delay** to 1.
 - 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. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation 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

Digital baseband modulation blocks can output an upsampled version of the modulated signal, while digital baseband 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.

Table 1-4: 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.
Modulation	Sample-based	Output vector is a scalar. Output sample time is $1/S$ times the input sample time.

Table 1-4: Processing of Upsampled Modulated Data (Except OQPSK Method) (Continued)

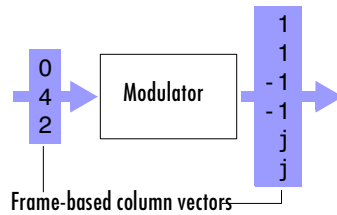
Computation Type	Input Frame Status	Result
Demodulation	Frame-based	Number of integers or binary words in the output vector is $1/S$ times the number of samples in the input vector. Output sample time equals the input sample time.
Demodulation	Sample-based	Output signal contains one integer or one binary 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, then 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.

Digital passband blocks can also process upsampled data, but only as an intermediate internal format. For more information about this, see the description of the **Baseband samples per symbol** parameter on the reference page for any digital passband modulation block. Also note that passband blocks process only sample-based data, not frame-based data.

Illustrations of Size or Rate Changes

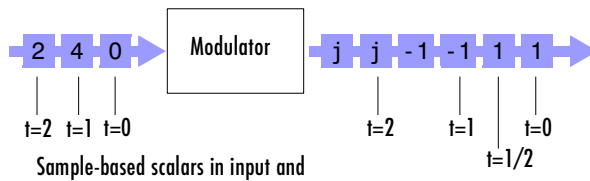
The following schematics illustrate how a baseband modulator (other than OQPSK) upsamples a triplet of frame-based and sample-based integers. In both cases, the **Samples per symbol** parameter is 2.

Frame-Based Upsampling



Output sample time = Input sample time
Output length = $2 \times$ (# of input integers)

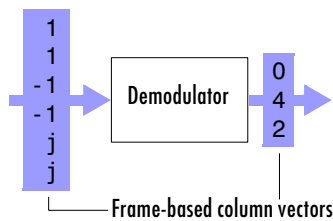
Sample-Based Upsampling



Output sample time = (Input sample time)/2
Output length = # of input integers

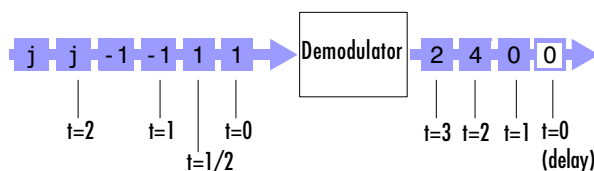
The following schematics illustrate how a demodulator (other than OQPSK or one from the CPM sublibrary) processes three doubly-sampled symbols using both frame-based and sample-based inputs. In both cases, the **Samples per symbol** parameter is 2. Notice that the sample-based schematic includes an output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

Frame-Based Upsampled Input



Output sample time = Input sample time

Sample-Based Upsampled Input



Output sample time = $2 \times$ (Input sample time)
Scalar input and output
First output element represents delay

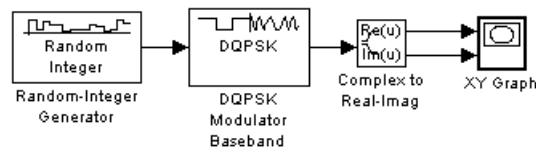
Examples of Digital Modulation

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” on page 1-104
- “Rectangular QAM Modulation and Scatter Diagram” on page 1-105
- “Phase Tree for Continuous Phase Modulation” on page 1-107
- “Passband Digital Modulation” on page 1-109

DQPSK Signal Constellation Points and Transitions

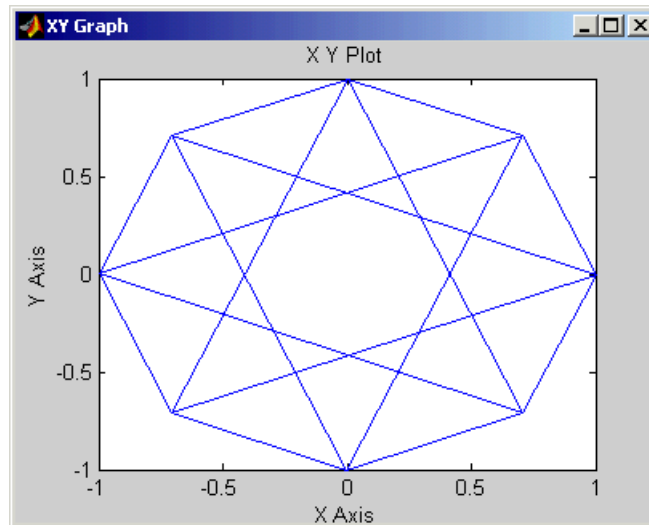
The model below plots the output of the DQPSK Modulator Baseband block. The image shows the possible transitions from each symbol in the DQPSK signal constellation to the next symbol.



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 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

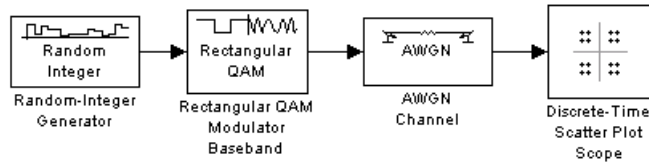
Use the blocks' default parameters unless otherwise instructed. Connect the blocks as in the figure. Running the model produces the following plot. The plot reflects the transitions among the eight DQPSK constellation points.



This plot illustrates $\pi/4$ -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.

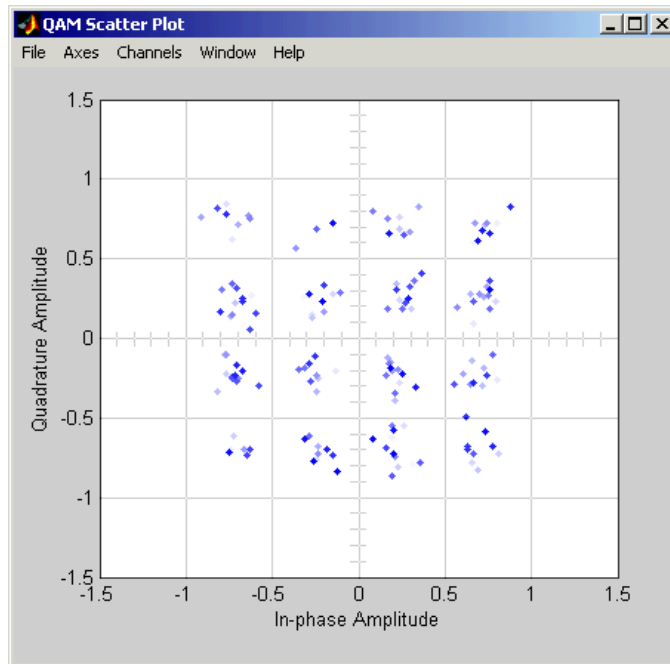


To open the completed model, click [here](#) in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Random Integer Generator, in the 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
 - Select **Show Plotting Properties**
 - Set **Points displayed** to 160
 - Set **New points per display** to 80
 - Select **Show Figure Properties**
 - Set **Scope position** to figposition([2.5 55 35 35]);
 - Set **Figure name** to QAM Scatter Plot

Connect the blocks as in the figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then, in the **Simulation 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

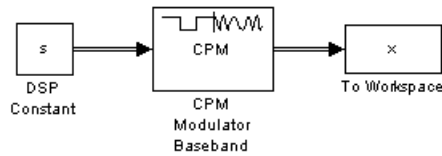
were no noise, the plot would show the 16 exact constellation points instead of clusters around the constellation points.



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.



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 DSP Sources library
 - Set **Constant value** to `s` (which will be 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');

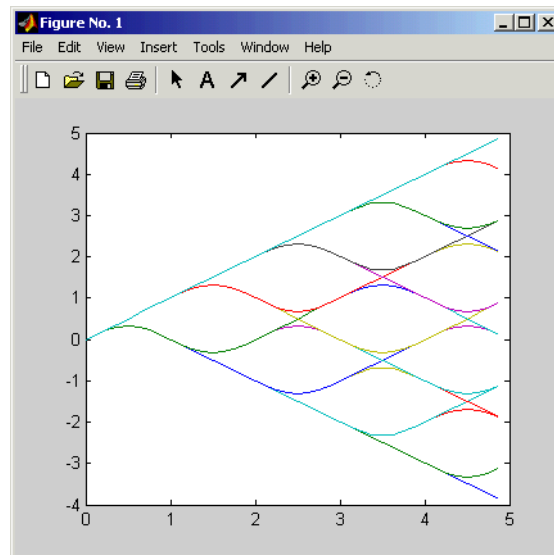
L = 5; % Symbols to display
```

```

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.

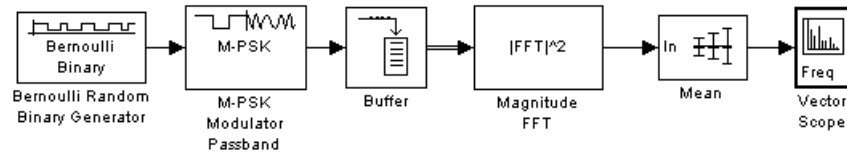


Passband Digital Modulation

The example below uses passband phase shift keying modulation and displays the spectrum of the modulated signal. The M-PSK Modulator Passband block's parameters satisfy necessary requirements for passband simulation because

- The input signal's sampling rate of 10 is less than the carrier frequency of 100.
- The modulated signal's sampling rate of 3000 exceeds the sum of twice the carrier frequency and twice the input sampling rate.

These requirements are mentioned on the reference page for the M-PSK Modulator Passband block.

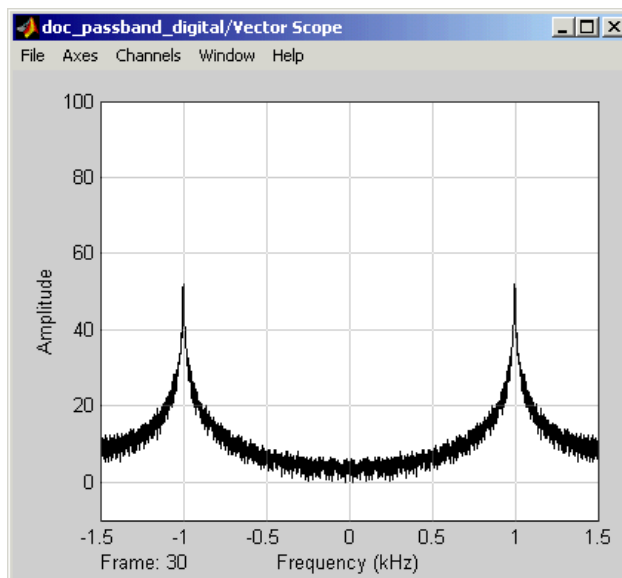


To open the completed model, click here in the MATLAB Help browser. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Data Sources sublibrary of the Comm Sources library.
 - Set **Probability of a zero** to [.5, .5].
 - Set **Initial seed** to any row vector containing 2 positive integers, preferably the output of the randseed function.
 - Set **Sample time** to .1.
- M-PSK Modulator Passband, in the PM sublibrary of the Digital Passband sublibrary of Modulation
 - Set **M-ary number** to 4.
 - Set **Input type** to **Bit**.
 - Set **Symbol period** to .1.
 - Set **Carrier frequency** to 1000.
 - Set **Carrier initial phase** to $\pi/4$.
 - Set **Output sample time** to 1/3000.
- Buffer, in the DSP Blockset Buffers sublibrary of the Signal Management library
 - Set **Output buffer size** to 1024.
- Magnitude FFT, in the DSP Blockset Power Spectrum Estimation sublibrary of the Estimation library

- Check the **Inherit FFT length from input dimensions** check box.
- Mean, in the DSP Blockset Statistics library
 - Check the **Running mean** check box.
 - Set **Reset port** to **None**.
- Vector Scope, in the DSP Blockset DSP Sinks library
 - Set **Input domain** to **Frequency**.
 - Check the **Axis properties** check box.
 - Set **Frequency range** to $[-F_s/2 \dots F_s/2]$.
 - Set **Maximum Y-limit** to 100.

Connect the blocks as in the preceding figure. Also, from the model window's **Simulation menu**, choose **Simulation parameters**; then in the **Simulation Parameters** dialog box, set **Stop time** to 10. Running the model produces the following spectral plot.



You might want to vary the modulation technique to see how this plot would change. For example, you can try replacing the M-PSK Modulator Passband block with the M-DPSK Modulator Passband or OQPSK Modulator Passband block.

Selected Bibliography for Digital Modulation

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg, *Digital Phase Modulation*, New York, Plenum Press, 1986.

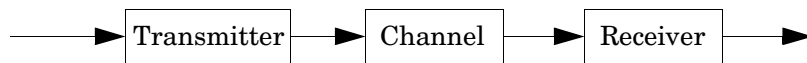
[2] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[3] 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.

[4] Smith, Joel G., "Odd-Bit Quadrature Amplitude-Shift Keying," *IEEE Transactions on Communications*, vol. COM-23, March 1975, pp. 385-389.

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. You can open the Channels library by double-clicking its icon in the main Communications Blockset library (`commlib`), or by typing

```
commchan2
```

at the MATLAB prompt.

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

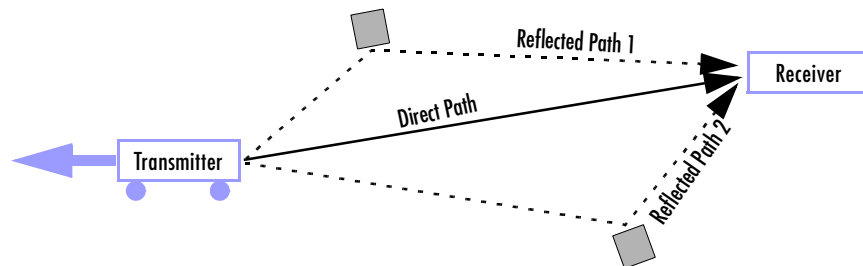
Fading Channels

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

- How to categorize the possible paths along which a signal can travel from the transmitter to the receiver in the situation that you want to model
- How to choose and configure a fading channel block based on the categorization
- An example that uses fading channels

Categorizing Signal Paths

The following figure depicts the two types of paths between a moving transmitter and a stationary receiver. The solid line is a *direct-line-of-sight* path, which might or might not exist in your situation. Each dotted line is a *reflected* path that the signal travels when it is reflected from one of the shaded shapes. The shaded shapes represent obstacles such as buildings or trees.



The situation in the figure is just an example. In general, you should analyze your system by considering these questions:

- Are there any reflected paths along which a signal can travel from transmitter to receiver? If so, how many?

- Is there a direct path from transmitter to receiver?
- What is the *relative* motion between the transmitter and receiver?

The first two questions will help you choose which fading channel blocks to use in your simulation, while the third question will help you choose appropriate parameters for the blocks.

Choosing and Configuring a Fading Channel Block

Once you categorize the types of signal paths in the situation you want to model, use the table below to determine the appropriate block (or blocks) for your simulation.

Table 1-1: Choosing a Fading Channel Block Based on Signal Paths

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

If a signal can use more than one reflected path, then 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, then they must have the same length; if exactly one of these parameters is a scalar, then the block expands it into a vector whose size matches that of the other **vector** parameter.)

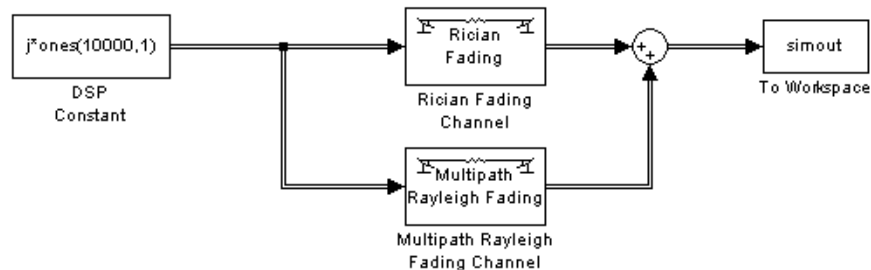
The relative motion between the transmitter and receiver influences the values of the blocks' parameters. For more details, see their reference pages, as well as the works listed in "Selected Bibliography for Channels" on page 1-118 if necessary.

Example: Using Fading Channels

The reference page for the Multipath Rayleigh Fading Channel block includes an example that illustrates the channel's effect on a constant signal.

Another example is the following model, which uses both the Multipath Rayleigh Fading Channel and the Rician Fading Channel blocks in parallel.

This combination of blocks simulates a mobile communication link in which the transmitted signal can travel to the receiver along a direct path as well as along three indirect paths. (The number of indirect paths is three because the Multipath Rayleigh Fading Channel block's **Gain vector** parameter is a vector of length three. Although the **Delay vector** parameter is a scalar, its value is applied to each of the three paths.)



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 DSP Blockset DSP Sources library
 - Set **Constant value** to `j*ones(10000,1)`.
 - Set **Output** to **Frame-based**.
 - Set **Frame period** to `.01`.
- Rician Fading Channel, with default parameter values
- Multipath Rayleigh Fading Channel
 - Set **Delay vector** to `[0 2e-6 3e-6]`.
 - Set **Gain vector** to `[0 -3 1]`.
- Sum, in the Simulink Math library
- To Workspace, in the Simulink directory
 - Set **Save format** to **Array**.

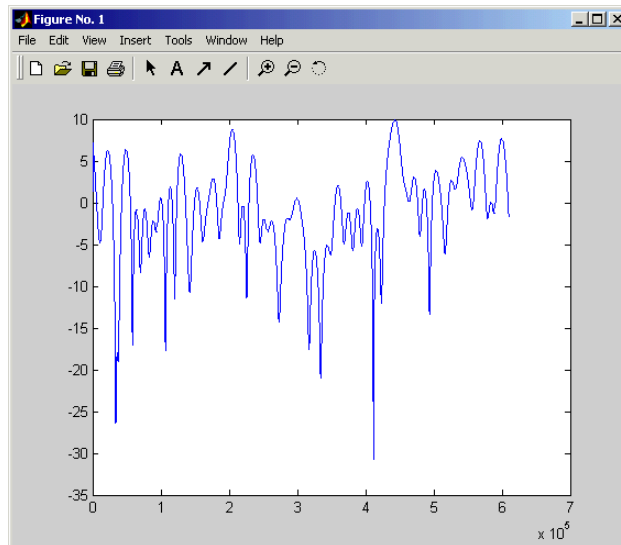
Connect the blocks as shown above. Also, from the model window's **Simulation** menu, choose **Simulation parameters**; then, in the **Simulation Parameters** dialog box, set **Stop time** to `0.6`.

Note To reduce execution time by logging less data to the workspace, set the **Decimation** parameter in the To Workspace block to 100. Then the variable `simout` will contain fewer entries, but its graph will look similar.

Run the model. After the simulation stops, plot the faded signal's power (versus sample number) by executing this command at the MATLAB prompt.

```
simout = simout.'; plot(20*log10(abs(simout(:))))
```

The resulting plot is shown in the figure below.



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.

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, pp. 422-431, April 1993.
- [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, Wiley, 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 the Blocks Model
- Scatter Plot Examples
- Example Using the RF Impairments Library Blocks

Types of RF Impairments 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 models 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.

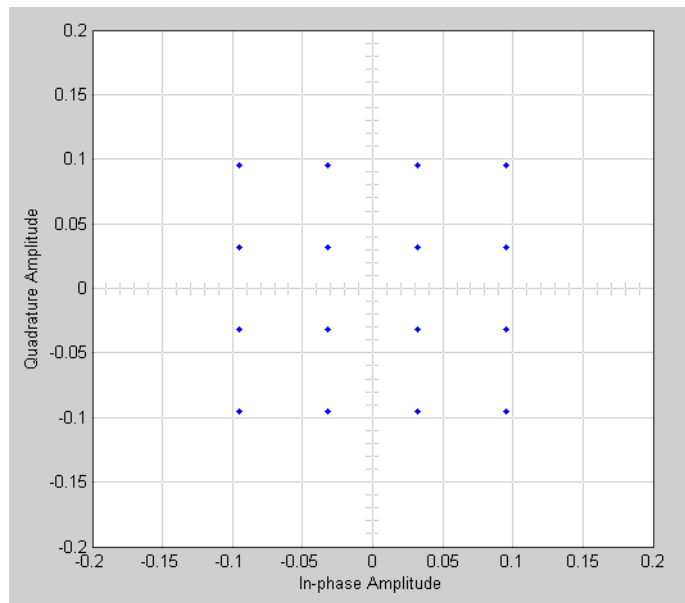
Receiver Thermal Noise and Free Space Path Loss

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:



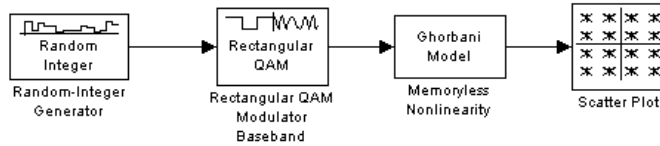
Constellation for 16-QAM

The scatter plots illustrate the effects of the following four blocks:

- Memoryless Nonlinearity Block
- I/Q Imbalance Block
- Phase/Frequency Offset Block
- Phase Noise Block

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

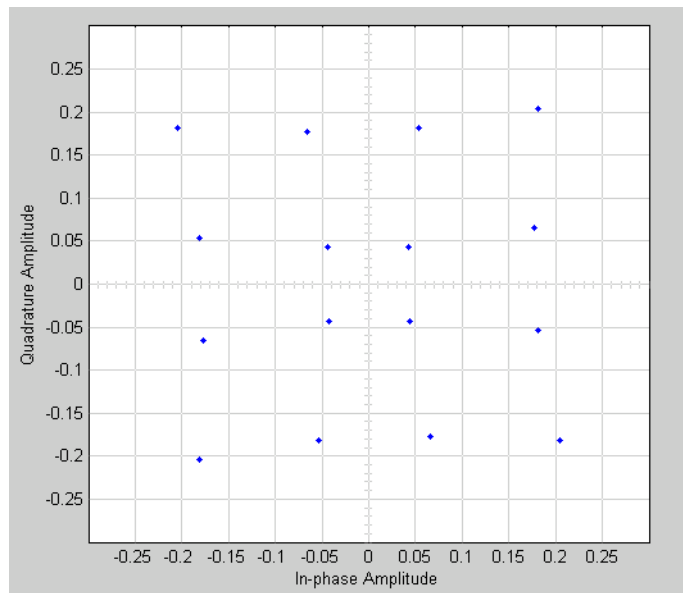
The model uses the Rectangular QAM Modulator Baseband block, from AM in the Digital Baseband Modulation sublibrary of the Modulation library. You can control the power of the block's output signal by the **Normalization method** parameter.

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 five 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

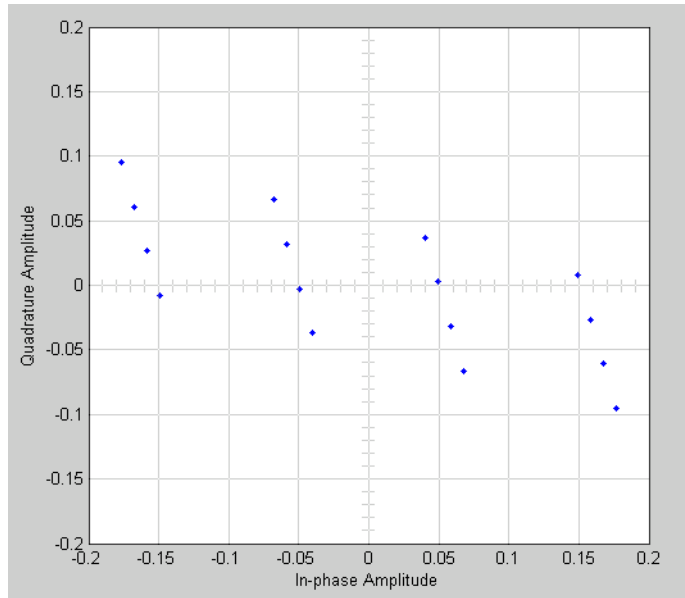
In the model shown in the preceding figure, the **Method** parameter is set to **Ghorbani model**. The following figure shows the scatter plot the model generates.



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

I/Q Imbalance Block

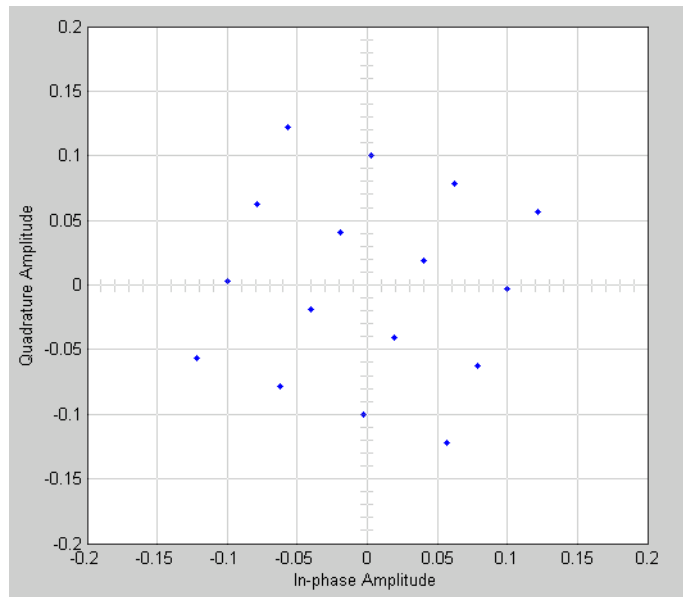
You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model with the I/Q Imbalance block. Set the block's **I/Q amplitude imbalance (db)** parameter to 10 and the **I/Q phase imbalance (deg)** parameter to 30.



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 with the Phase/Frequency Offset block. Set the block's **Frequency offset (Hz)** parameter to 0 and the **Phase offset (deg)** parameter to 70.

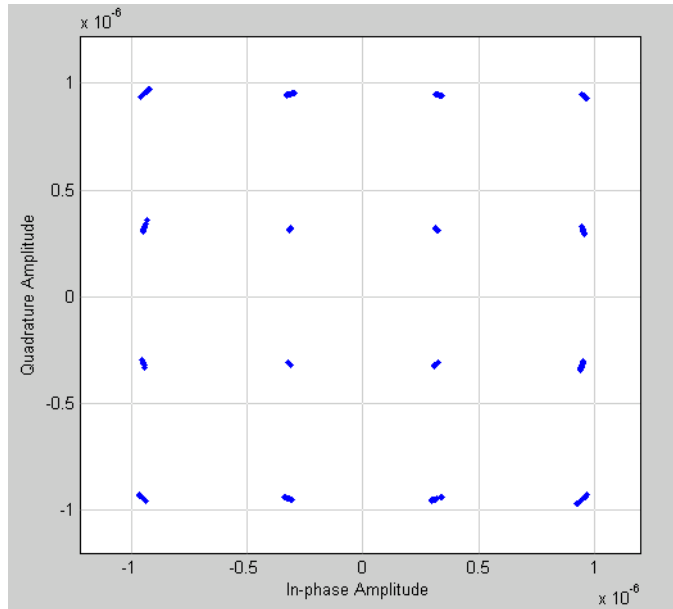


The **Frequency offset (Hz)** parameter adds a constant to the phase of the signal. The scatter plot corresponds to the standard constellation rotated by a fixed angle of 70 degrees.

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

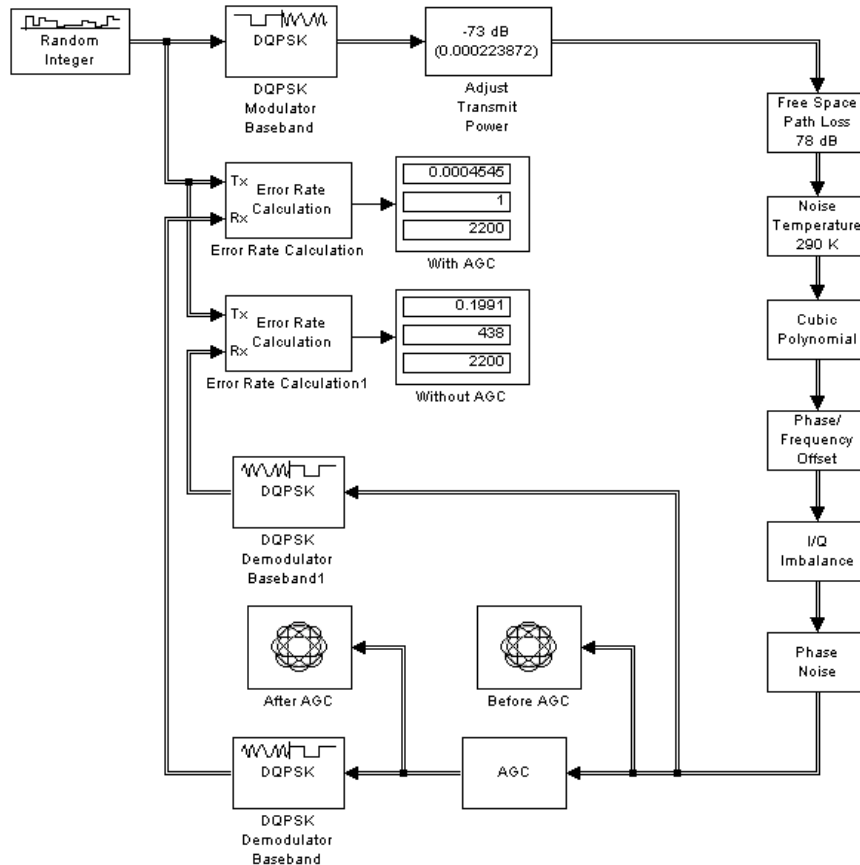
You can generate the next scatter plot by replacing the Memoryless Nonlinearity block in the 16-ary QAM Model with the Phase Noise block. Set the **Phase noise level (dBc/Hz)** parameter to -60 and the **Frequency offset (Hz)** parameter to 100.



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).



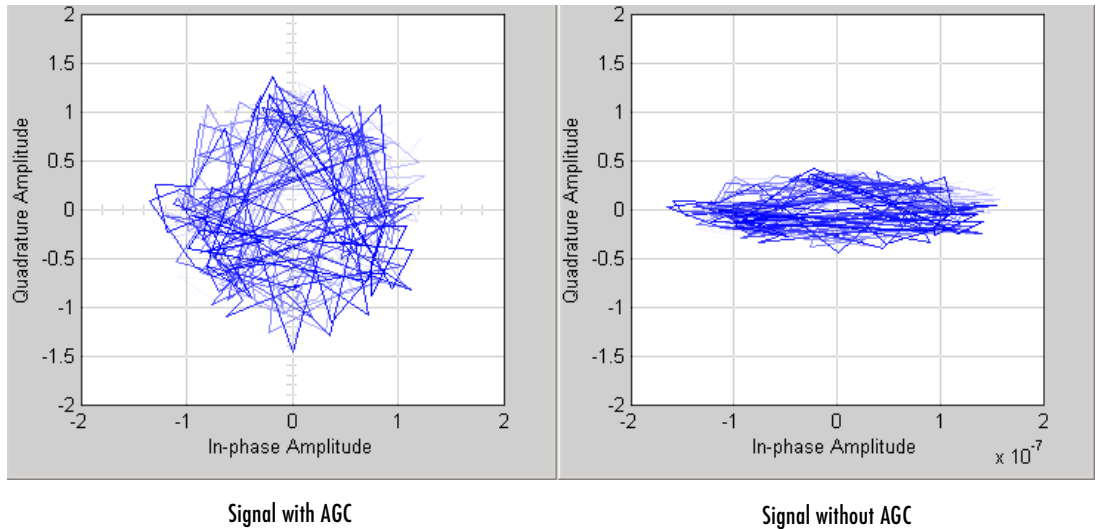
Overview of the Model

The model does the following:

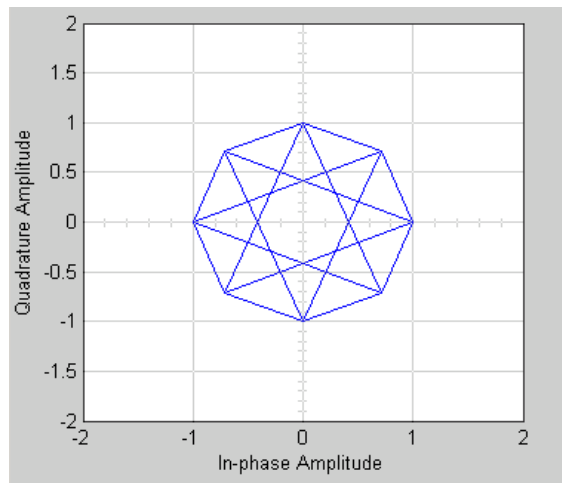
- Modulates a random signal using DQPSK modulation.
- Applies impairments to the signal using the blocks from the RF Impairments library.

- 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.



The trajectory of the signal with AGC more closely matches the undistorted trajectory for DQPSK, shown in the figure below, 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. A phase-locked loop, or PLL, can help accomplish this synchronization when used in conjunction with other components. 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.

Synchronization Features of the Blockset

This blockset contains four phase-locked loop blocks in its Synchronization library. You can open the Synchronization library by double-clicking its icon in the main Communications Blockset library (comm1ib), or by typing

```
commsync2
```

at the MATLAB prompt.

The following table indicates which block in the Synchronization library implements each supported type of PLL.

Table 1-2: Supported PLLs in Synchronization Library

Type of PLL	Block
Analog passband PLL	Phase-Locked Loop
Analog baseband PLL	Baseband PLL
Linearized analog baseband PLL	Linearized Baseband PLL
Digital PLL using a charge pump	Charge Pump PLL

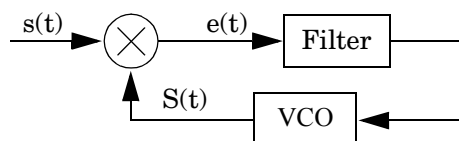
This section discusses these topics:

- “Overview of PLL Simulation” on page 1-131
- “Implementing an Analog Baseband PLL” on page 1-131
- “Implementing a Digital PLL” on page 1-132

For details about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization” on page 1-132.

Overview of PLL Simulation

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.



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. These 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

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

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, pp. 1849-1858, November 1980.
- [2] Gardner, F. M., "Phase Accuracy of Charge Pump PLLs," *IEEE Trans. on Communications*, vol. 30, pp. 2362-2363, October 1982.
- [3] Gupta, S. C., "Phase Locked Loops," *Proceedings of the IEEE*, vol. 63, pp. 291-306, February 1975 .
- [4] Lindsay, W. C. and C. M. Chie, "A Survey on Digital Phase-Locked Loops," *Proceedings of the IEEE*, vol. 69, pp. 410-431, April 1981.
- [5] Meyr, Heinrich and Gerd Ascheid, *Synchronization in Digital Communications*, vol. 1, New York, John Wiley & Sons, 1990.

Modeling Communication Systems

Computing Delays	2-3
Other References for Delays	2-3
Sources of Delays	2-4
ADSL Demo Model	2-4
Punctured Coding Model	2-9
Manipulating Delays	2-14
Delays and Alignment Problems	2-14
Aligning Words of a Block Code	2-17
Aligning Words for Interleaving	2-19
Aligning Words of a Concatenated Code	2-21
Comparing Baseband and Passband Simulation	2-24
Running a Passband Simulation	2-24
Running an Equivalent Baseband Simulation	2-25
Generating Error Curves	2-26
Speed of Baseband Versus Passband Models	2-28
Comparing Baseband and Passband Signals	2-30
Troubleshooting a Passband Simulation	2-32

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. The topics in this chapter are:

- “Computing Delays” on page 2-3
- “Manipulating Delays” on page 2-14
- “Comparing Baseband and Passband Simulation” on page 2-24

Because the examples in this chapter are larger than those of previous chapters, the discussions omit instructions for building the example models. You can open prebuilt copies of the models if you want to examine, run, or modify them.

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, then 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. The section covers the following topics:

- “Sources of Delays” on page 2-4
- “ADSL Demo Model” on page 2-4
- “Punctured Coding Model” on page 2-9

Other References for Delays

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

- “Delays in Digital Modulation” on page 1-97
- “Delays of Convolutional Interleavers” on page 1-77
- 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 Getting Started with the Communications Blockset. (See “Delays in the Model”.)
- “Example: A Rate 2/3 Feedforward Encoder” on page 1-56
- “Example: Soft-Decision Decoding” on page 1-60. (See “Delay in Received Data” on page 1-64.)
- “Example: Delays from Demodulation” on page 1-99

Sources of Delays

While some blocks are able to 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
- Viterbi Decoder block
- Buffering, downsampling, derepeating, and similar signal operations
- Explicit delay blocks, such as Integer 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 each of two Error Rate Calculation blocks in the model. The model includes delays from buffering, convolutional interleaving, and an explicit delay block. To open the ADSL demo model, type `adsl_sim` in the MATLAB Command Window.

In the ADSL demo, data follows one of two parallel paths, each of which incurs a different delay. 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 separately the path for noninterleaved data and the path for interleaved data.

Frame Periods in the Model

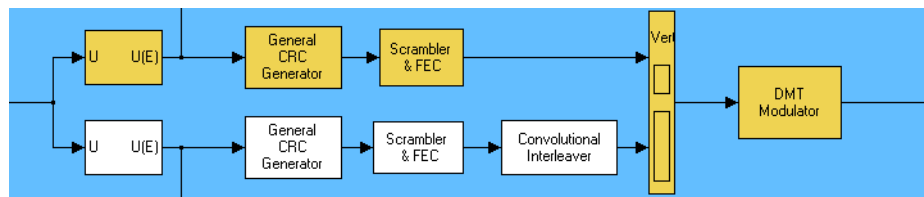
Before searching for individual delays, first observe that most signal lines throughout the model share the same frame period; to see this, enable the

Sample time colors option from the 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.

The four icons labeled Scrambler & FEC or Descrambler & FEC are yellow because they represent multirate systems. If you double-click any of those icons, you can see that inside the subsystems are yellow Buffer blocks whose output signals are the same color as the signals at the top level of the model. As a consequence, you can use output frames as a unit for measuring delays in the Buffer blocks and then add the result to any top-level delays when computing the cumulative delay.

Path for Noninterleaved Data

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

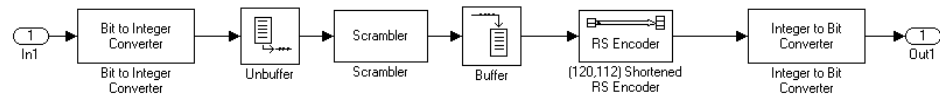


The table below summarizes the delays in the path for noninterleaved data. Subsequent paragraphs explain the delays in more detail and explain why the

total delay relative to the Error Rate Calculation block is two frames, or 1552 samples.

Block	Delay, in Output Samples from Individual Block	Delay, in Frames	Delay, in Input Samples to Error Rate Calculation Block
Buffer, in Scrambler & FEC subsystem	112	1	776
Buffer, in Descrambler & FEC subsystem	112	1	776
Total	N/A	2	1552

Scrambler & FEC. The Scrambler & FEC icon represents the following subsystem.



Notice that the subsystem includes an Unbuffer and Buffer pair. Buffering scalar data into vectors causes a delay because the block cannot produce fully meaningful output until it has received the specified number of samples from the scalar input stream. The set of parameters in the Buffer block causes the block to incur a delay of 112 samples, which represent one output frame.

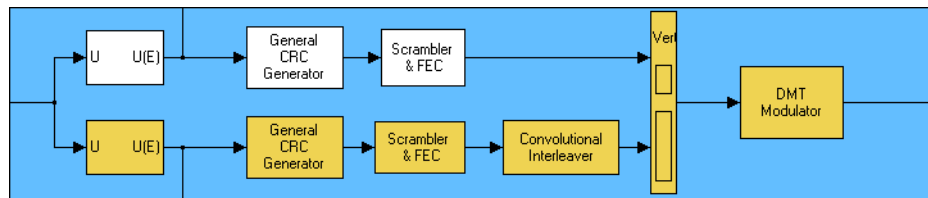
Descrambler & FEC. The noninterleaved path in the receiver portion of the model contains a corresponding Descrambler & FEC subsystem, which also contains a Buffer block. Like the transmitter's Buffer block, the receiver's Buffer block incurs a delay of one output frame.

Summing the Delays. No other blocks in the noninterleaved path of the demo cause any delays. Adding the two one-frame delays from the two Buffer blocks indicates that the total delay in the noninterleaved path is two frames.

Total Delay Relative to Error Rate Calculation Block. The Error Rate Calculation block that computes the value labeled Non Interleaved BER requires a **Receive delay** parameter value that is equivalent to two frames. The **Receive delay** parameter is measured in samples and each input frame to the Error Rate Calculation block contains 776 samples. Also, the frame period at the Buffer block's output equals the frame period at the Error Rate Calculation block's input. Therefore, the correct value for the **Receive delay** parameter is 1552 samples.

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 three frames, or 2328 samples.

Block	Delay, in Output Samples from Individual Block	Delay, in Frames	Delay, in Input Samples to Error Rate Calculation Block
Buffer, inside Scrambler & FEC subsystem	112	1	776
Buffer, inside Descrambler & FEC subsystem	112	1	776
Convolutional Interleaver and Convolutional Deinterleaver pair	40	1 (combined)	776 (combined)
Integer Delay	800		
Total	N/A	3	2328

Buffer Blocks. Like the noninterleaved path, the interleaved path contains a Buffer block in the transmitter and another Buffer block in the receiver. Together, these blocks cause a delay of two frames.

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 $5 \cdot 2 \cdot 4 = 40$ samples.

Integer Delay Block. The receiver portion of the interleaved path also contains an Integer Delay block, whose purpose is explained in “Aligning Words of a Block Code” on page 2-17. This block explicitly causes a delay of 800 samples having the same sample time as the 40 samples of delay from the

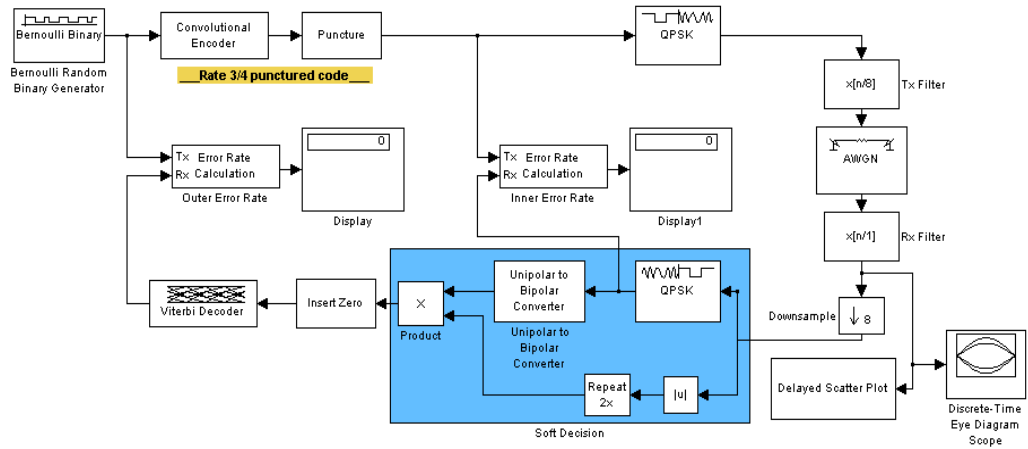
interleaver/deinterleaver pair. Therefore, the total delay from interleaving, deinterleaving, and the explicit delay is 840 samples. These 840 samples make up one frame of data leaving the Integer Delay block.

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

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 three frames. 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 outputs 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 2328 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, type `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 model window's **Format** menu. Notice that 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. As a consequence, 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 DSP 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 2-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 that 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 3 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.

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 output 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 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 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

Block	Delay, in Samples at Individual Block	Delay, in Frames at Predominant Frame Rate	Delay, in Input Samples to Outer Error Rate Block
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 2-10.

Viterbi Decoder Block. The Viterbi Decoder block decodes the convolutional code, and the algorithm's use of traceback path causes a delay. The block processes a frame-based signal and has the **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 $4 \times 3 + 96 = 108$ samples.

Note The Outer Error Rate block accounts for the 4-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 4-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.

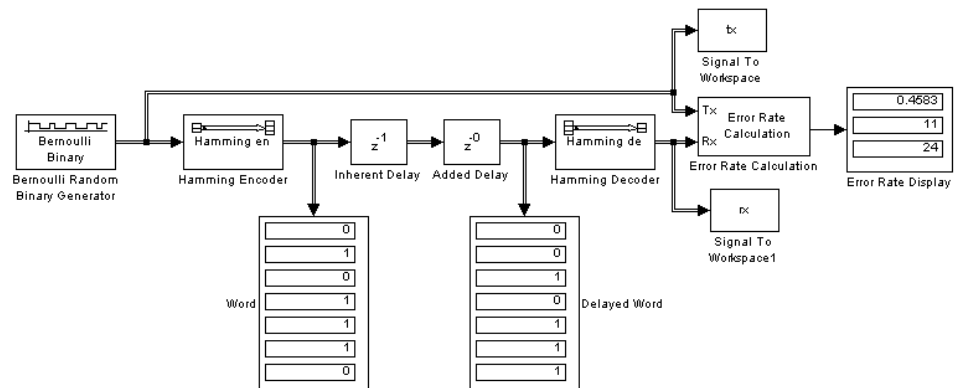
Manipulating Delays

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 code words 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 2-4). 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 2-3. 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” on page 2-14
- “Aligning Words of a Block Code” on page 2-17
- “Aligning Words for Interleaving” on page 2-19
- “Aligning Words of a Concatenated Code” on page 2-21

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 typing `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 Integer Delay block create a delay between the encoder and decoder. The two Integer 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 2-4 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 `ads1_sim` model contains an Integer 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 code word 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 code word is at the second bit in the frame, instead of coinciding with the beginning of the frame. That is, the code words 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 code word. 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, then 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 typing

```
[tx rx]
```

in the MATLAB Command Window, then 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. That is, 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. Notice that 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 code word rather than by a partial code word. Thus the decoder is correct in its assumption that a code word 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 code word length (7).

Aligning Words of a Block Code

The ADSL demo, discussed in “ADSL Demo Model” on page 2-4, illustrates the need to manipulate the delay in a model so that each frame of data that enters a block decoder has a code word 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 an Integer Delay block to manipulate the delay between the convolutional deinterleaver and the block decoder, and why the Integer Delay block is configured as it is. To open the ADSL demo model, type `adsl_sim` in the MATLAB Command Window.

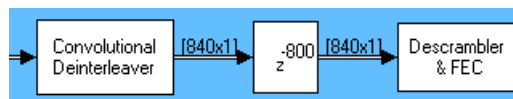
Misalignment of Code Words

Notice in the ADSL demo that 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 2-7, 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 code word. As described in “Observing the Problem” on page 2-15, this misalignment, between code words and the frames that contain them, hinders the decoder.

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 an Integer Delay block between the Convolutional Deinterleaver block and the Descrambler & FEC subsystem.



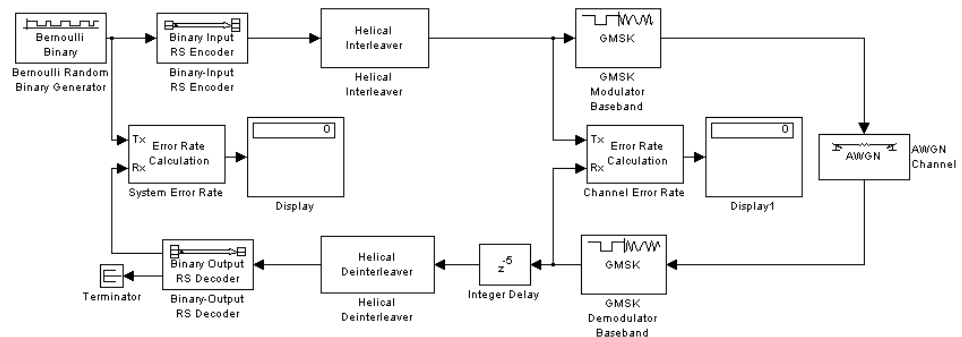
The **Delay** parameter in the Integer Delay block is 800 because that is the minimum number of samples required to shift the 41st sample of one 840-sample frame to the first sample of the next 840-sample frame. In other words, the sum of the inherent 40-sample delay (from the

interleaving/deinterleaving process) and the artificial 800-sample delay is a full frame of data, not a partial frame.

Note that 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 2-7.

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, type `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 1-97

Block	Delay, in Output Samples from Individual Block	Reference
Helical Deinterleaver	42	“Delays of Convolutional Interleavers” on page 1-77
Integer Delay	5	Integer 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 5 samples. The Integer Delay block between the GMSK Demodulator Baseband block and the Helical Deinterleaver block accomplishes such a delay. The Integer Delay block has its **Delay** parameter set to 5.

Combining the effects of the demodulator and the Integer 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 Code Words

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 an Integer Delay block between the Helical Deinterleaver block and the Binary-Output RS Decoder block.

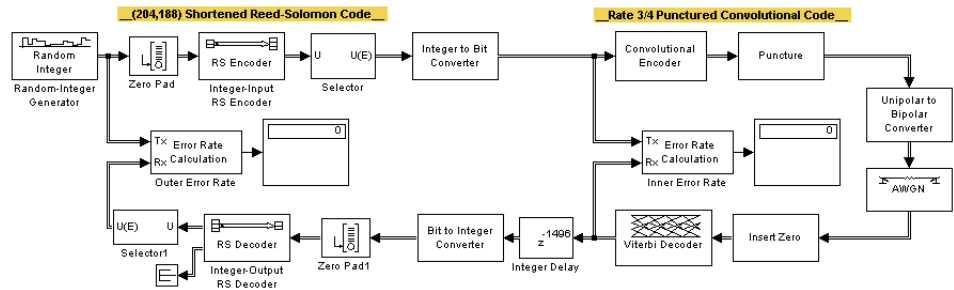
Computing Delays to Configure the Error Rate Calculation Blocks

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 table below 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 Integer 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, type `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
Integer Delay	1496 (that is, 1632 - 136)

Misalignment of Block Code Words

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 code word. 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 Integer Delay block between the Viterbi Decoder block and the Bit to Integer Converter block accomplishes such a delay. The Integer Delay block has its **Delay** parameter set to 1496.

Combining the effects of the Viterbi Decoder block and the Integer 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	188	One 188-sample frame, from the combination of the inherent delay of the Viterbi Decoder block and the added delay of the Integer Delay block

Comparing Baseband and Passband Simulation

This section uses a pair of examples to illustrate the differences between baseband and passband methods for conducting BER analysis. This section presents the passband case first because it might be more familiar to you. However, the equivalent baseband simulation, presented second, offers many advantages over the passband simulation. Compared to the passband simulation, the baseband simulation

- Takes much less time to process the same number of symbols. Furthermore, baseband simulation can use frame-based processing to make the simulation even faster.
- Achieves the same error rate.
- Is less likely to suffer from poor choices of parameters.

This discussion comprises these sections:

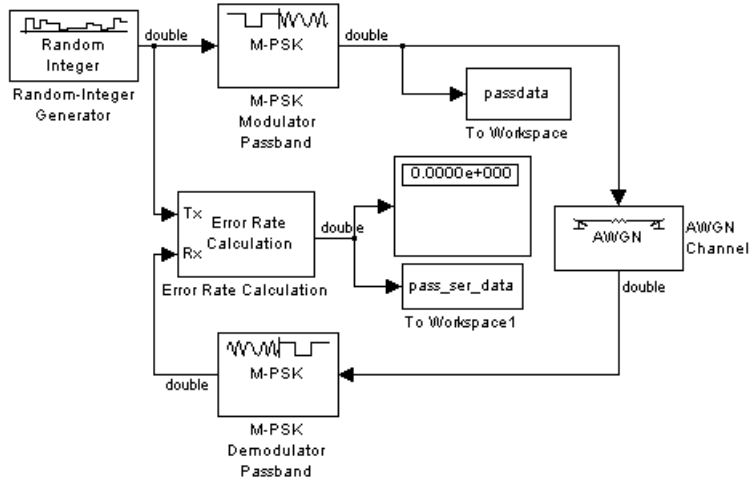
- “Running a Passband Simulation” on page 2-24
- “Running an Equivalent Baseband Simulation” on page 2-25
- “Generating Error Curves” on page 2-26
- “Speed of Baseband Versus Passband Models” on page 2-28
- “Comparing Baseband and Passband Signals” on page 2-30
- “Troubleshooting a Passband Simulation” on page 2-32

To learn the mathematical differences between baseband and passband representations of a signal, see “Baseband Modulated Signals Defined” on page 1-82 or a basic communications textbook.

Running a Passband Simulation

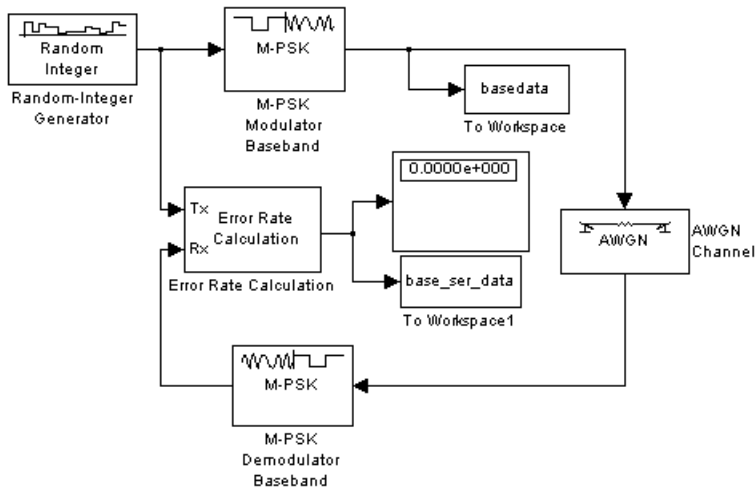
This section introduces a passband simulation model that shows the error rate of QPSK modulation over an AWGN channel with a varying noise level. The example in the next section, “Running an Equivalent Baseband Simulation” on page 2-25, achieves the same objective but runs more quickly because it uses baseband simulation.

To open the passband example model, type `pbmoddoc` in the MATLAB Command Window. Then you can run the simulation by choosing **Start** from the model window’s **Simulation** menu.



Running an Equivalent Baseband Simulation

To open a baseband simulation that computes the same error rates as in the passband model of the previous section, type `bbmoddoc` in the MATLAB Command Window. Then you can run the simulation by choosing **Start** from the model window's **Simulation** menu.



Complex Baseband Modulated Signal

Notice that the signal lines from the M-PSK Modulator Baseband block to the AWGN Channel block, and from the AWGN Channel block to the M-PSK Demodulator Baseband block have an annotation that says `double (c)`. The `(c)` portion indicates that the baseband modulated signal is a *complex* signal, not a real signal. By contrast, the unmodulated signal is a real signal. The meaning of the complex modulated signal appears in “Baseband Modulated Signals Defined” on page 1-82.

Differences Between the Passband and Baseband Examples

The differences in construction between this model and the passband model, `pbmoddoc`, of the previous section are

- This model uses baseband modulation blocks (M-PSK Modulator Baseband, M-PSK Demodulator Baseband) instead of passband modulation blocks.
- The two To Workspace blocks in this model use variable names that differ from those in the passband model, so that the data sets from the two models do not conflict with each other.
- The Error Rate Calculation block in this model uses a **Receive delay** parameter of 0 instead of 1, because baseband demodulation causes no delay, while passband demodulation causes a one-sample delay.

Performance differences between the two models are discussed in “Speed of Baseband Versus Passband Models” on page 2-28.

Generating Error Curves

To plot error rates as a function of noise level using the baseband simulation, execute the following code in MATLAB. The plot includes both the theoretical error rates and the error rates from running the simulation.

```
modelName='bbmoddoc';

EbNoVec = [0:8];
EsNoVec = EbNoVec + 10*log10(2);
numsims = length(EsNoVec);
SERVec = zeros(numsims,3);

% Set up model so it runs until 100 errors occur.
% Also, remove Display block so model runs more quickly.
```



```

load_system(modelname);
set_param(gcs,'StopTime','inf');
set_param([gcs '/Error Rate Calculation'],'stop','on');
delete_line(modelname,'Error Rate Calculation/1','Display/1')
delete_block([modelname '/Display'])

% For each noise level, run the model and save the error data.
for n = 1:numsims
    modex_EsNodB = EsNoVec(n);
    disp(['Iteration #' num2str(n) ' of ' num2str(numsims)]);
    sim(modelname);
    SERVec(n,:) = base_ser_data;
end

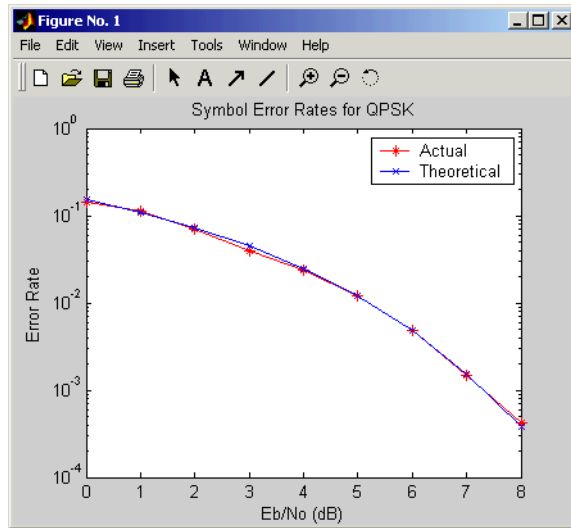
% Revert to original parameters in model.
close_system(modelname,0); % Close without saving changes.
open_system(modelname);

% Compute theoretical values.
linEbNoVec = 10 .^ (0.1 .* EbNoVec);
temp = 0.5.*erfc((sqrt(2.*linEbNoVec))./sqrt(2));
ser = temp.*(2 - temp);

% Plot actual and theoretical values on one graph.
figure;
semilogy(EbNoVec,SERVec(:,1),'r-*',EbNoVec,ser,'b-x');
legend('Actual','Theoretical');
xlabel('Eb/No (dB)'); ylabel('Error Rate');
title('Symbol Error Rates for QPSK');

```

The figure below shows the error curves. Notice that the values from the baseband simulation agree with the theoretical values.



Using the Passband Model to Generate Error Curves

If you want to generate error curves using the passband model, you can use the preceding code after making these modifications:

- Change the definition `modelName='bbmoddoc'`; to `modelName='pbmoddoc'`;
- Change the variable name `base_ser_data` to `pass_ser_data`.

Note The code might take a long time to complete its computations and produce the plot.

The resulting plot looks very similar to the plot produced using the baseband model, because baseband and passband simulation are equivalent for this purpose.

Speed of Baseband Versus Passband Models

The passband and baseband models produce error rates that differ from each other by less than 1%. However, the passband model takes a significantly longer time to process the same amount of data. Although the actual speed

depends on your system, the relative times in the tables below can serve as a guide. Each table shows the approximate clock time that simulations take to run, expressed as a multiple of the clock time that the model `bbmoddoc` takes to run. Notice these general trends:

- Baseband simulation is considerably faster than passband simulation. The difference in speed is especially dramatic when the carrier frequency in the passband simulation is high.
- Baseband simulation using large frames is faster than baseband simulation that does not use frames.

To time simulations on your own computer, use the functions `tic`, `toc`, and `sim`.

Relative Clock Times Corresponding to 1 Second of Simulation Time

Simulation	Relative Time
<code>bbmoddoc</code> (baseband)	1
<code>pbmoddoc</code> (passband)	52
<code>pbmoddoc</code> (passband), after multiplying carrier frequency and modulated signal sample time by 10 and 1/10, respectively	916

Relative Clock Times Corresponding to 3 Seconds of Simulation Time

Simulation	Relative Time
<code>bbmoddoc</code> (baseband)	1
<code>pbmoddoc</code> (passband)	100
<code>bbmoddoc</code> (baseband), using frame-based processing with 512-sample frames	0.28

Relative Clock Times to Execute Code as in “Generating Error Curves”

Simulation	Relative Time
bbmoddoc (baseband)	1
pbmoddoc (passband)	100
bbmoddoc (baseband), using frame-based processing with 512-sample frames	.30

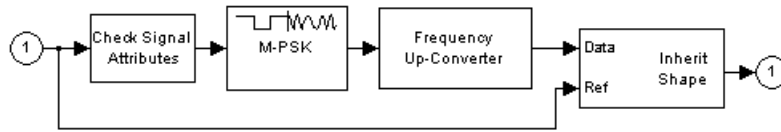
Comparing Baseband and Passband Signals

This section discusses the relationship between the baseband and passband models `pbmoddoc` and `bbmoddoc`. In particular, it shows that even though the baseband modulated signal is a complex-valued signal and not a real-valued sinusoid, it is equivalent to the real sinusoid that the passband model processes. While the section “Baseband Modulated Signals Defined” on page 1-82 gives a theoretical description of the equivalence between baseband and passband signals, this discussion uses a more hands-on approach; using the example model, this discussion shows how the baseband algorithm forms part of the passband algorithm and how the conversion from baseband to passband representation occurs.

Baseband Algorithm Within the Passband Algorithm

One way to see how the passband and baseband modulator blocks are related to each other is to look inside the passband algorithm. Open the passband model by typing `pbmoddoc` in the MATLAB Command Window and then follow these instructions to open the subsystem that the passband modulator block represents:

- 1 In the top of the model window, click the M-PSK Modulator Passband block.
- 2 From the model window’s **Edit** menu, choose **Look under mask**.

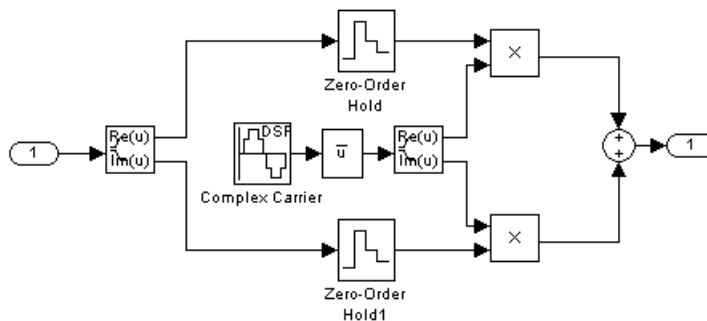


If you double-click the M-PSK icon in the subsystem and look in the title bar of the dialog box, then you can see that the block is the M-PSK Modulator Baseband block from the digital *baseband* phase modulation library of the blockset. This indicates that the baseband algorithm forms a part of the passband algorithm.

Conversion from Baseband to Passband Representation

The Frequency Up-Converter block represents another part of the passband modulation algorithm. This block converts a baseband modulated signal into an equivalent passband modulated signal. To see the conversion in more detail, follow these instructions:

- 1 In the subsystem window, click the Frequency Up-Converter block.
- 2 From the model window's **Edit** menu, choose **Look under mask**.



Studying this block diagram from left to right, you can see that it multiplies the real and imaginary parts of the baseband modulated signal by

$$\sqrt{2} \cos(2\pi f_c t + \theta)$$

and

$$-\sqrt{2} \sin(2\pi f_c t + \theta)$$

respectively, and then adds the results together. Here f_c and θ are the **Carrier frequency** and **Carrier initial phase** parameters, respectively, in the M-PSK Modulator Passband block. The result of this process is the real-valued sinusoidal signal that you would expect from a passband modulator.

Troubleshooting a Passband Simulation

Passband modulation can be difficult to use because it requires you to choose appropriate values for carrier-related parameters and because it requires Simulink to sample signals at a high sampling rate. These factors can reduce the accuracy and speed of a passband simulation. The speed is particularly noticeable if you need to process large amounts of data before the results are meaningful. These sections offer tips that might help you improve the accuracy and/or speed of your simulation:

- “Use Baseband Simulation”
- “Decrease the Sample Time” on page 2-33
- “Increase the Carrier Frequency” on page 2-34
- “Use the Simulink Accelerator to Increase Speed” on page 2-34

Use Baseband Simulation

Converting your model to a baseband simulation might improve the simulation’s accuracy and/or speed substantially. Baseband modulation blocks do not use carrier-related parameters and, therefore, do not suffer from poor choices of such parameters. If you use baseband simulation, you can safely ignore the following sections, “Decrease the Sample Time” and “Increase the Carrier Frequency”.

Also, baseband simulations usually run faster than passband simulations because baseband simulations do not involve sampling a carrier signal at a high rate. The difference in speed might be dramatic.

Frame-based processing, which is available with baseband but not passband blocks, might further speed your baseband simulation. You can typically experiment with frame-based processing by varying parameters in the source blocks, while most other blocks in the model can remain as they are. For example, you can switch to frame-based processing in the example model

bbmoddoc by changing two parameters in the Random Integer Generator block. Specifically, check the **Frame-based outputs** check box and set the **Samples per frame** parameter to an integer greater than 1 (such as 20).

Many algorithms can be simulated adequately at baseband. For example, to model the carrier frequency offset in a baseband simulation, multiply the transmitted baseband signal by a complex sinusoid. Because this sinusoid would typically have a much lower frequency than the carrier frequency, such a model is still less computationally intensive to simulate than an equivalent passband simulation would be.

Some situations require passband simulation, such as investigating the effects of radio frequency distortion. Even in such cases, you might be able to model part of the system at baseband initially and then switch to passband when you focus on the aspects of the system that require passband simulation.

Decrease the Sample Time

If you get results from a passband simulation that do not seem to match theoretical results, it could be that the sampling rate of the passband modulated signal is not sufficiently high. The sampling rate is the reciprocal of the sample time. You should decrease the sample time of the passband modulated signal, run the simulation again, and check the results. This sample time is the **Output sample time** parameter in digital passband modulator blocks and the **Input sample time** parameter in digital passband demodulator blocks.

After you decrease the sample time, the simulation might run more slowly.

Example of Excessive Sample Time of Modulated Signal. In the example model pbmoddoc, the sample time of the modulated signal is `modex_Td`, while the sample time of the unmodulated signal is `modex_Ts`. If you assign

```
modex_Td = modex_Ts/32;
```

in the MATLAB Command Window and run the simulation, the error rate shown in the Display block is over 0.0066. This error rate is more than one-third greater than it was before making this parameter change. The simulation results no longer agree with the theoretical expected results because the sample time `modex_Td` is too large compared to the sample time of the data. More specifically, the poorly chosen sample times cause aliasing of the signal spectrum in the frequency domain.

Increase the Carrier Frequency

If you get results from a passband simulation that do not seem to match theoretical results, it could be that the carrier frequency for passband modulation is not sufficiently high relative to the sampling frequency of the unmodulated signal. You should increase the **Carrier frequency** parameter of the modulator and demodulator blocks, run the simulation again, and check the results.

After you increase the carrier frequency, you might need to increase the sampling rate of the passband modulated signal to compensate.

Example of Insufficient Carrier Frequency. In the example model pbmoddoc, the carrier frequency of the modulated signal is `modex_fc`. Suppose you make the carrier frequency (`modex_fc`) four times the sampling frequency of the unmodulated signal (`modex_Ts`), and then use the inequalities on the modulation block's reference page to determine a threshold value for the modulated signal's sampling rate. For example, you might choose

```
modex_fc = 4/modex_Ts;  
modex_Td = modex_Ts/12;
```

These parameter values satisfy the inequalities and the simulation runs relatively quickly. However, the parameter values cause the model to produce an error rate of over 0.016, which is more than twice the theoretical expected result.

Note Although satisfying certain inequalities involving the passband modulation block's parameters is necessary for the block to operate, that alone is not sufficient for the block to produce meaningful results.

Use the Simulink Accelerator to Increase Speed

If you have access to the Simulink Accelerator, you can use it to make your simulation run more quickly. The Simulink Accelerator is part of the Simulink Performance Tools product.

Demonstration Models

Punctured Convolutional Coding Demo	3-2
Adaptive Equalization Demo	3-9
CPM Phase Tree Demo	3-11
GMSK vs. MSK Demo	3-14
Filtered QPSK vs. MSK	3-16
Rayleigh Fading Channel Demo	3-17
Gray Coded 8-PSK Demo	3-18
Discrete Multitone Signaling Demo	3-29
Iterative Decoding of a Serially Concatenated Convolutional Code (SCCC) - Demo	3-31
Phase Noise Effects in 256-QAM - Demo	3-36
PLL-Based Frequency Synthesis Demo	3-38
256-Channel ADSL Demo	3-50
Bluetooth Voice Transmission Demo	3-53
Digital Video Broadcasting Demo	3-56
HiperLAN/2 Demo	3-59
RF Satellite Link Demo	3-61
WCDMA Coding and Multiplexing Demo	3-70
WCDMA End-to-End Physical Layer Demo	3-71
WCDMA Spreading and Modulation Demo	3-79

Punctured Convolutional Coding Demo

The complexity of a Viterbi decoder increases rapidly with the code rate. Puncturing is a technique that allows the encoding and decoding of higher rate codes using standard rate 1/2 encoders and decoders. This example, `tstconvcod`, demonstrates how to use the Convolutional Encoder and Viterbi Decoder blocks to simulate a punctured coding system.

The example is somewhat similar to the one that appears in “Example: Soft-Decision Decoding” on page 1-60, which demonstrates convolutional coding without puncturing. The present example contains two blocks related to puncturing: Puncture and Insert Zero.

This description of the demo includes these topics:

- “Structure of the Demo” on page 3-2
- “Generating Random Data” on page 3-3
- “Convolutional Encoding” on page 3-3
- “Puncturing” on page 3-4
- “Transmitting Data” on page 3-4
- “Demodulating” on page 3-5
- “Inserting Zeros” on page 3-5
- “Viterbi Decoding” on page 3-6
- “Calculating the Error Rate” on page 3-6
- “Evaluating Results” on page 3-6
- “Bibliography” on page 3-8

Structure of the Demo

This example contains these blocks from the Communications Toolbox.

Communications Blockset Block	Purpose in Example
Bernoulli Binary Generator	Create random bits to use as message.
Convolutional Encoder	Encode message using the convolutional coding technique.

Communications Blockset Block	Purpose in Example
Puncture	Remove bits from the output of the Convolutional Encoder.
BPSK Modulator Baseband	Modulate encoded message to prepare for transmission.
AWGN Channel	Transmit data, adding random numbers to simulate a noisy channel.
Insert Zero	Insert zeros to substitute for bits removed by the Puncture block.
Viterbi Decoder	Decode the convolutional code using the Viterbi algorithm.
Error Rate Calculation	Compute the proportion of discrepancies between original and recovered messages.

You can get detailed information on each of these blocks by clicking on the name of the block above.

Generating Random Data

The Bernoulli Binary Generator block, in the Data Sources sublibrary of the Comm Sources library, produces the information source for this simulation. The block generates a frame of three random bits at each sample time. The **Samples per frame** parameter determines the number of rows of the output frame.

Note: The size of the output frame must be compatible with the length of the **Puncture vector** parameter in the Puncture block. See the section “Puncturing” for more details.

Convolutional Encoding

The Convolutional Encoder, in the Convolutional sublibrary of the Error Detection and Correction library, encodes the data from the Bernoulli Binary Generator. This demo uses the same code as detailed in “Example: Soft-Decision Decoding” on page 1-60.

Puncturing

The Puncture block, in the Sequence Operations sublibrary of the Basic Comm Functions library, carries out the puncturing. The Puncture block periodically removes bits from the encoded bit stream, thereby increasing the code rate.

The puncture pattern is specified by the **Puncture vector** parameter in the mask. The puncture vector is a binary column vector. A one indicates that the bit in the corresponding position of the input vector is sent to the output vector, while a zero indicates that the bit is removed.

For example, to create a rate 3/4 code from the rate 1/2, constraint length 7 convolutional code, the optimal puncture vector is $[1 \ 1 \ 0 \ 1 \ 1 \ 0]'$ (where the $'$ after the vector indicates the transpose). Bits in positions 1, 2, 4 and 5 are transmitted, while bits in positions 3 and 6 are removed. Now, for every 3 bits of input, the punctured code generates 4 bits of output (as opposed to the 6 bits produced before puncturing). This makes the rate 3/4.

Note In frame-based processing, the length of the puncture vector must divide the length of the input frame.

In this example, the output from the Bernoulli Binary Generator is a column vector of length 3. Since the rate 1/2 Convolutional Encoder doubles the length of each vector, the input to the Puncture block is a vector of length 6. Therefore, the length of the puncture vector must divide 6.

Transmitting Data

The AWGN Channel block, from the Channels library, simulates transmission over a noisy channel. The parameters for the block are set in the mask as follows:

- The **Mode** parameter for this block is set to **Signal to noise ratio (Es/No)** mode.
- The **Es/No** parameter is set to 2 dB. This value typically is changed from one simulation run to the next.
- The preceding modulation block generates unit power signals, so the **Input signal power** is set to 1 watt.
- The **Symbol period** is set to 0.75 seconds because the code has rate 3/4.

Demodulating

In this simulation, the Viterbi Decoder block is set to accept unquantized inputs. The BPSK Demodulator block produces hard decisions, so it cannot be used for demodulation in this model. Instead, the simulation passes the channel output through a Simulink Complex to Real-Imag block that extracts the real part of the complex samples.

Inserting Zeros

The Insert Zero block substitutes zeros for the bits that were removed by the Puncture block. Because the punctured bits are not transmitted, there is no information to indicate their values. Since BPSK is an antipodal modulation format, and 0 lies half way between +1 and -1, you can insert zeros in place of the punctured bits.

The locations of the inserted zeros is determined by the **Insert zero vector** parameter in the mask. The insert zero vector is a binary column vector, which will usually be the same as the puncture vector. Each 1 in the insert zero vector indicates that the block should place the next element of the input vector into the output vector (at the position of the 1). Each 0 indicates that the block should place a 0 into the output vector (at the position of the 0). This replaces the punctured bits with zeros.

Note In frame-based processing, the length of the **Insert zero vector** value must divide the length of an input frame.

Data Delay

In this example, there is no data delay between the Puncture block and the Insert Zero block. However, if you introduce another block into the model between the Puncture block and the Insert Zero block that produces a delay, then the Insert Zero block might insert zeros in locations other than where the Puncture block removed bits. To correct this, you should also place an Integer Delay block, in the Signal Operations library of the DSP Blockset, immediately before the Insert Zero block. Set the **Delay (samples)** parameter of the Integer Delay block to an integer such that the total delay between the Puncture block and the Insert Zero block (including the Delay block) is a multiple of the length of the **Insert zero vector** parameter.

For example, if there is a delay of 20, and the length of the insert zero vector is 6, then the **Delay (samples)** parameter should be 4. This makes the total delay 24, which is a multiple of 6, and brings the Insert Zero block into phase with the Puncture block.

Viterbi Decoding

The Viterbi Decoder block, in the Convolutional sublibrary of the Error Detection and Correction library, is configured to decode the same rate 1/2 code specified in the Convolutional Encoder block.

In this example, the decision type is set to **Unquantized**. For codes without puncturing, you would normally set the **Traceback depth** for this code to a value close to 40. However, for decoding punctured codes, a higher value is required to give the decoder enough data to resolve the ambiguities introduced by the inserted erasures.

Calculating the Error Rate

The Error Rate Calculation block, in the Comm Sinks library, compares the decoded bits to the original source bits. The output of the Error Rate Calculation block is a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

In the mask for this block, the **Receive delay** parameter is set to 96, because the **Traceback depth** value of 96 in the Viterbi Decoder block creates a delay of 96. If there were other blocks in the model that created delays, the **Receive delay** would equal the sum of all the delays.

BER simulations typically run until a minimum number of errors have occurred, or until the simulation processes a maximum number of bits. The Error Rate Calculation block uses its **Stop simulation** mode to set these limits and to control the duration of the simulation.

Evaluating Results

Generating a bit error rate curve requires multiple simulations. You can perform multiple simulations from the command line using the `sim` command. To do this:

- Change the value of the **Es/No** parameter in the AWGN Channel block mask from a constant to the variable `EsNodB`.

- Run the following code to generate the data for plotting the BER curve.

```
CodeRate = 0.75;
EbNoVec = [2:.2:10];
EsNoVec = EbNoVec + 10*log10(CodeRate);
BERVec = zeros(length(EsNoVec),3);
for n=1:length(EsNoVec),
    EsNoDB = EsNoVec(n);
    sim('tstconvcod');
    BERVec(n,:) = BER_Data;
end
```

To confirm the validity of the results, compare them to an established performance bound. The bit error rate performance of a rate $r = (n-1)/n$ punctured code is bounded above by the expression

$$P_b \leq \frac{1}{2(n-1)} \sum_{d=d_{free}}^{\infty} w_d \operatorname{erfc}(\sqrt{rd(E_b/N_0)})$$

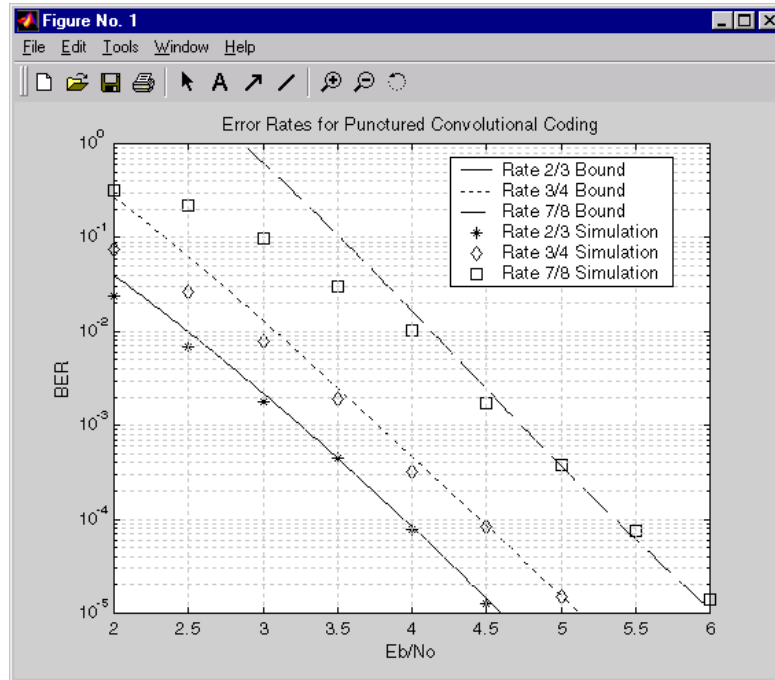
In this expression, erfc denotes the complementary error function, r is the code rate, and both d_{free} and w_d are dependent on the particular code. For the rate 3/4 code of this example, $d_{free} = 5$, $w_5 = 42$, $w_6 = 201$, $w_7 = 1492$, and so on. See reference [1] for more details.

The following commands compute an approximation of this bound in MATLAB using the first seven terms of the summation.

```
dist = [5:11];
nerr = [42 201 1492 10469 62935 379644 2253373];
CodeRate = 3/4;
EbNo_dB = [2:.02:10];
EbNo = 10.0.^(EbNo_dB/10);
arg = sqrt(CodeRate*EbNo'*dist);
bound = nerr*(1/6)*erfc(arg)';
```

The figure below shows simulation results and bounds for the rate 3/4 punctured code in this example, as well as other punctured codes of rates 2/3 and 7/8 derived from the same original constraint length 7 rate 1/2 code. The puncture patterns for these other rates are listed in reference [1]. The

simulations used to generate the data for this plot were set to stop after 1000 errors or 40 million bits, whichever came first.



In each case, the results agree well with the theoretical bounds. In some cases, at the lower bit error rates, the simulation results appear to indicate error rates slightly above the bound. This is not a result of simulation variance, since over 500 bit errors were observed at even the lowest bit error rate value. Rather, this is a result of the finite traceback depth in the decoder.

Bibliography

[1] Yasuda, Y., K. Kashiki, and Y. Hirata, "High Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding," *IEEE Transactions on Communications*, Vol. COM-32, pp. 315-319, March 1984.

Adaptive Equalization Demo

The Adaptive Equalization demo, `eq_sim`, demonstrates the behaviors of several algorithms that are commonly used in communications:

- Least Mean-Square (LMS)
- Recursive Least-Squares (RLS)
- Constant Modulus Algorithm (CMA)

To select any of these algorithms and to set up the parameters corresponding to each algorithm, double click the block in the model labeled “Initial Settings.”

The Least Mean-Square (LMS) algorithm tries to minimize the mean square error (MSE) by using instantaneous values of the error.

Both the LMS and RLS algorithms use a sequence of symbol estimation errors to drive the equalizer weight adaptation. The error is given by the difference between the equalizer's output symbol and the so-called desired symbol. The algorithms operate in one of two modes:

- Training mode, in which the desired symbol sequence exactly matches the transmitted symbol sequence (i.e., the receiver has knowledge of the transmitted data in this mode).
- Decision-directed mode, in which the "desired" symbols are derived from the output of the decision device.

In the demo, a manual switch controls these modes of operation. To toggle the mode, double-click the block.

To overcome some of the disadvantages of the LMS algorithms, researchers have proposed different modifications of the algorithms that can be used under different scenarios. Several of these algorithms have been implemented in the model: Sign LMS, Normalized LMS (NLMS), Variable Step-size LMS (VSLMS), and Leaky LMS.

The Recursive Least-Squares (RLS) algorithm uses a deterministic approach instead of stochastic as in the case of the LMS to update the coefficients. By increasing the computational complexity and risk of instability, the RLS achieves faster convergence than the LMS.

Finally, the Constant Modulus Algorithm (CMA), or Godard Algorithm, belongs to the family of blind equalization. It is mainly used when no

knowledge of the input sequence is available and only statistics of the source are known.

When the number of coefficients and the number of points in the constellation are equal to 2, the trajectory over the MSE and the CMA cost functions are presented when the simulation is stopped. To select the initial conditions over the cost function, double-click the block labeled “Plot Cost Function.”

For more information on the LMS adaptive filter or channel equalization, see the following references:

- [1] Haykin, S., *Adaptive Filter Theory*, Third Ed., Prentice Hall, 1996.
- [2] Farhang-Boroujeny, B. *Adaptive Filters – Theory and Applications*, John Wiley & Sons, 1999.
- [3] Johnson, C. R., *et al.*, “Blind Equalization Using the Constant Modulus Criterion: A Review,” *Proc. IEEE*, Vol. 86, No. 10, Oct. 1998.

CPM Phase Tree Demo

The CPM Phase Tree demo, `cpmphasetree`, illustrates a way to use the Discrete-Time Eye Diagram Scope block to view the phase trajectory, phase tree, and instantaneous frequency of a CPM modulated signal. This document highlights these aspects of the demo:

- “Structure of the demo” on page 3-11
- “Variables” on page 3-11
- “Visible Results of the Demo” on page 3-12

Structure of the demo

This demo uses various Communications Blockset, DSP Blockset and Simulink blocks to model a baseband CPM signal. The demo includes the following blocks:

- Random Integer block, which provides a source of uniformly distributed random integers in the range $[0, M-1]$, where M is the constellation size of the CPM signal
- Integer to Bit Converter block
- CPM Modulator Baseband block
- Complex to Magnitude-Angle Converter block
- Phase Unwrap block
- Zero-Order Hold block
- Discrete Transfer Function block
- Gain block
- Four copies of the Discrete-Time Eye Diagram Scope block

Variables

When the model is loaded, `cpmphasetree_init.m` is called to create several variables in the MATLAB workspace, using the `PreLoadFcn` model callback parameter.

Visible Results of the Demo

When you run the demo, the four Discrete-Time Eye Diagram Scope blocks in the model show how the CPM signal changes over time:

- The block labeled “Modulated Signal” displays the in-phase and quadrature signals. Double-click the block to open the scope.

The modulated signal is easy to see in the eye diagram only when the **Modulation index** parameter in the CPM Modulator Baseband block is set to 0.5. If you set the **Modulation index** to another value, for example $2/3$, the features of the modulated signal are difficult to decipher for this more complex modulation. Unwrapping the phase and plotting it is another way to illustrate these more complex CPM modulated signals.

- The block labeled “Phase Trajectory” displays the CPM phase. Double-click the block to open the scope.

The Phase Trajectory Eye Diagram Scope block reveals that the signal phase is also difficult to view because it drifts with the data input to the modulator.

- The block labeled “Phase Tree” displays the phase tree of the signal.

The CPM phase is processed by a few simple blocks to make the CPM pulse shaping easier to view. This processing holds the phase at the beginning of the symbol interval and subtracts it from the signal. This resets the phase to zero every three symbols. The resulting plot shows the many phase trajectories that can be taken by the signal from any given symbol epoch.

- The block labeled “Instantaneous Frequency” displays the instantaneous frequency of the signal.

The CPM phase is differentiated to produce the frequency deviation of the signal. Viewing the CPM frequency signal enables you to observe the frequency deviation qualitatively, as well as make quantitative observations, such as measuring peak frequency deviation.

Experimenting with the Demo

To learn more about the demo, try changing the following parameters in the CPM Modulator Baseband block:

- Change **Pulse length** to one of the values 1, 2, ... 6.
- Change **Frequency pulse shape** to one of the other settings, such as **Raised Cosine** or **Gaussian**.

You can observe the effect of changing these parameters on the phase tree and instantaneous frequency of the modulated signal.

GMSK vs. MSK Demo

The GMSK vs. MSK demo, `gmskvsmk`, visually compares Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes. This document highlights these aspects of the demo:

- “Structure of the Demo” on page 3-14
- “Visible Results of the Demo” on page 3-14

Structure of the Demo

This demo uses various Communications and DSP Blockset blocks to model GMSK and MSK modulation schemes. The demo includes the following blocks:

- Random Integer block, which provides a source of uniformly distributed random integers in the range $[0, M-1]$, where M is the constellation size of the GMSK or MSK signal
- Unipolar to Bipolar Converter block
- GMSK Modulator Baseband block
- MSK Modulator Baseband block
- AWGN Channel block
- Two copies of the Discrete-Time Eye Diagram Scope block
- Two copies of the Discrete-Time Signal Trajectory Scope block

Visible Results of the Demo

The demo illustrate the difference between the two modulation schemes. The Discrete-Time Eye Diagram Scope blocks show the eye diagrams of GMSK and MSK signals corrupted by noise. The eye diagrams show the similarity between the GMSK and MSK signals when you set the **Pulse length** of the GMSK Modulator Baseband block to 1. Setting the **Pulse length** to 3 or 5 enables you to view the difference that a partial response modulation can have on the eye diagram. The number of paths increases demonstrating that the CPM waveform depends on values of the previous symbols as well as the present symbol. You can change the pulse length to 2 or 4, but you should change the **Phase offset** to $\pi/4$ for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use

instrumentation that enables you to view the phase of the signal, as in described in “CPM Phase Tree Demo” on page 3-11.

Filtered QPSK vs. MSK Demo

The QPSK vs. MSK demo, `qpskvmsk`, enables you to visually compare filtered quadrature phase shift keying (QPSK) and minimum shift keying (MSK) modulation schemes. This document highlights these aspects of the demo:

- “Structure of the Demo” on page 3-16
- “Visible Results of the Demo” on page 3-16

Structure of the Demo

This demo uses various Communications and DSP Blockset blocks to model Filtered QPSK and MSK modulation schemes. The demo includes the following blocks:

- Sources of uniformly distributed random integers in the range $[0, M-1]$, where M is the constellation size of the modulation scheme. Two sources are required since QPSK is a quaternary modulation method while MSK is a binary modulation method.
- A Baseband QPSK Modulator block
- A Baseband MSK Modulator block
- FIR Interpolator block that implements raised cosine filtering
- A Unipolar to Bipolar Converter block
- An Additive White Gaussian Noise (AWGN) Channel block
- Eye Diagram blocks

Visible Results of the Demo

The demo include these visual aids to illustrate the difference between the two modulation schemes:

Eye Diagram blocks that show the eye diagrams of Filtered QPSK and MSK signals plus noise. In FQPSK, the value of both the inphase and quadrature components of the signal are permitted to change at any symbol interval. However, for MSK, the symbol interval is half that for QPSK but the inphase and quadrature components change values in alternate symbol epochs. Therefore, the ideal sampling time for QPSK is 0.5, 1.5, 2.5, ... while the ideal sampling period for MSK is 0.5, 1.5, 2.5, ... for the inphase signal and 1, 2, 3 ... for the quadrature signal.

Rayleigh Fading Channel Demo

The Rayleigh Fading model, `rayleighfading`, illustrates the effect of multipath Rayleigh fading on a signal modulated by quadrature phase shift keying (QPSK). The following aspects of the demo are described:

- “Structure of the Demo”
- “Visible Results”

Structure of the Demo

This demo uses various Communications and DSP Blockset blocks to model multi-path Rayleigh fading. The demo includes the following blocks:

- Random Integer block, which provides source of uniformly distributed random integers in the range $[0, M-1]$, where M is the M -ary number
- QPSK Modulator Baseband
- Multipath Rayleigh Fading Channel
- FIR Interpolation block, which implements raised cosine filtering
- Discrete-Time Scatter Plot Scope

Visible Results

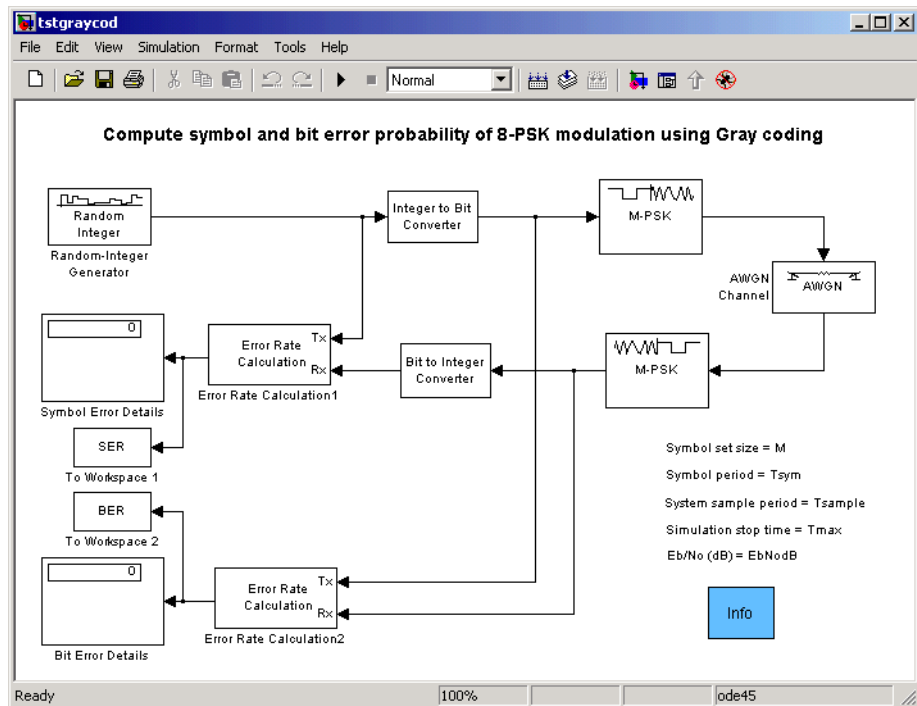
The scatter plot illustrates the effect of fading on the signal constellation. The channel is presently set to contain two paths. However, you can change this by varying the number of elements and their values in the delay and gain vectors. In addition, the **Maximum Doppler shift (Hz)** parameter in the Multipath Rayleigh Fading Channel block changes the fading pattern. The amplitude values for each path are drawn from the Rayleigh distribution while the Doppler values are drawn from the Doppler spectrum from the Jakes channel model.

Gray Coded 8-PSK Demo

Gray coding is a technique often used in multilevel modulation schemes to minimize the bit error rate by ordering modulation symbols so that the binary representations of adjacent symbols differ by only one bit. This demo simulates a communications link using Gray-coded 8-PSK modulation.

The sections that follow discuss the execution of the model and the variables used in the model. The section “Learning More About the Gray Coding Demo” on page 3-26 discusses some ways you can modify the model and compare its results with theoretical values.

Note For more information about how parts of the model work, click on blocks within the figure below.



How the Model Executes

This model executes in the following sequence:

- 1 The Random Integer Generator block serves as the source, producing a sequence of integers.
- 2 The Integer to Bit Converter block converts each integer into a corresponding binary representation.
- 3 The M-PSK Modulator Baseband block modulates the data in complex envelope format, using a Gray-coded constellation ordering.
- 4 The AWGN Channel block adds white Gaussian noise to the modulated data.
- 5 The M-PSK Demodulator Baseband block demodulates the corrupted data.
- 6 The Bit to Integer Converter block converts each binary representation to a corresponding integer.
- 7 One copy of the Error Rate Calculation block (labeled Error Rate Calculation1 in this model) compares the demodulated *integer* data with the original source data, yielding symbol error statistics.
- 8 Another copy of the Error Rate Calculation library block (labeled Error Rate Calculation2 in this model) compares the demodulated *binary* data with the binary representations of the source data, yielding bit error statistics.

Variables in the Model

Loading this model automatically defines in the MATLAB workspace five variables that are used in the demo's blocks and subsystems. To clarify the

discussion of the individual blocks and subsystems, the variables' meanings and values are listed in the table below:

Table 3-1: Preset Variables in the Gray Coding Demo

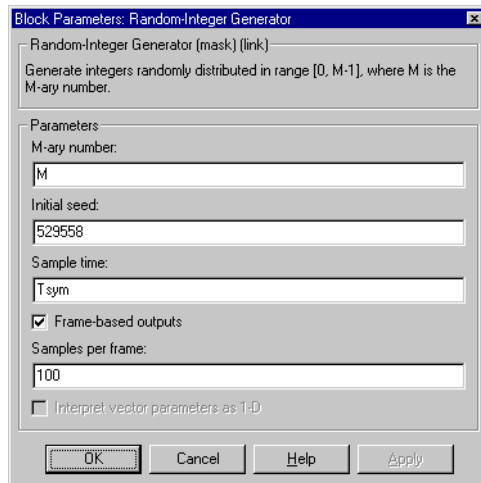
Name	Meaning	Value
M	Symbol set size	8
Tsym	Symbol period	0.2 s
Tsample	Sample period	0.01 s
Tmax	Simulation stop time	10,000 s
EbNodB	Ratio of energy per bit to noise power spectral density (E_b/N_o)	3 dB

Components of the Gray Coding Demo

This section discusses the purpose, behavior, and relevant parameters of each top-level component within the demo model. This section covers components in the order in which they process data in the simulation.

Random Integer Generator

The Random Integer Generator block produces random data that is used as the information in this simulation. This block generates one 100-symbol frame of integers in the range 0 to M-1 every Tsym seconds.



Integer-to-Bit Conversion

The Integer to Bit Converter block converts integer symbols to their equivalent binary representations. Its parameter is the number of bits in each integer.

Gray Coded M-PSK Modulation

The M-PSK Modulator Baseband block:

- Accepts binary-valued inputs that represent integers between 0 and $M-1$
- Maps binary representations to constellation points using a Gray-coded ordering
- Produces unit-magnitude complex phasor outputs, with evenly spaced phases between 0 and $2\pi(M-1)/M$

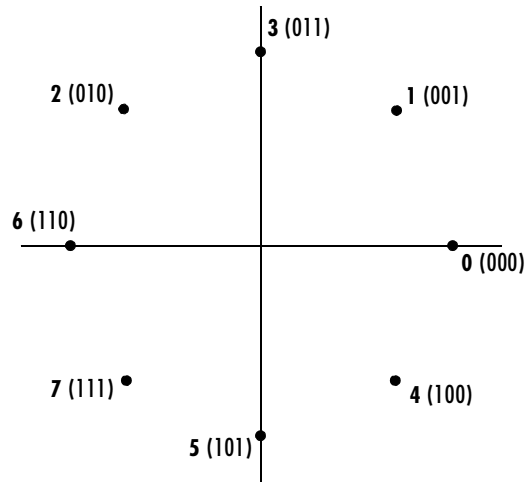
The table indicates which binary representations in the input correspond to which phasors in the output. The second column of the table is an intermediate representation that the block uses in its computations.

Modulator Input	Gray-Coded Ordering	Modulator Output
000	0	e^0
001	1	$e^{j\pi/4}$
010	3	$e^{j3\pi/4}$
011	2	$e^{j\pi/2} = e^{j2\pi/4}$
100	7	$e^{j7\pi/4}$
101	6	$e^{j3\pi/2} = e^{j6\pi/4}$
110	4	$e^{j\pi} = e^{j4\pi/4}$
111	5	$e^{j5\pi/4}$

The table below sorts the first two columns of the table above, according to the *output* values. This sorting makes it clearer that the overall effect of this subsystem is a Gray code mapping as shown in the figure below. Notice that the numbers in the second column of the table below appear in counterclockwise order in the figure.

Modulator Output	Modulator Input
e^0	000
$e^{j\pi/4}$	001
$e^{j\pi/2} = e^{j2\pi/4}$	011
$e^{j3\pi/4}$	010
$e^{j\pi} = e^{j4\pi/4}$	110
$e^{j5\pi/4}$	111

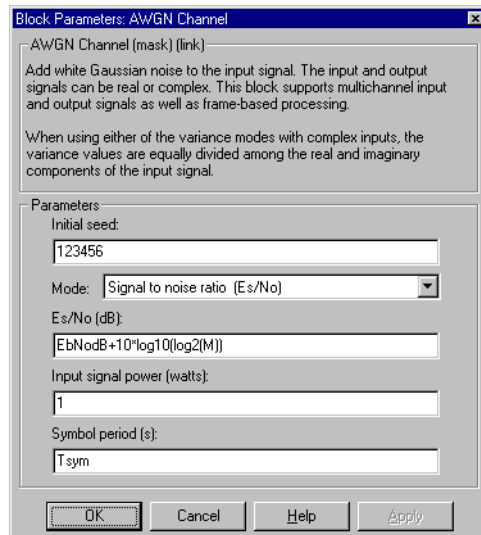
Modulator Output	Modulator Input
$e^{j3\pi/2} = e^{j6\pi/4}$	101
$e^{j7\pi/4}$	100



AWGN Channel

The AWGN Channel library block simulates transmission over a noisy channel. Its **Signal to noise ratio (Es/No)** mode uses these quantities to determine the variance:

- E_s/N_o , the ratio of energy per symbol to noise power spectral density
- The input signal power
- The symbol period



The values for these parameters are chosen as follows:

- The **Es/No** parameter is computed from the workspace variables $E_b N_{odB}$ and M . The conversion from bit energy to symbol energy reflects the fact that each symbol carries $\log_2(M)$ bits of information.
- The signal power is 1 watt because the M-PSK Modulator Baseband block produces unit power signals.
- The symbol period of the channel is set to T_{sym} .

Gray Coded MPSK Demodulation

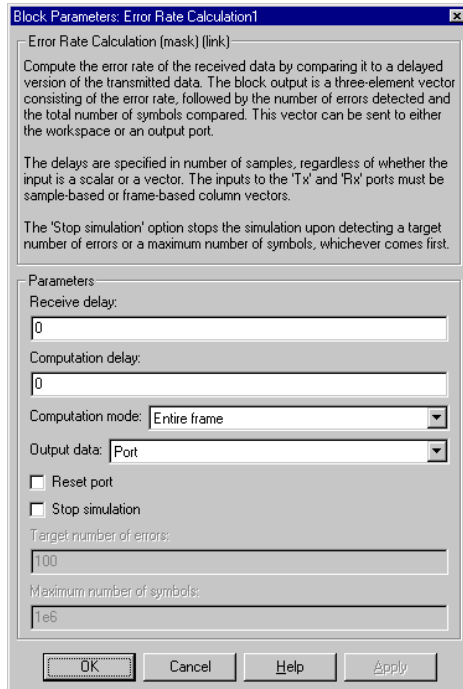
The M-PSK Demodulator Baseband block mirrors the Gray-coded modulation process. Notice that corresponding parameters in the modulator and demodulator blocks have the same values.

Bit-to-Integer Conversion

The Bit to Integer Converter block converts binary representations of symbols to their integer equivalents. Its parameter is the number of bits in each integer.

Error Rate Calculation

The Error Rate Calculation block compares demodulated symbols to original source symbols to compute the error rate. This model uses two Error Rate Calculation blocks, one to compute the symbol error rate and the other to compute the bit error rate. Both blocks use the parameter values shown below.



Symbol Error Details and Bit Error Details

Simulink's Display block shows the running error statistics throughout the simulation. Each Display block in the diagram lists three numbers, which represent:

- The symbol or bit error rate
- The total number of errors
- The total number of comparisons that the Error Rate Calculation block made

Sending Data to the MATLAB Workspace

Simulink's To Workspace block sends the complete set of error statistics to the MATLAB workspace. When the simulation ends, the MATLAB variables SER and BER are both three-column matrices whose columns represent these quantities at each time step:

- The symbol or bit error rate
- The total number of errors
- The total number of comparisons that the Error Rate Calculation block made

Learning More About the Gray Coding Demo

To learn more about a particular library block in this model, see its reference page in the Communications Blockset documentation. If you have the model open, then you can click the **Help** button in the block's dialog box to display the reference page.

The rest of this section indicates how you can analyze the data that the demo produces to compare theoretical performance with simulation performance.

Data Analysis Using the Demo

The theoretical symbol error probability of MPSK is given by

$$P_E(M) = \operatorname{erfc}\left(\sqrt{\frac{E_s}{N_0}} \sin\left(\frac{\pi}{M}\right)\right)$$

where erfc is the complementary error function, E_s/N_0 is the ratio of energy in a symbol to noise power spectral density, and M is the number of symbols.

To determine the bit error probability, the symbol error probability, P_E , needs to be converted to its bit error equivalent. There is no general formula for the symbol to bit error conversion. Upper and lower limits are nevertheless easy to establish. The actual bit error probability, P_b , can be shown to be bounded by

$$\frac{P_E(M)}{\log_2 M} \leq P_b \leq \frac{M/2}{M-1} P_E(M)$$

The lower limit corresponds to the case where the symbols have undergone Gray coding. The upper limit corresponds to the case of pure binary coding.

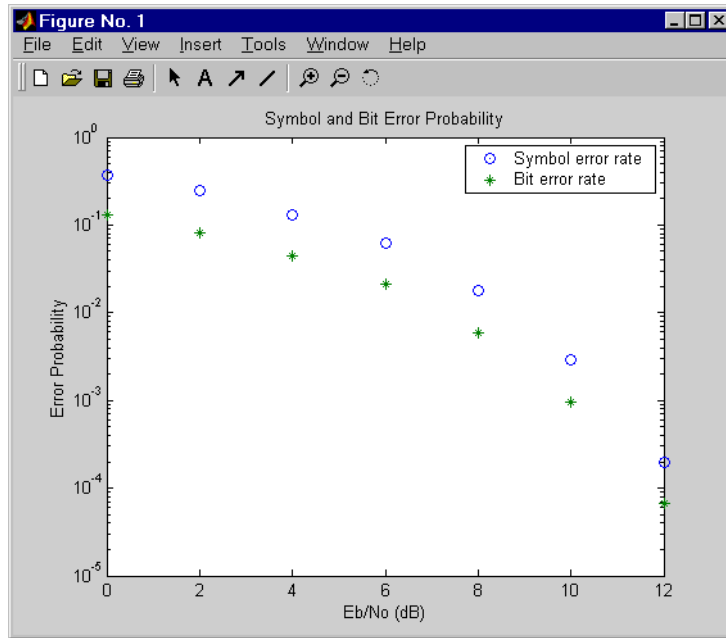
Simulation Results. To test the Gray code modulation scheme in this model, simulate the `tstgraycod` model for a range of E_b/N_o values. Because increasing the value of E_b/N_o lowers the number of errors produced, the length of each simulation must be increased to ensure that the statistics of the errors remain stable.

Using the `sim` command to run a Simulink simulation from the MATLAB command window, the following code generates data for symbol error rate and bit error rate curves. It considers E_b/N_o values in the range 0 dB to 12 dB, in steps of 2 dB.

```
M = 8;
Tsym = 0.2;
Tsample = 0.01;
BERVec = [];
SERVec = [];
EbNoVec = [0:2:12];
TVec = [1000 1000 1000 15000 20000 100000 100000]*Tsym;
for n=1:length(EbNoVec);
    Tmax = TVec(n);
    EbNodB = EbNoVec(n);
    sim('tstgraycod');
    SERVec(n,:) = SER;
    BERVec(n,:) = BER;
end;
```

After simulating for the full set of E_b/N_o values, you can plot the results using these commands:

```
semilogy(EbNoVec,SERVec(:,1),'o',EbNoVec,BERVec(:,1),'*');
legend('Symbol error rate','Bit error rate');
xlabel('Eb/No (dB)'); ylabel('Error Probability');
title('Symbol and Bit Error Probability');
```



Comparison with Pure Binary Coding and Theory. As a further exercise, you can plot the theoretical curves on the same axes with the simulation results. You can also compare Gray coding with pure binary coding, by modifying the M-PSK Modulator Baseband and M-PSK Demodulator Baseband blocks so that their **Constellation ordering** parameters are **Binary** instead of **Gray**.

Discrete Multitone Signaling Demo

The Discrete Multitone Signaling demo, `dmt_sim`, models a modulation technique that is part of the asymmetric digital subscriber line (ADSL) technology for transmitting data and multimedia information over telephone lines. The discrete multitone (DMT) signaling technique divides the channel into many subchannels and modulates each one individually. This document highlights these aspects of the demo:

- Structure of the demo, and use of Communications Blockset blocks
- An alternative model for the DMT technique, `dmt_sim_alt`

Structure of the Demo

This demo uses various Communications Blockset blocks to model DMT signaling. To see how the modulation or demodulation blocks are arranged, first open the DMT Modulator or DMT Demodulator systems at the top level of the model, and then look under the mask of the Modulator Bank or Demodulator Bank subsystems. Notice that each of the 16 Modulator Bank icons represents a set of 16 Rectangular QAM Modulator Baseband blocks. The DMT technique allocates different numbers of bits to different subchannels. Each copy of the modulator block acts as a distinct subchannel, and uses the 256-element vector `b` in the MATLAB workspace to determine the **M-ary number** parameter that is appropriate for that subchannel.

The demo also includes:

- A plot of the number of bits that each of the 256 subchannels transmits. To see this plot, double-click on the icon labeled, “Load and Plot Bit Allocation Vector.”
- A plot of the spectrum of the transmitted signal.
- A display icon that shows the bit error rate, the number of bit errors, and the total number of bits processed.
- Frame-based processing, so that the simulation processes many bits in each time step. The double connector lines between blocks indicate frame-based signals.

Discrete Multitone Signaling Demo, Alternative Form

The model `dmt_sim_alt` illustrates an alternative way to model discrete multitone signaling. Because it uses fewer blocks, it loads and initializes more quickly. To see how the alternative version uses fewer blocks, compare the alternative DMT Modulator subsystem with the original DMT Modulator subsystem.

Original: 256 Modulator Blocks

In the original form, each of 16 Modulator Bank icons represents a set of 16 modulator blocks. The system has 256 modulator blocks in total. This arrangement closely resembles the definition of 256-channel DMT signaling.

Alternative: Ten Modulator Blocks

In the alternative form, ten modulator blocks implement the ten different signal constellations in this modulation scheme. The system sends selected bits to the modulator block that is appropriate for them. This approach deviates from the specified definition of frame-based signals, however, because a frame of bits that enters one of the modulator blocks is not a set of successive samples from a time series. If you use an approach like this in your own models, first be sure that you understand the possible implications. (Refer to the online documentation for the Communications Blockset for more information about the definition of frame-based signals.)

For more information about other aspects of `dmt_sim_alt`, see “Discrete Multitone Signaling Demo” and the original `dmt_sim` demo.

Selected Bibliography

[1] Maxwell, Kim. “Asymmetric Digital Subscriber Line: Interim Technology for the Next Forty Years.” *IEEE Communications Magazine*, October 1996. 100-106.

Iterative Decoding of a Serially Concatenated Convolutional Code (SCCC) - Demo

Note This demo presents technology covered under U.S. Patent Number 6,023,783, “Hybrid concatenated codes and iterative decoding,” assigned to the California Institute of Technology. The end user of this product is hereby granted a limited license to use this demo solely for the purpose of assessing possible commercial and educational applications of the technology. Any other use or modification of this demo may constitute a violation of this and/or other patents.

The `sccc_sim` demo illustrates how to use an iterative process to decode a serially concatenated convolutional code. This document highlights these aspects of the demo:

- Structure of the demo
- Creating a serially concatenated code
- Decoding using an iterative process
- Visible results of the demo

Structure of the Demo

To summarize briefly, the simulation generates information bits, encodes them using a serially concatenated convolutional code, and transmits the coded information along a noisy channel. The simulation then decodes the received coded information using an iterative decoding process, and computes error statistics based on different numbers of iterations. Throughout the simulation, the error rates appear in a Display block.

Variables in the Demo

When you open the demo, it loads several variables into the MATLAB workspace. Note that this operation overwrites variables in the workspace that have the same names. If you accidentally delete this model’s variables and need to recreate them, open the Global Parameters block’s mask and then press **OK**.

The Global Parameters block lets you vary the values of some variables that the model uses. The table below indicates their names and meanings.

Name	Meaning
E_b/N₀	E_b/N_0 for channel noise, measured in dB; used to compute the variance of the channel noise
Block size	The number of bits in each frame of uncoded data
Number of iterations	The number of iterations to use when decoding
Seed	The initial seed in the Random Interleaver and Random Deinterleaver blocks

Creating a Serially Concatenated Code

The encoding portion of the demo uses a Convolutional Encoder block to encode a data frame, a Random Interleaver block to shuffle the bits in the code words, and another Convolutional Encoder block to encode the interleaved bits. Because these blocks are connected in series with each other, the resulting code is called a serially concatenated code.

Together, these blocks encode the 1024-bit data frame into a 3072-bit frame representing a concatenated code. These sizes depend on the model's **Block size** parameter (See the Global Parameters block.). The code rate of the concatenated code is $1/3$.

In general, the purpose of interleaving is to protect code words from burst errors in a noisy channel. A burst error that corrupts interleaved data actually has a small effect on each of several code words, rather than a large effect on any one code word. The smaller the error in an individual code word, the greater the chance that the decoder can recover the information correctly.

Convolutional Encoding Details

The two instances of the Convolutional Encoder block use their **Trellis structure** parameters to specify the convolutional codes. The table below lists

the polynomials that define each of the two convolutional codes. The second encoder has two inputs and uses two rows of memory registers.

	Outer Convolutional Code	Inner Convolutional Code
Generator Polynomials	$1 + D + D^2$ and $1 + D^2$	First row: $1 + D + D^2$, 0, and $1 + D^2$ Second row: 0, $1 + D + D^2$, and $1 + D$
Feedback Polynomials	$1 + D + D^2$	$1 + D + D^2$ for each row
Constraint Lengths	3	3 for each row
Code Rate	1/2	2/3

Decoding Using an Iterative Process

The decoding portion of this demo consists of two APP Decoder blocks, a Random Deinterleaver block, and several other blocks. Together, these blocks form a loop and operate at a rate six times that of the encoding portion of the demo. The loop structure and higher rate combine to make the decoding portion an iterative process. Using multiple iterations improves the decoding performance. You can control the number of iterations by setting the **Number of iterations** parameter in the model's Global Parameters block. The default number of iterations is six.

Computations in Each Iteration

In each iteration, the decoding portion of the demo decodes the inner convolutional code, deinterleaves the result, and decodes the outer convolutional code. The outer decoder's $L(u)$ output signal represents the updated likelihoods of original message bits (that is, input bits to the outer encoder).

The looping strategy in this demo enables the inner decoder to benefit in the next iteration from the outer decoder's work. To understand how the loop works, first recall the meanings of these signals:

- The outer decoder's $L(c)$ output signal represents the updated likelihoods of code bits from the outer encoder
- The inner decoder's $L(u)$ input represents the likelihoods of input bits to the inner encoder

The feedback loop recognizes that the primary distinction between these two signals is in the interleaving operation that occurs between the outer and inner encoders. Therefore, the loop interleaves the $L(c)$ output of the outer decoder to replicate that interleaving operation, delays the interleaved data to ensure that the inner decoder's two input ports represent data from the same time steps, and resets the $L(u)$ input to the inner decoder to zero after every six iterations.

Results of the Iterative Loop

The result of decoding is a 1024-element frame whose elements indicate the likelihood that each of the 1024 message bits was a zero or a one. A nonnegative element indicates that the message bit was probably a one, and a negative element indicates that the message bit was probably a zero. The Hard Decision block converts nonnegative and negative values to ones and zeros, respectively, so that the results have the same form as the original uncoded binary data.

Visible Results of the Demo

The demo includes a large Display block that shows error rates after comparing the received data with the transmitted data. The number of error rates in the display is the number of iterations in the decoding process. The first error rate reflects the performance of a decoding process that uses one iteration, the second error rate reflects the performance of a decoding process that uses two iterations, and so on. The series of error rates shows that the error rate generally decreases as the number of iterations increases.

Selected Bibliography

[1] Benedetto, S., D. Divsalar, G. Montorsi, and F. Pollara. "Serial Concatenation of Interleaved Codes: Performance Analysis, Design, and Iterative Decoding." *JPL TDA Progress Report*, vol. 42-126, August 1996. [This electronic journal is available at http://tmo.jpl.nasa.gov/tmo/progress_report/42-126/title.htm.]

[2] Divsalar, Dariush and Fabrizio Pollara. *Hybrid Concatenated Codes and Iterative Decoding*. U. S. Patent No. 6,023,783, Feb. 8, 2000.

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

Phase Noise Effects in 256-QAM - Demo

The `phasenoise_sim` demo illustrates the effect of a receiver's phase noise on 256-ary quadrature amplitude modulation (QAM). A QAM modulation scheme with a large number of constellation points is relatively sensitive to phase noise. This document highlights these aspects of the demo:

- Overall structure of the demo
- Visible results of the demo

Structure of the Demo

This demo uses various Communications Blockset blocks to model a QAM transceiver with phase noise. The demo contains only a small number of blocks, including:

- A source of integers between 0 and 255
- A baseband 256-QAM modulator
- An additive white Gaussian noise (AWGN) channel
- A source of phase noise
- A baseband 256-QAM demodulator
- An error statistic calculator
- A display icon that shows the error statistics while the simulation runs
- A scatter plot that shows the received signal, including the phase noise

Phase Noise Block

The Phase Noise block shifts the phase of the received signal by a random amount. You can adjust the variance of the random phase shift by adjusting the **Phase noise level** parameter in the Phase Noise block's mask.

Visible Results of the Demo

The demo includes these visual ways to understand its performance:

- A display icon that shows the running error statistics for the system. These statistics are the error rate, the number of errors detected, and the total number of symbols compared.

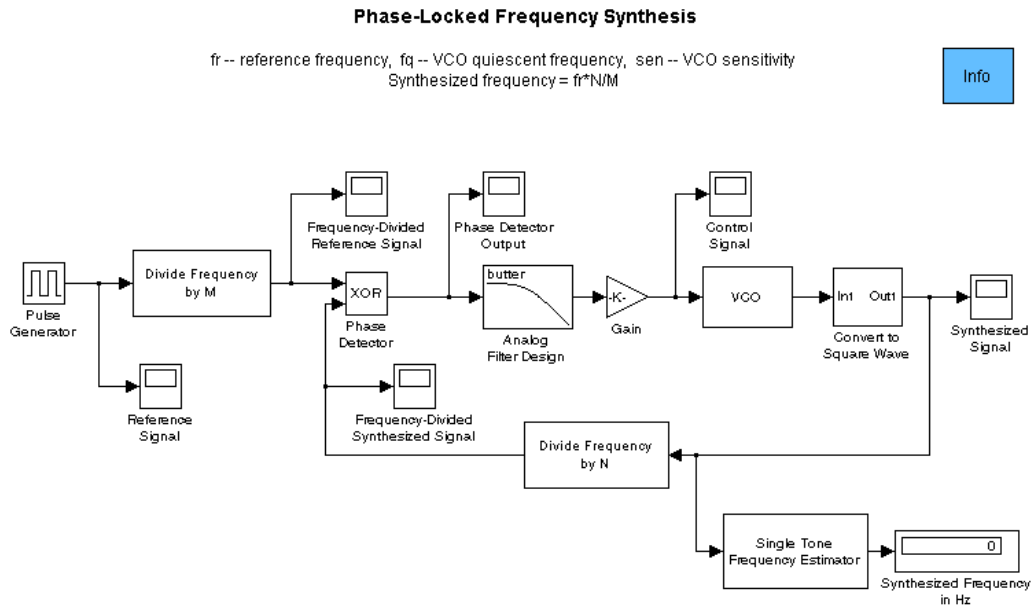
- A scatter plot that shows the received signal, including both the white Gaussian noise and the phase noise. Near each constellation point is a cluster of points. Near constellation points that are far from zero, the cluster is close to an arc. The arc shape is an effect of phase noise.
- A figure that shows bit error rates for this system with various levels of phase noise. To see the figure, double-click on the Display Figure icon in the demo. Each curve in the plot shows the bit error rate as a function of E_b/N_o in the AWGN channel, for a fixed amount of phase noise.

To create plots like this yourself, you can run the simulation multiple times, varying the parameters and recording the numerical results. An efficient way to do this is to replace key parameters in the model with variables, insert a To Workspace block for recording error statistics, and then to run the simulation using a loop in a MATLAB script. For more information about this technique, see the `sim` function, and the “Learning More About the Gray Coding Demo” section.

PLL-Based Frequency Synthesis Demo

This example shows how to simulate a phase-locked loop (PLL) frequency synthesizer. The model multiplies the frequency (f_r) of a reference signal by a constant N/M , to produce a synthesized signal whose frequency is $f_r * N/M$. A feedback loop maintains the frequency of the synthesized signal at this level.

To open the model, type `freqsyn_sim` at the MATLAB prompt (or click here if you are reading this in the MATLAB **Help** browser). In addition to the model window, three Scope windows open, labelled “Control Signal”, “Synthesized Signal” and “Reference Signal”.



Variables in the Model

When you load the model, it creates several variables, using the `PostLoadFcn` model callback parameter. Besides the variables N and M , there are:

- f_r = frequency of the reference signal

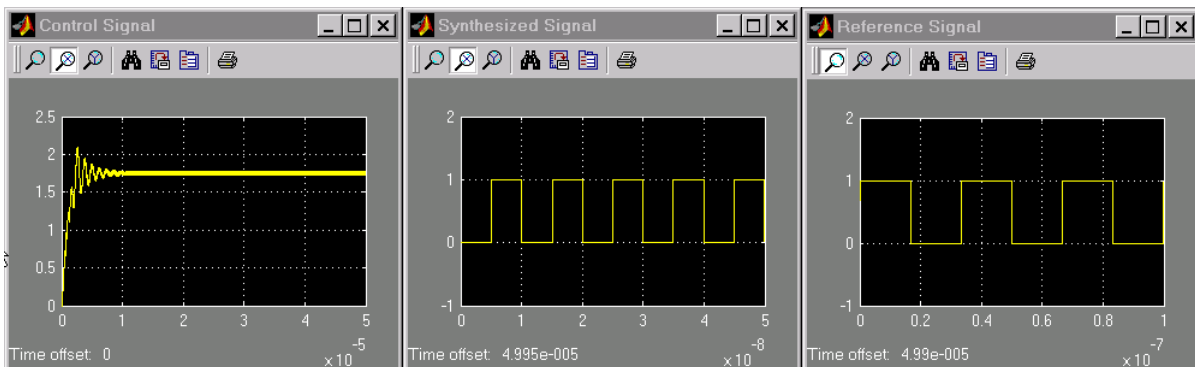
- f_q = quiescent frequency in the Voltage-Controlled Oscillator (VCO) block
- sen = Voltage-Controlled Oscillator input sensitivity

The model initially assigns values to these variables as follows: $N = 10$, $M = 3$, $f_r = 30$ MHz, $f_q = 30$ MHz and $sen = 40$ MHz/V. The frequency of the synthesized signal will then be 100 MHz. After running the simulation with these values, you can later change them by typing new values at the MATLAB prompt, if you want to experiment with the model.

Note These are the same variables as those in the Fractional N-Frequency Synthesis demo, but they are assigned different initial values. If you change the values of these variables at the MATLAB command line, use the same upper and lower case letters in the variable names as given above.

Running the Simulation

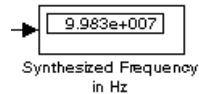
When you run the simulation, you see signals appear in the three Scope windows, as shown below.



The control signal, which the Voltage-Controlled Oscillator block uses to maintain the frequency of the synthesized signal, initially fluctuates for about 10 microseconds, but then stabilizes to a constant value of $7/4$. This occurs when the model reaches a steady state – that is, when the frequency of the synthesized signal is close to 100 MHz. Similarly, the synthesized signal oscillates back and forth at first, but then stabilizes to a square pulse of

frequency 100 MHz. The reference signal is a square pulse of frequency 30 MHz.

The Display block at the lower right of the model window displays the frequency of the synthesized signal, as shown in the figure below.



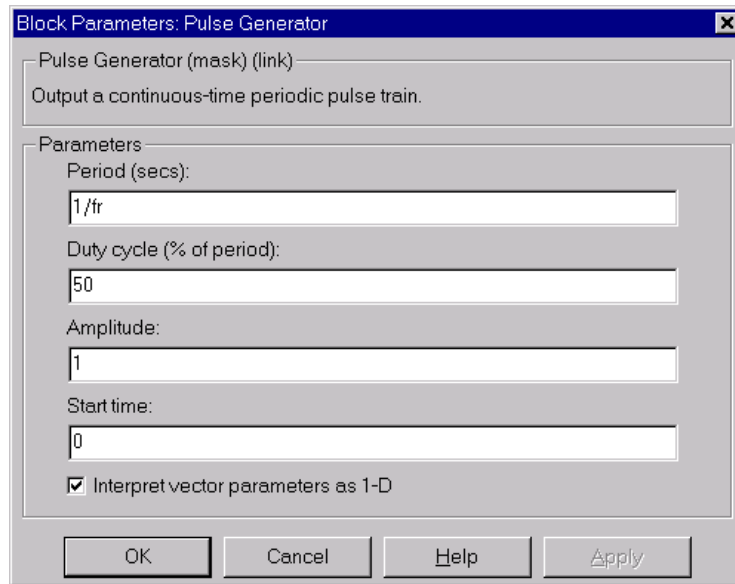
Blocks in the Model

The following table lists the most important blocks in the model and describes their purpose.

Block	Purpose in Example
Pulse Generator	Generates the reference signal, which is a periodic pulse train.
Logical Operator (XOR)	XOR's the frequency-divided reference signal with the frequency-divided synthesized signal.
Analog Filter Design	Uses a lowpass Butterworth filter to generate the control signal (along with the Gain block) and filter out high frequencies.
Gain	Multiplies the signal by a constant.
Voltage-Controlled Oscillator	Controls the frequency of the synthesized signal, by means of the input control signal.

Pulse Generator

The Pulse Generator block, from the Simulink Sources library, generates the reference signal. The block produces a periodic pulse train. Double-click on the block to open the mask, as shown below.



The variable f_r , initially set to 30 MHz, denotes the frequency of the pulse train. The period of the pulse train is $1/f_r$, which you can see in the **Period** parameter box.

The recommended way to change the value of the period is to change the value of the variable f_r in the MATLAB Command Window. This way the new value of f_r will be updated in all the blocks whose parameters are set using the variable f_r .

Divide Frequency by M

The Divide Frequency by M subsystem divides the frequency of the reference signal by the variable M. With the default values of the variables, the output of the block is a pulse train of frequency 10 MHz, called the *frequency-divided reference signal*.

Notice that there is also a Divide Frequency by N subsystem, which divides the frequency of the synthesized signal by the variable N. The output of this subsystem is called the *frequency-divided synthesized signal*.

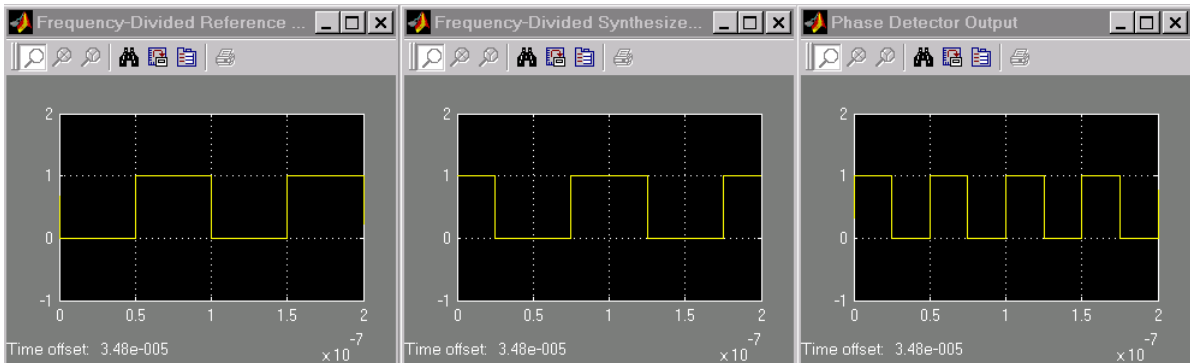
You can change the divisor in these subsystems by changing the value of M or N at the MATLAB prompt.

Phase Detector

The Logical Operator block, from the Simulink Math library, functions as a phase detector. It compares the frequencies of the frequency-divided reference signal and the frequency-divided synthesized signal. Since the block's **Operator** parameter is set to XOR in the mask, the output signal is 0 where the two input signals are equal, and 1 where they are not equal.

At steady state, the signal is a pulse train with frequency of 20 MHz. The reason for this is that both inputs to the block have a frequency of 10 MHz, but they are out of phase by 1/4 of their period. As a result, the XOR'ed signal is a periodic pulse train with frequency 20 MHz.

You can compare the two input signals to the Phase Detector with the output signal by clicking on their respective Scope blocks, as shown below.



Analog Filter Design

The Analog Filter Design block filters high frequencies out of the signal coming from the phase detector. The block uses a lowpass Butterworth filter. You can use a higher order filter or another filter type to improve the stability of the synthesized signal.

In the steady state of the model, the amplitude of the block's output signal will be approximately constant, with a value of .5. This is the average value of the output from the phase detector.

Gain Block

A Gain block, from the Simulink Math library, multiplies the output signal from the Analog Filter Design block by a constant to produce the control signal.

The **Gain** parameter in the mask is set to $\left(f_r \cdot \frac{N}{M} - f_q \right) \cdot \frac{2}{s_{en}}$. This expression

ensures that when the model is at steady state, the frequency of the synthesized signal will remain at 100 MHz, even if you make changes to the variables f_q and s_{en} .

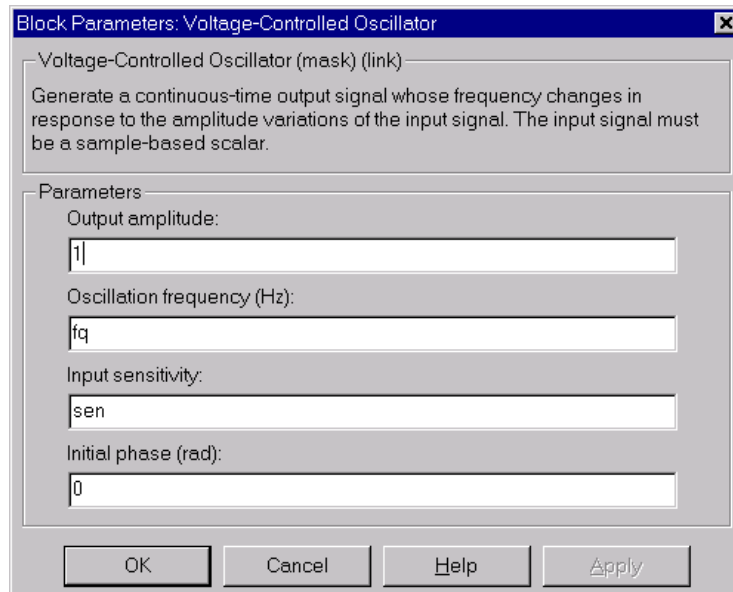
For the default values of the variables, the gain is equal to $7/2$. Thus, in the steady state of the model, the output of the Gain block is approximately constant, with a value of $7/4$.

Voltage-Controlled Oscillator

The Voltage-Controlled Oscillator block generates the synthesized signal (along with Convert to Square Wave subsystem) and adjusts the frequency of the synthesized signal according to the Voltage-Controlled Oscillator input signal.

When the control signal is close to its steady state value of $7/4$, the Voltage-Controlled Oscillator block generates a signal whose frequency is close to $f_r \cdot N/M$, which is 100 MHz for the model's pre-assigned parameters. If the output frequency drops, the control signal will rise, boosting the frequency of the output signal. If the output frequency rises, the control signal will fall, lowering the output frequency.

Double-click on the Voltage-Controlled Oscillator block to open the mask.



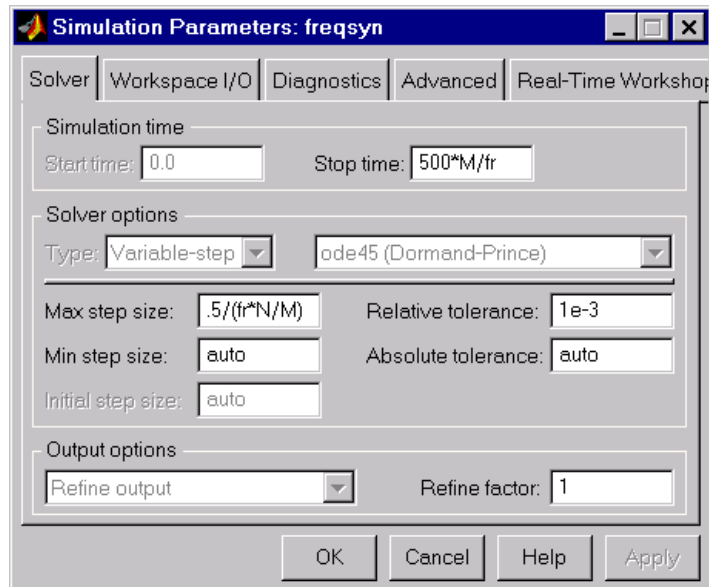
The **Oscillation frequency** parameter is just the quiescent frequency, f_q . The difference between the block's output signal frequency and the oscillation frequency is proportional to the input signal, interpreted as voltage. The oscillation frequency is set to the variable f_q , which is initially assigned a value of 30 MHz. You can change this value in the **Oscillation frequency** dialog box, or by changing the value of f_q at the MATLAB prompt.

The **Input sensitivity** parameter scales the input voltage, and thus controls the shift from the oscillation frequency. The units of the parameter are Hertz per volt. The input sensitivity is set to the variable sen , which is initially assigned a value of 40 MHz/V.

Changing the values of f_q and sen will not affect the steady-state frequency of the synthesized signal, because the corresponding change to the gain value exactly compensates for the change.

Simulation Parameters

You can control the simulation parameters by selecting **Simulation Parameters** from the **Simulation** menu in the model window. This brings up the dialog box shown below.



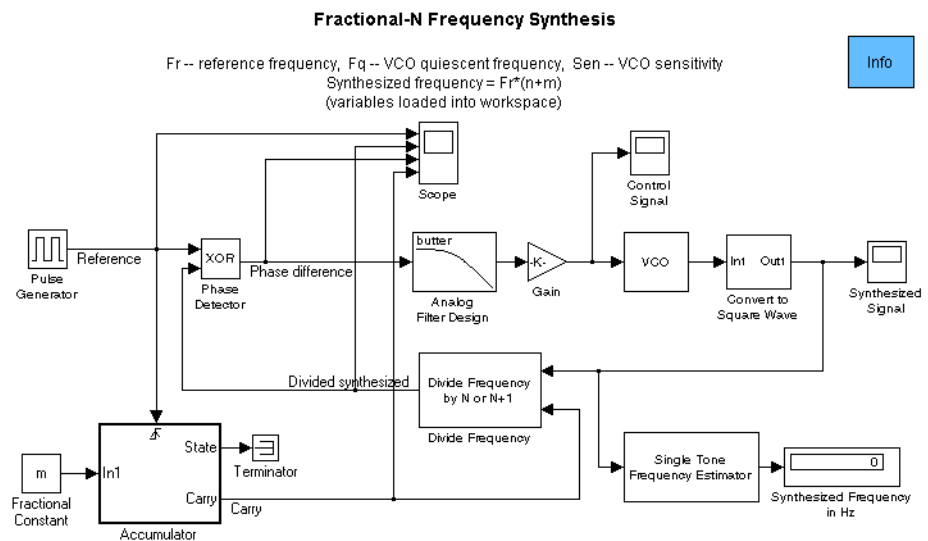
The **Max step size** parameter determines the maximum step size that Simulink's variable-step solver uses to do calculations. This is set to half the period of the synthesized signal. In general, the **Max step size** should be less than the smallest period of all signals occurring in the model.

Fractional-N Frequency Synthesis Demo

This example shows how to simulate a phase-locked fractional-N frequency synthesizer. The model multiplies the frequency, F_r , of a reference signal by a constant $n+m$, to produce a synthesized signal of frequency $F_r \cdot (n+m)$. A feedback loop maintains the frequency of the synthesized signal at this level.

This example is similar to the “Fractional-N Frequency Synthesis Demo” on page 3-46, which produces a synthesized signal of frequency $f_r \cdot N/M$, where N and M are integers. In this example, n is an integer and m is a fraction between 0 and 1. There are several advantages to this approach, since it enables you to approximate the frequency of the synthesized signal with relatively small values for n and m . It also enables you to use a larger reference frequency. See the “Reference” on page 3-49 for more information.

To open the model, type `fracsyn_sim` at the MATLAB prompt (or click here if you are reading this in the MATLAB **Help** browser). In addition to the model window, shown below, this opens two Scope windows, labelled “Control Signal” and “Synthesized Signal.”



Variables in the Model

When you load the model, it creates three variables besides n and m :

- F_r = frequency of the reference signal
- F_q = quiescent frequency in the Voltage-Controlled Oscillator (VCO) block
- S_{en} = Voltage-Controlled Oscillator input sensitivity

Note If you change the values of these variables at the MATLAB command line, use the same upper and lower case letters in the variable names as given above. The variables in the Phase-Locked Frequency Synthesis demo have similar names, but with different cases.

The model initially assigns values to these variables as follows: $n = 10$, $m = .3$, $F_r = 10$ MHz, $F_q = 90$ MHz and $S_{en} = 10$ MHz/V. The frequency of the synthesized signal at the model's steady state is then 103 MHz. After running the simulation with these values, you can later change them by typing new values at the MATLAB prompt, if you want to experiment with the model.

Blocks and Subsystems in the Model

Most of the blocks in this model function in the same way as they do in the “PLL-Based Frequency Synthesis Demo” on page 3-38. You can refer to the documentation for that model for more information about these blocks function. There are two subsystems in this example that are not present in the Phase-Locked Frequency Synthesis demo. They are labeled “Accumulator” and “Divide Frequency.”

Accumulator

The Accumulator subsystem repeatedly adds the constant m to a cumulative sum. While the sum is less than 1, the output labelled “Carry” is 0. At a time step when the sum becomes greater than or equal to 1, the carry output is 1 and the cumulative sum is reset to its fractional part. The fraction of the time when the carry output is 1 is equal to m , while the fraction of the time when it is 0 is equal to $1-m$.

Divide Frequency

The Divide Frequency subsystem divides the frequency of the synthesized signal by n when the output of the Accumulator subsystem is 0, and divides it by $n+1$ when the output is 1. As a result, the average amount that frequency is divided by is

$$(1-m)n + m(n+1) = n + m = 10.3$$

The line leading out of the Divide Frequency subsystem is labeled “Divided synthesized.” At steady state, when the frequency of the synthesized signal is 103 MHz, the divided synthesized signal has an average frequency of 10 Mhz.

Phase Detector

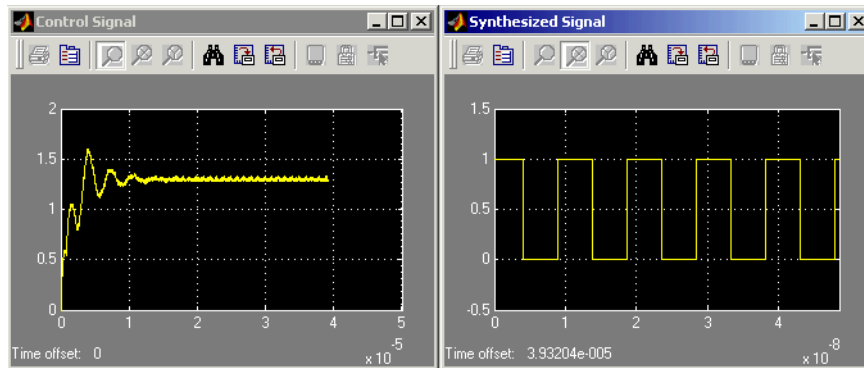
The Logical Operator block, from the Simulink Math library, functions as a phase detector. It applies the XOR operation to the frequencies of the reference signal and the frequency of the output from the Divide Frequency subsystem. The block’s output, labeled “Phase difference,” is 0 where the two input signals are equal, and 1 where they are not equal.

At steady state, the block’s output is a pulse train with frequency of 20 MHz. The reason for this is that both inputs to the block have an average frequency of 10 MHz, but they are out of phase by 1/4 of their period. As a result, the XOR’ed signal is a periodic pulse train with an average frequency of 20 MHz.

You can view the signals these blocks generate by double-clicking on the block labeled “Scope” at the top of the model window.

Running a Simulation

When you run a simulation, two scope windows appear, as shown below.



The left-hand scope displays the control signal, which the Voltage-Controlled Oscillator block uses to maintain the frequency of the synthesized signal. The right-hand scope displays the synthesized signal.

Reference

For further information on phase-locked frequency synthesis, see William F. Egan, *Frequency Synthesis by Phase Lock*, Second Ed. John Wiley & Sons, N.Y.

256-Channel ADSL Demo

The 256-Channel ADSL demo, `adsl_sim`, models part of the asymmetric digital subscriber line (ADSL) technology for transmitting data and multimedia information over telephone lines. It illustrates a downstream path from the central office to the end user. It incorporates the discrete multitone (DMT) signaling modulation technique, which is the focus of the `dmt_sim` demo. This document highlights these aspects of the `adsl_sim` demo:

- Structure of the demo
- Transmitting data
- Processing received data
- Displaying error statistics

Alternatively, an animated tour of an ADSL model is at http://www.mathworks.com/products/dsp_comm/demos.shtml. Use this link if you are reading this in the MATLAB Help browser.

Structure of the Demo

The model generates random binary data frames, transmits them according to the ADSL specification, simulates a telephone line using an FIR filter of length 101 and the AWGN Channel block, tries to recover the information from the received data, and computes error statistics. The model uses frame-based processing, thereby processing many bits in each time step. The double connector lines between blocks indicate frame-based signals.

Because these processes involve many blocks, the demo uses subsystems to organize some groups of blocks, and it uses Goto/From block pairs and colored regions to make the block diagram visually neater.

Transmitting Data

The transmitter portion of the model, shaded in blue at the top of the model, contains two parallel paths. One path (the fast buffer) processes the first 776 bits of each 1552-bit data frame, while the other path (the interleaved buffer) processes the last 776 bits of each data frame. Each path appends eight cyclic redundancy check (CRC) bits to its 776-bit frame, scrambles the bits, and encodes them using a shortened Reed-Solomon code. The scrambling and encoding operations interpret the bits as integers between 0 and 127. In the

second path but not the first, a Convolutional Interleaver block interleaves the encoded data. This interleaving operation increases the second path's resistance to burst errors but also its latency. Finally, the data from the two routes are concatenated and modulated. Data from the fast buffer is modulated to the low frequency subcarriers, while data from the interleaved buffer is modulated to the high frequency subcarriers, according to the bit allocation vector `b`. This demo assumes that the bit allocation vector is known and uses the vector to calculate the channel. Type `get_param('adsl_sim', 'preLoadFcn')` to see the calculations involved. For more information about the DMT Modulator block in this demo, see “Discrete Multitone Signaling Demo”.

Processing Received Data

The receiver attempts to undo each operation that the receiver performed. Much of the receiver's design is straightforward; for example, to undo the actions of the Convolutional Interleaver block, use a Convolutional Deinterleaver block with the same mask parameters. The frequency domain equalizer in the DMT Demodulator subsystem mitigates the channel distortion.

Aligning Frames to Account for Delays

One subtle point in the receiver portion is the Integer Delay block that follows the Convolutional Deinterleaver block. This Integer Delay block delays the deinterleaved data by 800 samples. Because the delay between the original and restored sequences is 40 samples (5 shift registers times a maximum delay of $2 \times (5-1)$ samples among all shift registers), the extra 800-sample delay ensures that bits are properly aligned in the 840-bit frame.

Displaying Error Statistics

Two display icons show error statistics for comparisons between the transmitted and received data in the two paths (with and without interleaving). Two other display icons show error statistics based on the CRC bits, where any nonzero bit among the eight CRC bits indicates a frame error.

In each of the display icons, the error statistics consist of the bit error rate, the number of bit errors, and the total number of bits processed.

Selected Bibliography

[1] Bingham, John A. C. *ADSL, VDSL, and Multicarrier Modulation*. Wiley: New York, 2000.

[2] *ITU-T Recommendation G.992.1 Asymmetric Digital Subscriber Line (ADSL) Transceivers*. Geneva: Telecommunication Standardization Sector of International Telecommunication Union, 1999.

[3] Maxwell, Kim. "Asymmetric Digital Subscriber Line: Interim Technology for the Next Forty Years." *IEEE Communications Magazine*, October 1996. 100-106.

Bluetooth Voice Transmission Demo

The Bluetooth Voice Transmission demo, `bluetooth_voice`, models part of a Bluetooth system. Bluetooth is a short-range radio link technology that operates in the 2.4 GHz Industrial, Scientific, and Medical (ISM) band. The demo modulates the signal using Gaussian frequency shift keying (GFSK) over a radio channel with maximum capacity of 1Mbps.

The demo uses frequency hopping over a 79 MHz frequency range to avoid interference with other devices transmitting in the band. In this scheme, the sender divides transmission time into 625 microsecond slots, and uses a new hop frequency for each slot. Although the data rate is only 1Mbps, a much larger bandwidth of 79MHz is required to simulate the frequency hopping effects.

This document highlights the following aspects of the Bluetooth Voice Transmission demo:

- “Structure of the Demo”
- “Mask Variables”
- “Results and Display”

Structure of the Demo

The demo contains the following elements:

- Master transmitter,
- Radio channel
- IEEE 802.11b interferer
- Slave receiver,
- Bit error rate (BER) display
- Instrumentation.

The transmitter subsystem performs speech coding, buffering, framing, header error control (HEC), forward error correction (FEC), GFSK modulation, and frequency hopping. Channel effects modeled include thermal noise, path loss and interference. The Free Space Path Loss block, from the RF Impairments library, models path loss. The IEEE 802.11b interferer is a masked subsystem that opens up a mask dialog for user input on double-clicks. Mean packet rate, packet length, power and frequency location in the ISM band can be specified

in the mask dialog. The Slave Receiver recovers speech from the transmitted signal, performing all the complementary operations that the transmitter does, but in reverse order..

The demo makes extensive use of frame-based processing, which can propagate large frames of samples at each execution step, allowing for much faster simulation of digital systems. The double connector lines between the blocks indicate frame-based signals.

The demo also uses subsystems to organize groups of blocks, and it uses Goto/From block pairs and colored regions to make the block diagram visually neater.

Mask Variables

You can open the **Model Parameters** mask dialog by double-clicking the block labeled “Double-click to select Model Parameters.” In the mask dialog, you can specify

- The high-quality voice (HV) packet type in the **HV Packet type** field
- The initial slot pair type in the **Initial Slot Pair for HV3** field

When you load the model, the `bluetooth_init.m` executes, creating several variables in the MATLAB workspace by using the `PreLoadFcn` model callback parameter.

Results and Display

To examine the performance of the demo, double-click the switches to bring up error rate display and instrumentation.

The error rate display shows three types of error rates:

- Raw bit error rate
- Residual bit error rate
- Frame error rate (FER)

The raw bit error rate displays the inconsistencies between the bits in the transmitted signal and the received signal. Frame error rate refers to the ratio of frame failure to the total number of frames. Frame failure, caused by noise and interference, is determined if the HEC fails to match the header info or if less than 57 bits are correct in the access code. If the frame fails, this is

captured by a zero-valued Frame OK signal, which is used in the FER calculation as well as to exclude bad frames from the residual BER calculation.

The Instrumentation brings up the spectrum of the transmitted Bluetooth signal (narrow-band) with IEEE 802.11b interference. The timing diagram for the Bluetooth and interferer slots is also available. A dynamic plot of packet frequency versus time is shown by the Spectrogram plot. The thin lines are the Bluetooth transmissions, while the larger, more colorful blocks are the interferer slots. Most of the time, due to frequency hopping, there is not much overlap of these slots. In a few cases, the signals do collide, as the Spectrogram plot clearly shows.

Reference

[1] <http://www.bluetooth.com>

Digital Video Broadcasting Demo

The Digital Video Broadcasting demo, `dvbt_sim`, models part of the ETSI (European Telecommunications Standards Institute) EN 300 744 standard for terrestrial transmission of digital television signals. The standard prescribes the transmitter design and sets minimum performance requirements for the receiver. The purpose of this demo is to:

- Model the transmitter in its “2k mode,” as prescribed in the standard
- Model one possible receiver design
- Generate error statistics that will help determine whether the receiver model satisfies the performance requirements

This document highlights these aspects of the demo:

- The overall structure of the demo, which mimics the block diagram schematic in the standard
- Variables in the demo
- Design of the receiver portion of the demo
- Visible results of the demo

Structure of the Demo

Using a list and a schematic, the standard shows the major processes that the data undergoes. The top row of blocks in the demo mimics the structure of the schematic, by including subsystems that perform such major processes. The table below shows which subsystems correspond to processes from the schematic.

Process in Schematic	Subsystem or Block in Demo
Outer coder	(204, 188) Shortened Reed-Solomon Encoder
Outer interleaver	Convolutional Interleaver I=12
Inner coder	Rate 3/4 Punctured Convolutional Code
Inner interleaver	DVB-T Inner Interleaver

Process in Schematic	Subsystem or Block in Demo
Mapper	DVB-T 64-QAM Mapper
OFDM	OFDM Transmitter

The bottom row of icons in the demo represent subsystems that make up the receiver. The demo also includes a source of random data, a channel model, error statistic calculators, and several sinks.

Variables in the Demo

When you load the model, it creates several variables in the MATLAB workspace, using the `PreLoadFcn` model callback parameter.

Variable	Purpose in Demo
<code>Ts</code>	Sample time of random integer source
<code>dvb_bit_int_table</code>	Table for Bit Interleaver and Bit Deinterleaver
<code>dvb_sym_int_table</code>	Table for Symbol Interleaver and Symbol Deinterleaver
<code>dvbt_qam</code>	Signal constellation for 64-QAM mapping

To see how MATLAB computes the values of these variables, see the script `dvbt_table_gen.m`.

Design of the Receiver

The standard does not specify how to implement the receiver, although some inverse operations, such as deinterleaving, are clearly defined. This demo illustrates one possible receiver design by using these features:

- A 64-QAM demapper that makes soft decisions, producing a set of six real numbers for each complex number in its input. These six numbers represent soft decisions on the real and imaginary components' first bit, second bit, and

third bit. The Viterbi Decoder subsystem interprets the soft-decision numbers and uses them to decode the punctured convolutional code properly.

To examine the exact mapping more closely, see the DVB-T 64-QAM Demapper subsystem, as well as the `dvbt_qam` variable in the MATLAB workspace.

- A traceback depth of 136 in the Viterbi Decoder library block. This library block appears within the top-level Viterbi Decoder subsystem.

Visible Results of the Demo

To examine the performance of the demo, use the sink blocks that are included in it, listed in the table below.

Icon or Window	What it Shows
Leftmost Display icon	Error statistics for the entire system
Rightmost Display icon	Error statistics for the inner coder
Spectrum Scope window	Spectrum of the received OFDM signal
Delayed Scatter Plot window	Scatter plot of the received 64-QAM signal

Selected Bibliography

[1] *ETSI Standard EN 300 744: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television.* Valbonne, France: European Telecommunications Standards Institute, 1997.

HiperLAN/2 Demo

The HIPERLAN/2 demo, `hiperlan2`, models part of HIPERLAN/2 (high performance radio local area network), European (ETSI) Standard for high-rate wireless LANs. It employs Orthogonal Frequency Division Multiplexing (OFDM) that operates in the 5GHz band and offers raw data rates up to 54 Mbps. The model shows transmitter side coding and modulation for the 16 QAM, $\frac{3}{4}$ code rate mode with a corresponding ideal receiver chain and AWGN channel. This document highlights these aspects of the demo:

- The overall structure of the demo
- Visible results and display of the demo

Structure of the demo

The demonstration contains components that model the essential features of the HiperLAN/2 standard. The top row of blocks contains the transmitter components while the bottom row contains the receiver components. The table below shows which blocks and subsystems correspond to processes from the standard.

Process in Standard	Block or Subsystem in the Demo
FEC Coding	Convolutional Encoder and P2 Puncture
Data interleaving	Matrix Interleaver and General Block Interleaver
Signal constellations and mapping	Normalize
Modulation technique (OFDM)	OFDM Transmitter and OFDM Receiver

The demo also includes the Bernoulli Binary Generator block as a data source, the AWGN Channel block to simulate noise, and the Error Rate Calculation block and Display block to show error statistics.

Visible Results and Display

To examine the performance of the demo, use the sink blocks that are included in it, listed in the table below:

Icon or Window	What it Shows
Display icon	Error statistics for the entire system
Spectrum Scope window	Flat-top spectrum of the OFDM signal
Scatter Plot after normalization	Scatter plot of transmitted 16QAM signal
Scatter Plot after OFDM Receiver	Scatter plot of received 16QAM signal

The Spectrum Scope and the Scatter Plot after OFDM Receiver are visible upon running the demo. The Scatter Plot after normalization can be viewed by double-clicking on the 'Transmitted Signal' Scatter Plot block

References

[1] *ETSI TS 101 475 V1.2.2 (2001-02) Broadband Radio Access Networks (BRAN): HIPERLAN Type 2: Physical (PHY) Layer*. Available on <http://www.etsi.org>

RF Satellite Link Demo

This RF Satellite Link demo, `rf_satlink`, presents the simulation of a satellite link with blocks from the Communications Blockset's RF Impairments Library (red blocks). These blocks simulate the following impairments:

- Free space path loss
- Receiver thermal noise
- Memoryless nonlinearity
- Phase noise
- In phase and quadrature imbalances
- Phase/frequency offsets

If you are familiar with Simulink and RF impairments, double-click the block labeled “RF Link Demo: Settings” in the lower left corner of the demo and follow the suggested scenarios.

By modeling the gains and losses on the link, this model is an implementation of the link budget calculations that determine if a downlink can be closed with a given bit error rate (BER). The gain and loss blocks, including the Free Space Path Loss block and the Receiver Thermal Noise block, determine the data rate that can be supported on the link in an additive white Gaussian noise channel. The demonstration contains additional RF impairment blocks such as the Memoryless Nonlinearity, I/Q Imbalance, Phase Noise, Phase/Frequency Offset, and I/Q Imbalance blocks so you can view the effect of the corresponding impairments on the link. The description of this demo is divided into the following sections:

- “Structure of the demo” on page 3-61
- “Mask Parameters” on page 3-63
- “Results and Displays” on page 3-66
- “Experimenting with the Demo” on page 3-66
- “Selected Bibliography” on page 3-69

Structure of the demo

The demo highlights both the satellite link model and signal instrumentation. The model consists of a Satellite Downlink Transmitter, Downlink Path, and

Ground Station Downlink Receiver. The blocks that comprise each of these sections are:

- Satellite Downlink Transmitter
 - Random Integer Generator – Creates a random data stream.
 - Rectangular QAM Modulator Baseband – Maps the data stream to 16-QAM constellation.
 - FIR Interpolator (RRC Tx. Filter) – Upsamples and shapes the modulated signal using the square root raised cosine pulse shape.
 - Memoryless Nonlinearity (High Power Amplifier) – Model of a traveling wave tube amplifier (TWTA) using the Saleh model.
 - Gain (Tx. Dish Antenna Gain) – Gain of the transmitter parabolic dish antenna on the satellite.
- Downlink Path
 - Free Space Path Loss (Downlink Path) – Attenuates the signal by the free space path loss.
 - Phase/Frequency Offset (Doppler and Phase Error) – Rotates the signal to model phase and Doppler error on the link.
- Ground Station Downlink Receiver
 - Receiver Thermal Noise (Satellite Receiver System Temp) – Adds white Gaussian noise that represents the effective system temperature of the receiver.
 - Gain (Rx. Dish Antenna Gain) – Gain of the receiver parabolic dish antenna at the ground station.
 - Phase Noise – Introduces random phase perturbations that result from $1/f$ or phase flicker noise.
 - I/Q Imbalance – Introduces DC offset, amplitude imbalance, or phase imbalance to the signal.
 - DC Removal (DC Offset Comp.) – Estimates and removes the DC offset from the signal. Compensates for the DC offset in the I/Q Imbalance block.
 - Magnitude AGC / I and Q AGC (Select AGC) – Automatic Gain Control Compensates the gain of both inphase and quadrature components of the signal either jointly or independently.
 - Phase/Frequency Offset (Doppler and Phase Compensation) – Rotates the signal to represent correction of phase and Doppler error on the link. This

block is a static block that simply corrects using the same values as the Phase/Frequency Offset block.

- FIR Interpolator (RRC Rx. Filter) and Downsample – Matched filter the modulated signal using the square root raised cosine pulse shape.
- Rectangular QAM Demodulator Baseband - Demaps the data stream from the 16-QAM constellation space.

Mask Parameters

Double-click the block labeled “RF Link Demo: Settings” in the lower left corner of the model window to view the parameter settings for the demo. All of the mask parameters are tunable. The demo is updated when you click **OK** or **Apply**. The parameters are:

- **Satellite altitude (km)** – The distance between the satellite and the ground station. Changing this parameter updates the Free Space Path Loss block. The default setting is 35600.
- **Frequency (MHz)** – The carrier frequency of the link. Changing this parameter updates the Free Space Path Loss block. The default setting is 8000.
- **Transmit and receive antenna sizes (m)** – The first element in the vector represents the transmit antenna diameter and is used to calculate the gain in the Tx Dish Antenna Gain block. The second element represents the receive antenna diameter and is used to calculate the gain in the Rx Dish Antenna Gain block. The default setting is [2 2].
- **Noise temperature (K)** – Allows you to select from three effective receiver system noise temperatures. The select noise temperature changes the Noise Temperature of the Receiver Thermal Noise block. The default setting is 0 K. The choices are:
 - **0 (no noise)** – Use this setting to view the other RF impairments without the perturbing effects of noise.
 - **20 (very low noise level)** – Use this setting to view how easily a low level of noise can, when combined with other RF impairments, degrade the performance of the link.
 - **290 (typical noise level)** – Use this setting to view how a typical quiet satellite receiver operates.

- **HPA backoff level** – Allows you to select from three backoff levels. This parameter is used to determine how close the satellite high power amplifier is driven to saturation. The selected backoff is used to set the input and output gain of the Memoryless Nonlinearity block. The default setting is 30 dB (negligible nonlinearity). The choices are:
 - **30 dB (negligible nonlinearity)** – Sets the average input power to 30 decibels below the input power that causes amplifier saturation (i.e. the point at which the gain curve becomes flat) This causes negligible AM-to-AM and AM-to-PM conversion. AM-to-AM conversion is an indication of the amplitude nonlinearity varies with the signal magnitude. AM-to-PM conversion is a measure of how the phase nonlinearity varies with signal magnitude
 - **7 dB (moderate nonlinearity)** – Sets the average input power to 7 decibels below the input power that causes amplifier saturation. This causes moderate AM-to-AM and AM-to-PM conversion.
 - **1 dB (severe nonlinearity)** – Sets the average input power to 1 decibel below the input power that causes amplifier saturation. . This causes severe AM-to-AM and AM-to-PM conversion.
- **Phase correction** - Allows you to select from three phase offset values to correct for the average AM-to-PM conversion in the High Power Amplifier. The selection updates the Phase/Frequency Offset (Doppler and Phase Compensation) block. The default setting is None. The choices are:
 - **None** – No correction. Use to view uncorrected AM-to-PM conversion.
 - **Correct for moderate HPA AM-to-PM** – Corrects for average AM-to-PM distortion when the HPA backoff is set to 7 dB
 - **Correct for severe HPA AM-to-PM** – Corrects for average AM-to-PM distortion when the HPA backoff is set to 1 dB
- **Doppler error** – Allows you to select from three values of Doppler on the link and the corresponding correction, if any. The selection updates the Phase/Frequency Offset (Doppler and Phase Error) and Phase/Frequency Offset (Doppler and Phase Compensation) blocks. The default setting is None. The choices are:
 - **None** - No Doppler on the link and no correction.
 - **Doppler (0.7 Hz - uncorrected)** – Adds 0.7 Hz Doppler with no correction at the receiver.

- **Doppler (3 Hz - corrected)** – Adds 3 Hz Doppler with the corresponding correction at the receiver, -3 Hz.
- **Phase noise** – Allows you to select from three values of phase noise at the receiver. The selection updates the Phase Noise block. The default setting is Negligible (-100 dBc/Hz @ 100 Hz). The choices are:
 - **Negligible (-100 dBc/Hz @ 100 Hz)** – Almost no phase noise.
 - **Low (-55 dBc/Hz @ 100 Hz)** – Enough phase noise to be visible in both the spectral and I/Q domains, and cause additional errors when combined with thermal noise or other RF impairments.
 - **High (-48 dBc/Hz @ 100 Hz)** – Enough phase noise to cause errors without the addition of thermal noise or other RF impairments.
- **I/Q imbalance** – Allows you to select from five types of inphase and quadrature imbalances at the receiver. The selection updates the I/Q Imbalance block. The default setting is None. The choices are;
 - **None** – No imbalances
 - **Amplitude imbalance (3 dB)** – Applies a 1.5 dB gain to the inphase signal and a -1.5 dB gain to the quadrature signal.
 - **Phase imbalance (20 deg)** – Rotates the inphase signal by 10 degrees and the quadrature signal by -10 degrees.
 - **In-phase DC offset (2e-6)** – Adds a DC offset of 2e-6 to the inphase signal amplitude. This offset will change the received signal scatter plot, but will not cause errors on the link unless combined with thermal noise or other RF impairments.
 - **Quadrature DC offset (1e-5)** – Adds a DC offset of 1e-5 to the quadrature signal amplitude. This offset will cause errors on the link even when not combined with thermal noise or another RF impairment. This offset also causes a DC spike in the received signal spectrum.
- **DC offset compensation** – Allows you to enable or disable the DC Offset block. The selection updates the DC Removal block. The default setting is **Disabled**.
- **AGC type** – Allows you select the automatic gain control for the link. The selection updates the Select AGC block, which is labeled Magnitude AGC or I and Q AGC, depending on whether you select **Magnitude only** or **Independent I and Q**, respectively. The default setting is **Magnitude only**.

- **Magnitude only** - Compensates the gain of both inphase and quadrature components of the signal by estimating only the magnitude of the signal.
- **Independent I and Q** - Compensates the gain of the inphase signal using an estimate of the inphase signal magnitude and the quadrature component using an estimate of the quadrature signal magnitude.

Results and Displays

When you run this demo, the following displays are available to you:

- **Bit error rate (BER) display** – In the lower right corner of the model is a display of the BER of the model. The BER computation is reset every 5000 symbols to allow you to view the impact of the changes in the model without having to restart the model.
- **Spectrum Scope** – Double-clicking on this block to turn the switch to the ON position allows you to view the spectrum of the modulated/filtered signal (blue) and the received signal before demodulation (red). If both spectra are identical, then the display will show one green spectrum. Comparing these spectra allows you to view the effect of the following RF impairments:
 - Spectral regrowth due to HPA nonlinearities caused by the Memoryless Nonlinearity block,
 - Thermal noise caused by the Receiver Thermal Noise block, and
 - Phase flicker (i.e. $1/f$ noise) caused by the Phase Noise block.
- **End to End Constellation** – Double-clicking this block to turn the switch to the ON position allows you to view the scatter plots of the signal after QAM modulation (blue) and before QAM demodulation (red). Comparing these scatter plots allows you to view the impact of all of the RF impairments on the received signal and the effectiveness of the compensations.
- **Constellation Before and After HPA**– Double-clicking this block to turn the switch to the ON position allows you to view the constellation before and after the HPA (blue and red respectively). Comparing these plots allows you to view the effect that the nonlinear HPA behavior has on the signal.

Experimenting with the Demo

This section describes some ways that you can change the demo parameters, in order to experiment with the effects of the blocks from the RF Impairments

library and other blocks in the demo. The following lists some suggested scenarios along with the changes from the default settings:

- Link gains and losses – Change **Noise temperature** to **290 (typical noise level)** or to **20 (very low noise level)**. Change the value of the **Satellite altitude (km)** or **Satellite frequency (MHz)** parameters to change the free space path loss. In addition, increase or decrease the **Transmit and receive antenna size (m)** parameter to increase or decrease the received signal power. You can view the changes in the received constellation in the received signal scatter plot scope and the changes in received power in the spectrum scope. In cases where the change in signal power is large (greater than 10 dB), the AGC filter causes the received power (after the AGC) to oscillate before settling to the final value.
- Raised cosine pulse shaping – Make sure **Noise temperature** is set to **0 (no noise)**. Turn on the Constellation Before and After HPA instrumentation. Observe that the square-root raised cosine filtering results in intersymbol interference (ISI). This results in the points being scattered loosely around ideal constellation points, which you can see in the After HPA scatter plot. The square-root raised cosine filter in the receiver, in conjunction with the transmit filter, control the ISI, which you can see in the received signal scatter plot.
- HPA AM-to-AM conversion and AM-to-PM conversion – Change the **HPA backoff level** parameter to **7 dB (moderate nonlinearity)** and observe the AM-to-AM and AM-to-PM conversions by comparing the Transmit RRC filtered signal scatter plot with the RRC signal after HPA scatter plot. Note how the AM-to-AM conversion varies according to the different signal amplitudes. You can also view the effect of this conversion on the received signal in the received signal scatter plot. In addition, you can observe the spectral regrowth in the received signal spectrum scope. You can view the AM-to-PM conversion compensation in the receiver by setting the **Phase correction** parameter to **Correct for moderate HPA AM-to-PM**. You can also view the phase change in the received signal in the received signal scatter plot scope.

Change the **HPA backoff level** parameter to **1 dB (severe nonlinearity)** and observe from the scopes that the AM-to-AM and AM-to-PM conversion and spectral regrowth have increased. You can view the AM-to-PM conversion to compensate in the receiver by setting the **Phase correction** to

Correct for severe HPA AM-to-PM. You can view the phase change in the received signal scatter plot scope.

- Phase noise plus AM-to-AM conversion – Set the **Phase Noise** parameter to **High** and observe the increased variance in the tangential direction in the received signal scatter plot. Also note that this level of phase noise is sufficient to cause errors in an otherwise error-free channel. Set the **Phase Noise** to **Low** and observe that the variance in the tangential direction has decreased somewhat. Also note that this level of phase noise is not sufficient to cause errors. Now, set the **HPA backoff level** parameter to **7dB (moderate nonlinearity)** and the **Phase correction** to **Correct for moderate HPA AM-to-PM conversion**. Note that even though the corrected, moderate HPA nonlinearity and the moderate phase noise do not cause bit errors when applied individually, they do cause bit errors when applied together.
- DC offset and DC offset compensation – Set the **I/Q Imbalance** parameter to **In-phase DC offset (2e-6)** and view the shift of the constellation in the received signal scatter plot. Set **DC offset compensation** to **Enabled** and view the received signal scatter plot to view how the DC offset block estimates the DC offset value and removes it from the signal. Set **DC offset compensation** to **Disabled** and change **I/Q imbalance** to **Quadrature DC offset (1e-5)**. View the changes in received signal scatter plot for a large DC offset and the DC spike in the received signal spectrum. Set **DC offset compensation** to **Enabled** and view the received signal scatter plot and spectrum scope to see how the DC component is removed.
- Amplitude imbalance and AGC type – Set the **I/Q Imbalance** parameter to **Amplitude imbalance (3 dB)** to view the effect of unbalanced I and Q gains in the received signal scatter plot. Set the **AGC type** parameter to **Independent I and Q** to demonstrate how the independent I and Q AGC compensate for the amplitude imbalance.
- Doppler and Doppler compensation – Set **Doppler error** to **0.7 Hz (uncorrected)** to demonstrate the effect of uncorrected Doppler on the received signal scatter plot. Set the **Doppler error** to **3 Hz corrected** to demonstrate the effect of correcting the Doppler on a link. Without changing the Doppler error setting, repeat the following scenarios:
 - DC offset and DC offset compensation

- Amplitude imbalance and AGC type
to view the effects that occur when DC offset and amplitude imbalances occur in circuits that do not have a constant phase reference.

Selected Bibliography

- [1] "Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers", Adel A. M. Saleh, IEEE Transactions on Communications, Vol. COM-29, No. 11, November 1981.
- [2] Kasdin, N.J., "Discrete Simulation of Colored Noise and Stochastic Processes and $1/(f^\alpha)$; Power Law Noise Generation," The Proceedings of the IEEE, May, 1995, Vol. 83, No. 5
- [3] "Discrete Simulation of Power Law Noise", N. Jeremy Kasdin and Todd Walter, 1992 IEEE Frequency Control Symposium.
- [4] Digital Communications, Fundamentals and Applications, Bernard Sklar, Prentice Hall, Englewood Cliffs, NJ, copyright 1988.

WCDMA Coding and Multiplexing Demo

The WCDMA Coding and Multiplexing demo, `wcdma_muxandcoding`, presents a simulation of the multiplexing and channel decoding structure for the frequency division duplex (FDD) downlink as specified by the Third Generation Partnership Project (3GPP), Release 1999. The demo comprises half of the model described in “WCDMA End-to-End Physical Layer Demo” on page 3-71.

WCDMA End-to-End Physical Layer Demo

The WCDMA End-to-End Physical Layer Demo, `wcdma_player`, models part of the frequency division duplex (FDD) downlink physical layer of the third generation wireless communication system known as wideband code division multiple access (WCDMA).

WCDMA is one of five air-interfaces for the next generation of wireless communications being developed within the framework of the International Mobile Telecommunications (IMT)-2000, as defined by the International Telecommunication Union (ITU). The WCDMA technology is officially known as IMT-2000 Direct Spread.

The specifications of the WCDMA system are being developed by the Third Generation Partnership Project (3GPP), Release 1999, which is a joint effort between standards bodies from Europe, Japan, Korea, USA, and China.

The WCDMA air interface is a direct spread technology. This means that it spreads encoded user data at a relatively low rate over a much wider bandwidth (5 MHz), using a sequence of pseudo-random units called chips at much higher rate (3.84 Mcps). By assigning a unique code to each user, the receiver, which has knowledge of the code of the intended user, can successfully separate the desired signal from the received waveform.

This document highlights the following aspects of the demo:

- “Overall Structure of the Physical Layer” on page 3-71
- “Parameters in the Demo” on page 3-74
- “Visible Results of the Demo” on page 3-77
- “References” on page 3-78

Overall Structure of the Physical Layer

The physical layer is in charge of providing transport support to the data generated at higher layers. This data is exchanged between the higher layers and the physical layer in the form of transport channels. There can be up to eight transport channels processed simultaneously. Each transport channel has associated a different transport format that contains information of how the data needs to be processed by the physical layer. The physical layer processes this data before sending it to channel.

There are seven main subsystems in the model, whose functions are summarized in the following table.

Subsystem	Function
WCDMA DL Tx Channel Coding Scheme	Transport channel encoding and multiplexing
WCDMA Tx Physical Channel Mapping	Physical channel mapping
WCDMA BS Tx Antenna	Modulation and spreading
WCDMA Channel Model	Channel model
WCDMA UE Rx Antenna	Despreading and demodulation
WCDMA Rx Physical Channel Demapping	Physical channel demapping
WCDMA DL Rx Channel Decoding Scheme	Transport channel demultiplexing and decoding

WCDMA DL Tx Channel Coding Scheme

The WCDMA DL Tx Channel Coding Scheme subsystem processes each transport channel independently according to the transport format parameters associated to it. This subsystem implements the following functions:

- Cyclic redundancy code (CRC) attachment
- Transport block concatenation and segmentation
- Channel encoding
- Rate matching
- First interleaving
- Radio frame segmentation

The different transport channels are then combined together to generate a coded combined transport channel (CCTrCH). The CCTrCH is then sent to the WCDMA Tx Physical Mapping subsystem.

WCDMA Tx Physical Mapping

This subsystem implements the following functions:

- Physical channel segmentation
- Second interleaver
- Slot builder

The output of this subsystem constitutes a dedicated physical channel (DPCH), which is passed to the WCDMA BS Tx Antenna Spreading and Modulation subsystem.

WCDMA BS Tx Antenna Spreading and Modulation

The WCDMA BS Tx Antenna Spreading and Modulation subsystem performs the following functions:

- Modulation
- Spreading by a real-valued orthogonal variable spreading factor (OVSF) code
- Scrambling by a complex-valued Gold code sequence
- Power weighting
- Pulse shaping

WCDMA Channel Model

The WCDMA Channel Model subsystem simulates a wireless link channel containing additive white Gaussian noise (AWGN) and, if selected, a set of multipath propagation conditions. You can modify the multipath profile with the **Propagation conditions environment** parameter, as described in “Propagation conditions environment” on page 3-77

WCDMA UE Rx Antenna

The received signal at the WCDMA UE Rx Antenna subsystem is the sum of attenuated and delayed versions of the transmitted signals due to the so-called multipath propagation introduced by the channel. At the receiver side, a RAKE receiver is implemented to resolve and compensate for such effect. A Rake receiver consists of several rake fingers each of them associated to a different received component. Each rake finger is made of chip correlators to perform the despreading, channel estimation to gauge the channel and a derotator that using the knowledge provided by the channel estimator corrects the phase of

the data symbol. The subsystem coherently combines the output of the different rake fingers to recover the energy across the different delays.

WCDMA RX Physical Channel Demapping and Channel Decoding Scheme

The WCDMA RX Physical Channel Demapping and the WCDMA DL Rx Channel Decoding Scheme subsystem decode the signal by performing the inverse of the functions of the WCDMA DL Tx Channel Coding Scheme subsystem, as described above.

Parameters in the Demo

You can view or change parameters in the model by double-clicking the block labeled "WCDMA Demo: Initial Settings." This displays the **Block Parameters** dialog.

The **Power for [DPCH, P-CPICH, PICH, PCCPCH, SCH] in dB** parameter consists of a row vector containing the powers in decibels corresponding to the the different physical channels.

The **Show Transport Channel Settings** check box enables you to specify the parameters corresponding to the WCDMA Tx Channel Coding Scheme subsystem, the WCDMA Tx PhCh Mapping subsystem, and its corresponding subsystems at the receiver side. When the box is selected, the dialog displays the following parameters:

Parameter	Description
DL Measurement channels	<p>The down link (DL) measurement channels. There are four channels whose settings are specified by the standard:</p> <ul style="list-style-type: none"> • 12.2 Kbps • 64 Kbps • 144 Kbps • 384 Kbps <p>If you select one of these channels, the parameters listed below are greyed out. To change these parameter settings, select User Defined.</p>
Transport block set size	Integer row vector representing the transport block set size as defined by the standard associated to each transport channel.
Transport block size	Integer row vector representing the transport block size as defined by the standard associated to each transport channel.
TTI in ms	Integer row vector representing the transmission time interval (TTI) in ms as defined by the standard associated to each transport channel.
CRC Size	Integer row vector representing the CRC size in number of bits associated to each transport channel.
Type of error Protection	<p>Integer row vector representing the coding scheme associated to each transport channel. The different options are:</p> <ul style="list-style-type: none"> • 1 for no coding • 2 for convolutional encoding • 3 for turbo coding.

Parameter	Description
Rate matching attribute	Integer row vector representing the rate matching attribute as defined by the standard associated to each transport channel
Position of TrCH in radio frame	Sets the position of the transport channels in the radio frame to be Fixed or Flexible as defined by the standard
Number of PhCH	Integer from 1 to 3 corresponding to the number of physical channels used.
Slot format (0..16)	Sets the corresponding slot format parameter as defined by the standard.

The **Show Antenna Settings** check box enables you to specify the parameters corresponding to the WCDMA BS Tx Antenna and WCDMA UE Rx Antenna subsystems. When the box is checked, the dialog displays the following parameters:

Parameter	Description
DPCH Code number	Integer number from 0 to the value of the spreading factor minus 1 corresponding to the index of the orthogonal code assigned to the DPCH channel.
Scrambling code	Vector of two elements corresponding to the index of the Scrambling Code assigned to the Base Station.
Number of filter taps for RRC	Number of filter coefficients for the root-raised cosine filter.

Parameter	Description
Number of coefficients for channel estimation filters	Number of filter coefficients for low pass filter implemented in channel estimation.
Oversampling factor	Integer value corresponding to the number of samples per symbol

The **Show Channel Model Settings** check box enables you to specify the parameters corresponding to the WCDMA Channel Model subsystem:

Parameter	Description
Propagation conditions environment	Selects between the different pre-built propagation conditions environments.
Number of enable fingers	Integer from 1 to 4 that sets the number of enable fingers
SNR (in dB)	Value of the signal to noise ratio in decibels.
Relative delay of Rx signals (in s)	Vector corresponding to the delay (in s) of the different paths.
Average Power of Rx signals (in dB)	Vector corresponding to the Power (in dB) of the different path.
Speed of Terminal (in Km/h)	Value of the speed of the UE (User Equipment) in Km/h

Visible Results of the Demo

The following blocks calculate various error rates in the demo:

- BLER (Block Error Rate) Calculation shows the block error rate of the combined transport channels.
- BER (Bit Error Rate) Calculation shows the results of the BER computation block associated to each transport channel separately.

The following scopes display the signal in various ways. To view the scopes, double-click on the switches when the simulation is running:

- Time Scopes shows the bit stream before spreading, after spreading and after combining the different weighted physical channels. It shows both the real and the imaginary part separately. It also displays both the real and the imaginary part of the output of the channel estimator for the first rake finger.
- Power spectrum shows the power spectrum of the signal before spreading, after spreading, after pulse shaping and at the input of the receiver antenna.
- Scatter plots show the constellation at signal at the output of the data correlator, after phase derotation and after amplitude correction.

References

[1] <http://www.3gpp.org>

WCDMA Spreading and Modulation Demo

The WCDMA Spreading and Modulation demo, `wcdma_spreadandmod`, simulates spreading and modulation for an FDD downlink DPCH channel as specified by Third Generation Partnership Project (3GPP), Release 1999. The demo comprises half of the model described in “WCDMA End-to-End Physical Layer Demo” on page 3-71.

A

- A-law companders 1-38
- amplitude modulation (AM)
 - example model 1-87
- analog modulation libraries 1-81
- analog-to-digital conversion 1-30

B

- baseband simulation 1-82
 - compared to passband 2-24
 - signals in 1-82
- binary codes 1-42
- binary numbers, order of digits and 1-45
- binary vector format 1-43
- block coding
 - features 1-42
 - methods supported in blockset 1-42
 - techniques for 1-42
 - terminology and notation 1-43
- block coding library 1-41
- block interleaving library 1-72

C

- Channel Coding library 1-41
- Channels library 1-113
- code generator matrices
 - representing 1-50
- codebooks
 - representing 1-31
- codewords
 - definition 1-43
 - representing 1-43
- column vector signals 1-3
- Comm Sinks library 1-23
- Comm Sources library 1-7

- companders 1-38
 - example 1-38
- compression of data 1-30
- compressors 1-38
 - example 1-38
- converting analog to digital 1-30
- convolutional coding
 - delays 1-56
- convolutional coding library 1-54
- convolutional interleaving library 1-75

D

- data compression 1-30
- decision timing
 - and eye diagrams 1-25
 - and scatter diagrams 1-25
- delays
 - from analog demodulator's filter 1-90
 - in convolutional coding 1-56
 - in digital modulation 1-97
 - in example model 1-64
 - in interleaving 1-77
 - in serial-signal channel coding 1-44
- delta modulation 1-35
 - example 1-36
 - parameters 1-31
- demodulation 1-81
- diagrams
 - example 1-26
 - eye 1-24
 - scatter 1-25
- differential pulse code modulation (DPCM) 1-35
 - example 1-36
 - parameters 1-31
- digital modulation libraries 1-92

DPCM 1-35

- example 1-36
- parameters 1-31

E

- erasure insertion 3-5
- error, predictive 1-35
- error-correction capability
 - of Hamming codes 1-50
 - of Reed-Solomon codes 1-52
- examples
 - analog modulation timing 1-83
 - companders 1-38
 - continuous phase modulation 1-107
 - convolutional coding 1-56
 - convolutional decoding 1-60
 - convolutional interleaving 1-78
 - delays from filtering 1-90
 - delays in digital demodulation 1-99
 - differential pulse code modulation 1-36
 - eye and scatter diagrams 1-26
 - fading channels 1-115
 - filter cutoffs 1-87
 - Hamming coding 1-46
 - passband digital modulation 1-109
 - quantization 1-32
 - quantized sine wave 1-33
 - Reed-Solomon coding 1-47
 - scatter diagrams 1-105
 - signal constellations 1-104
- expanders 1-38
 - example 1-38
- eye diagrams 1-24
 - example 1-26

F

- filters, post-demodulation
 - choosing cutoff frequency 1-87
 - designing 1-87
 - resulting time lag 1-90
- frame attribute 1-4
- frame-based signals, definition of 1-4
- full matrix signal, definition of 1-3

G

- generator matrices 1-50

H

- Hamming codes 1-50

I

- integer format for messages and codewords 1-45
- interleaving delays 1-77
- Interleaving library 1-72

L

- linear predictors 1-36

M

- messages
 - definition 1-43
 - representing 1-43
- modulation
 - analog 1-81
 - definition of 1-81
 - delta 1-35
 - example 1-36

differential pulse code. *See* differential pulse
code modulation
digital 1-92
Modulation library 1-81
mu-law companders 1-38
example 1-38

N

nonbinary codes 1-42
Reed-Solomon 1-52

O

one-dimensional arrays, definition of 1-3
order of digits in binary numbers 1-45
orientations, of vector signals 1-3

P

$\pi/4$ DQPSK modulation 1-104
partitions 1-31
passband simulation 1-82
compared to baseband 2-24
phase modulation (PM) example 1-90
 $\pi/4$ DQPSK modulation 1-104
PLLs 1-130
predictive error 1-35
predictive quantization 1-35
example 1-36
features 1-30
parameters 1-31
predictors 1-35

Q

quantization

coding 1-34
example 1-32
features 1-30
parameters 1-31
predictive 1-35
example 1-36
vector 1-30

R

random signals 1-8
randseed 1-20
representing
codebooks 1-31
codewords 1-43
generator matrices 1-50
messages 1-43
partitions 1-31
predictors 1-36
quantization parameters 1-31
truth tables 1-50
row vector signals 1-3

S

sample times
in sources 1-18
sample-based signals, definition of 1-4
scalar quantization
coding 1-34
example 1-32
features 1-30
parameters 1-31
scalar signals
definition of 1-3
scatter diagrams 1-25
example 1-26

- signal formatting features 1-30
- sinks library 1-23
- sizes, of matrix signals 1-3
- source coding features 1-30
- Source Coding library 1-30
- sources library 1-7
- synchronization 1-130
- Synchronization library 1-130

T

- timing, decision
 - and eye diagrams 1-25
 - and scatter diagrams 1-25
- truth tables 1-50

V

- vector quantization 1-30
- vector signals, definition of 1-3